



UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Management of large geospatial datasets

Ka Hin Lau

INF-3990 Master's Thesis in Computer Science - May 2022

Abstract

In large simulations, like predicting the movement of ocean particles, it is common that simulation executions are related when they share one or more inputs. When the number of simulations increases, it becomes harder for users who run the simulations to keep track of all the simulations. Also, more storage spaces are wasted if there are multiple copies of the same input files.

This thesis describes a system that collects data from previous simulations, allowing users to search for the data they need to run the next simulation. Also, the system identifies the same files that were used in previous simulations, which allows users to re-use these files instead of copying the files to a new simulation folder to use them.

Among the simulations that were executed in our current environment, the system identifies around 11% of input files that are shared by the simulations. Users can refer to the same file to use it instead of copying the file to new simulation folders.

The conclusion is that the system helps users who run simulations to reduce their efforts and time to find input files that are used in previous simulations when they set up for a new simulation. Also, it saves storage space on the computing cluster where the simulations run on by identifying the duplicated data.

Acknowledgements

I would like to thank both of my supervisors, Professor Jonas Juselius and Professor Lars Ailo Bongo. Thank you Jonas for helping me to define the thesis scope, providing a lot of solid advice and wonderful guidance, and many precious ideas and insights on the thesis. Thank you Lars for giving me a lot of constructive feed-backs on the thesis and many valuable comments to improve the thesis. I am very grateful to have you two as my supervisors.

Also, I would like to thank my family and my friends for your support.

Contents

Abstract	i
Acknowledgements	iii
List of Figures	vii
1 Introduction	1
1.1 Related work	5
1.2 Outline	6
2 Relationships tracking	7
2.1 Introduction	7
2.2 Preparation of simulation	8
2.3 Design	9
2.3.1 Knowledge Graph	9
2.3.2 Parser	11
2.4 Implementation	12
2.4.1 Knowledge Graph	12
2.4.2 Parser	13
3 Centralized Data Storage	17
3.1 Introduction	17
3.2 Design	18
3.2.1 Identify the same files when storing files	18
3.2.2 Data Storage Directory Structure	20
3.3 Use files in centralized data storage	21
3.4 Snapshot of the Simulation Folder	22
4 Domain Driven Design in MdMP	25
4.1 Introduction	25
4.2 Motivation	26
4.3 Design	26
4.3.1 Data Transfer Object (DTO)	26
4.3.2 Domain Object Type	29

4.4	Implementation	29
4.4.1	Conversion to DTO	29
4.4.2	Conversion to Domain Object Type	31
5	Dependency Management in MdMP	33
5.1	Introduction	33
5.2	Motivation	34
5.3	Design	35
5.4	Implementation	40
5.4.1	Instruction as Functor	40
5.4.2	Program as Free Monad	42
5.4.3	Interpreter	42
6	Evaluation	45
6.1	Evaluation	45
7	Conclusion	47
7.1	Lesson Learned	47
7.2	Limitation	48
7.3	Future Work	48
	Bibliography	49

List of Figures

1.1	An illustration of how the inputs and outputs are related between simulations. The dotted line from output to input indicate that the output is used as input for another simulation, while the solid line from input to output indicate that the input produce the corresponding output.	2
1.2	An illustration about the process how MdMp creates the knowledge graph and stores the files in the centralized data storage.	3
1.3	An illustration about the four approaches implemented in MdMp.	5
2.1	This diagram shows the overview about the steps of running a simulation.	8
2.2	The diagram shows the file and simulation node with their own properties	9
2.3	The diagram shows the relationships between the file node and the simulation node.	10
2.4	An example showing the knowledge graph of one simulation created in Neo4j. The node with grey color represents the simulation, and the nodes with red color represents the file node. The arrows with the color black, green, blue and pink indicate the "HAS_Input", "HAS_OUTPUT", "HAS_INPUT_CONFIG" and "HAS_Tree" relationship between the simulation node and the file node respectively.	12
2.5	An example showing the knowledge graph of two simulations. Both of the simulations share some of the same input, and the second simulation use one of the output from the first simulation as input.	13
2.6	An example showing the metadata we need to extract from the configuration file, including the title, input and output file path, and various input files. Note that the 'none' filename indicated that section is not being used in the simulation. . .	14
2.7	An example showing the "&NML_SURFACE_FORCING" section containing the wind filename.	15

2.8	An example showing the successful result as a string array containing the metadata in different sections.	16
3.1	An example of the new configuration file after replacing the original input file name with the new input file name.	19
3.2	An example of how the new filenames look like with the checksum. Note that the checksum of the configuration file and the output file are the same.	19
3.3	This diagram shows the decision flow of storing files to the data storage.	20
3.4	This diagram illustrate the folder structure in the data storage.	21
3.5	An example showing that user need to copy the same input file A and input file B from simulation A folder to simulation B folder.	22
3.6	An example of showing the symbolic link files in the snapshot simulation directory pointing to the actual files in the data storage.	23
3.7	An example of the name format of the snapshot of the simulation directory.	24
4.1	The diagram shows the conversion from the content in the configuration file to the DTO type.	27
4.2	The diagram shows the process of saving data from MdMp to the database.	28
4.3	The diagram shows the process of converting data from the database to our DTO type.	28
4.4	The figure showing the conversion from string to DTO type by calling the "toDto" function.	30
4.5	The figure showing the "toDto" function in the "IOInput" module.	30
4.6	The figure shows the "NodeDto" union type with the 5 cases.	31
4.7	The figure shows the "toDomain" function.	31
4.8	The figure shows an example of the validation when creating the Domain Object.	32
5.1	The figure shows the impure dependencies in our program. .	36
5.2	The figure shows an example of normal functions without creating instructions to delay the impure actions.	37
5.3	The figure shows an example of how we create the instructions with a "next" function to delay the impure actions. . . .	37
5.4	The diagram shows an example of adding the "stop" instruction to the instruction set.	38
5.5	The diagram shows an example of the approach of using an interpreter to execute the instructions.	39

5.6	The diagram shows an example of the approach of using an interpreter to execute the instructions.	40
5.7	The figure shows an example of the "createNodes" instruction to create nodes in the database.	41
5.8	The figure shows an example of the "CopyDirectoryDecision" decision that contains two instructions to be execute.	41
5.9	The diagram shows the "Program" with the "Free" part and the "Pure" part.	42
5.10	The diagram shows the interpreter. It recursively execute the instructions until a "stop" instruction	43



Introduction

Large simulations can provide better insights and directions for future large-scale and long-term planning. For instance, a weather simulation allows better preparation for the impacts of climate changes, and a simulation on predicting the movement of ocean particles enables exploiting ocean resources more efficiently. However, executing these large simulations requires a lot of energy and money for CPU power and storage space. Often, simulation executions are related because the inputs are shared by many executions, or the outputs from a previous simulation are used as inputs in the next simulation (Figure 1.1). Generally, to prepare for a simulation, a new folder is created to store the inputs it requires and the output it produced. However, these dependencies are usually not maintained, so resources are wasted in terms of preparation time for users to gather the essential inputs of the simulations, and the storage spaces for the redundant inputs. In particular, the following problems arise when the number of simulations increases:

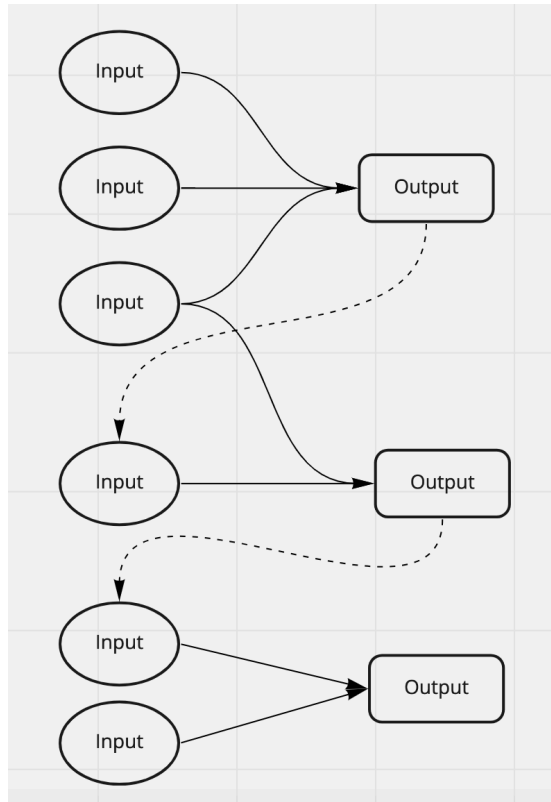


Figure 1.1: An illustration of how the inputs and outputs are related between simulations. The dotted line from output to input indicate that the output is used as input for another simulation, while the solid line from input to output indicate that the input produce the corresponding output.

1. **Longer time for users to set up a new simulation:** The required input files are usually scattered over many different folders, so it takes longer time for users to gather all these input files from different locations to set up the next simulation.
2. **Inputs duplication:** Input files are copied from the previous simulation folders to the new simulation folder. Hence, the same input files can be duplicated multiple times and therefore takes up unnecessary storage space.

To address the above problems, we propose the Metadata Management Program (MdMP). By managing the input and output data between simulations,

MdMp reduces the time and efforts required to prepare files for a simulation, and it saves storage spaces by re-using the same inputs using the following approaches:

1. To reduce users' efforts and time to prepare files for a simulation, MdMp, tracks the relationships of the input and output data between simulations by creating a knowledge graph [1]. It provides users with a set of commands on the command line tools to search for inputs and outputs in previous simulations that are stored in the knowledge graph to use as inputs for the next simulation .
2. To avoid redundant copies of files in simulation folders, MdMp stores the files in a centralized storage and identifies the files with the same content to save storage spaces.

To create the knowledge graph and store the files in the centralized data storage, MdMP needs to obtain the information about the files and their metadata in the simulation folders. By running MdMP in a simulation folder after the simulation is completed, it gets the information about the files and their metadata in the simulation, then store them in the centralized data storage and knowledge graph respectively.

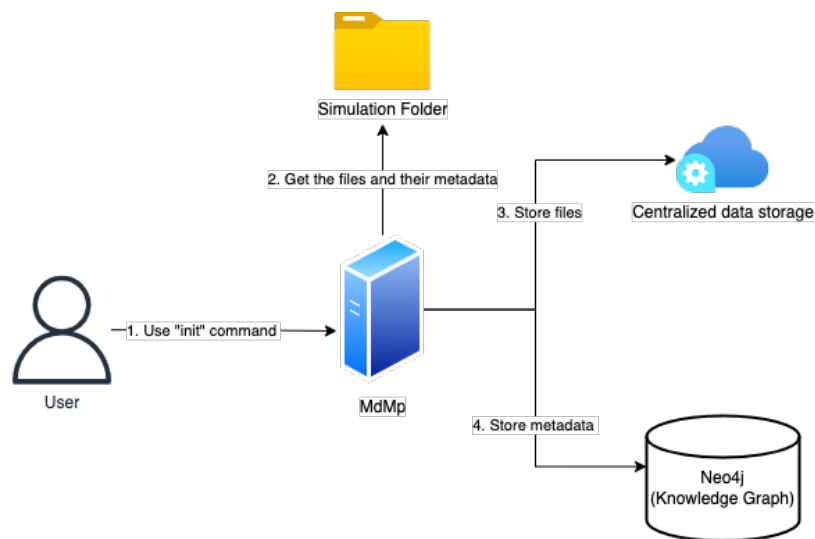


Figure 1.2: An illustration about the process how MdMp creates the knowledge graph and stores the files in the centralized data storage.

So, when users prepare for the next simulation, they can use MdMp to search for the files by the relationships that are stored in the knowledge graph, and use files in the centralized data storage.

When we were developing MdMP, we encountered two major challenges. The first one is about ensuring the data validity in MdMp. If users receive invalid or corrupted data from MdMp to run a simulation, they need to re-run the simulation. It is costly to re-run the simulation, which may take up to a few weeks. To solve this challenge, we implemented DDD (Domain Driven Design) [19] [12] to ensure the data validity in MdMp. The second challenge is managing the dependencies in MdMp with different services, such as the database service for creating the knowledge graph and the File IO service for the centralized data storage. If the dependencies are not managed properly, it is difficult to add new services, or replace the current services with other services in the future. To overcome this challenge, we use Free Monad [6] approach to manage the dependencies.

So, there are four main approaches adopted in MdMp, which provide benefits to different types of users. The approach of creating knowledge graph benefits the users who run the simulations, helping them to search for files through the relationships between simulations. The approach of storing files to the centralized data storage benefits the system managers who manage the cluster that the simulations run on, helping them to save storage spaces by not having redundant copies of files in simulation folders. The approach of implementing DDD lowers the chances of re-running simulation due to invalid data in MdMp. Finally, the approach of using Free Monad helps the system managers and developers to add or switch out services easily.

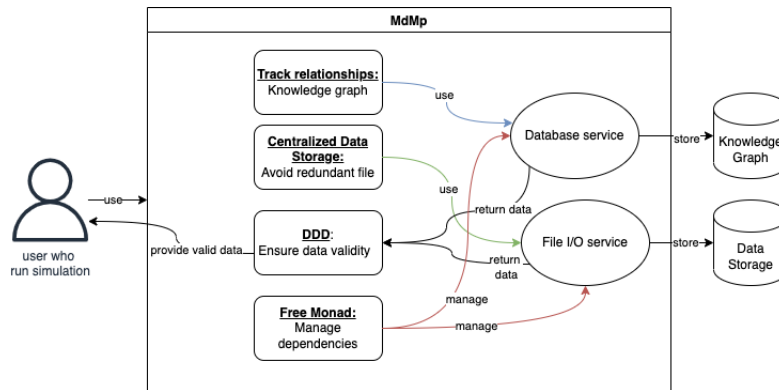


Figure 1.3: An illustration about the four approaches implemented in MdMp.

One of the most important simulation models we used on our HPC (High-Performance Computing) cluster is FVCOM (The Finite-Volume, Community Ocean Model) [8]. FVCOM is used to simulate the circulation and ecosystem dynamics from global to specific regions characterized by irregular complex coastlines, islands, inlets, creeks, and inter-tidal zones. It is an integrated high-resolution model system that is capable of nowcasts, hindcasts, and forecasts of circulation and key ecosystem processes in estuaries and coastal oceans, and it is widely used by private companies, government agencies and scientists at academic universities and institutions in the world. MdMp supports obtaining information from a simulation using FVCOM by extracting metadata of files used in the simulation, which is stored in the configuration file inside the simulation folder. MdMp stores the metadata of the files to construct a knowledge graph to track the data relationships between simulations, allowing users to search for data used in previous simulations and use the data for a new simulation.

1.1 Related work

To allow users to search for files in previous simulations to use them as inputs for a new simulation, we create knowledge graph to track the data between

simulations. Google Search [10] also utilizes a knowledge graph to provide more relevant results for users' search.

To store files and their metadata to the centralized data storage and knowledge graph from a simulation folder after the simulation, users need to run an "init" command to let MdMp achieve that. It is similar to running a "git init" command in a folder to let Git [2] to track the files in that folder. Our approaches to checking whether two files share the same content, and the folder structure of the centralized data storage are also closely tied to the Git file-system.

1.2 Outline

The outline is organized as follows. Chapter 2 discusses the tracking of data relationships between simulations in MdMp. Chapter 3 discusses the usage of centralized data storage. Chapter 4 covers the application of the DDD in MdMp. Chapter 5 covers dependency management using Free Monad in MdMp. Chapter 6 then provides the evaluation of MdMp. Finally, we conclude and discuss possible future works in Chapter 7.

/2

Relationships tracking

2.1 Introduction

In this chapter, we describe the solution implemented in the Metadata Management Program (MdMp) for tracking the relationships of input and output data between simulations. MdMp implements tracking for the simulation model FVCOM (The Finite-Volume, Community Ocean Model). To track the relationships between simulations, we first need to know what files are used as input, and what output files are generated in a simulation. The file paths of these files are specified in the configuration file during the simulation preparation phase. Thus, we created a standalone parser [3] library to read the content of the configuration file and extract the files. MdMp uses the parser library as an external service because we want to separate the implementation details of the parser and MdMp such that adding new implementations to the parser with other simulation models in the future would not affect the implementation of MdMp. After reading the contents of the configuration file, MdMp tracks the inputs and outputs by storing them in a knowledge graph.

This chapter starts with an overview of the preparation process of a simulation using FVCOM (Section 2.2). We then discuss the design (Section 2.3) and our implementation of tracking the relationships of inputs and outputs with the help of the parser and the knowledge graph in MdMp (Section 2.4).

2.2 Preparation of simulation

To prepare for a simulation, the steps are general, but FVCOM is used as an example throughout the chapter. Below are the steps that users need to take:

1. Create a simulation folder that contains all the input files, an configuration file and an output directory. The configuration file contains information about the file paths of the input and output files, along with various configuration parameters for the simulation, such as the number of rivers, the units of the grid and the type of heat.
2. Specify all the parameters in the configuration file, including the input and output file directory, the input file paths, and other miscellaneous attributes for different types of input.

After performing the above steps, we can start executing the simulation by running the corresponding commands provided by FVCOM. Once the simulation finishes, the output files are generated in the output file directory specified in the configuration file.

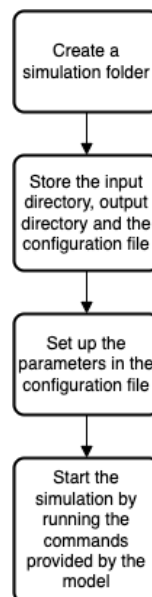


Figure 2.1: This diagram shows the overview about the steps of running a simulation.

2.3 Design

2.3.1 Knowledge Graph

To track the input and output data relationships between simulations, MdMp stores the relationships and the metadata in a knowledge graph using Neo4j graph database [18]. In Neo4j, the "Node" type and the "Relationship" type represent the data and the relationship between the data, respectively. There are two major "Node" types created in the graph database:

1. **File Nodes** represents a file that is used in a simulation. The file can be an input file, an output file and a configuration file. There are three major properties in the File Node: checksum, file format and file name. The checksum is a SHA-1 check-sum [2] used as part of the key to identify the file node.
2. **Simulation Node** represents a simulation. It only has one property: checksum; this SHA-1 checksum value is generated based on the SHA-1 check-sums of the input files in that simulation. It represents a unique set of input files.

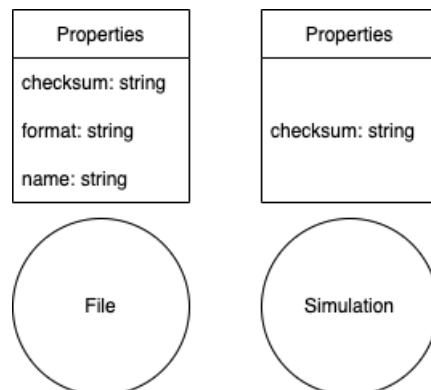


Figure 2.2: The diagram shows the file and simulation node with their own properties

Moreover, we define four major relationships between the file and simulation nodes:

1. **HAS_INPUT**: The "HAS_INPUT" relationship links the input file to the simulation. It has a property "Type" to indicate the configuration type of the input file in the simulation. The "Type" value is extracted when parsing the configuration file.
2. **HAS_INPUT_CONFIG**: The "HAS_INPUT_CONFIG" relationship links the configuration file to the simulation. It does not have any property.
3. **HAS_OUTPUT**: The "HAS_OUTPUT" relationship links the output file to the simulation. It does not contain any property.
4. **HAS_TREE**: The "HAS_TREE" relationship links the tree commit file to the simulation. It does not contain any property. The tree commit file stores the information about the related files in the simulation, which is used as a backup to keep the information about the related files in the simulation in case the graph database is down.

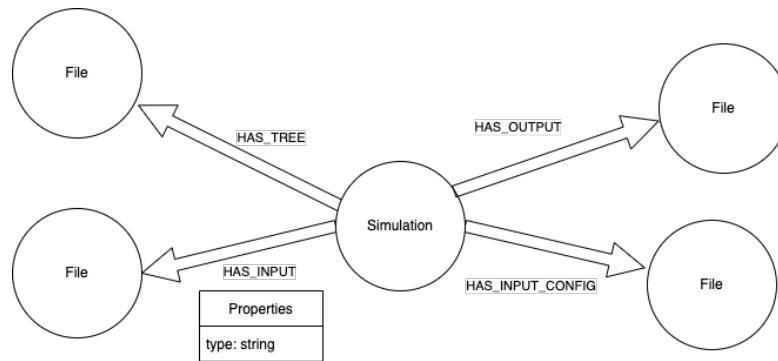


Figure 2.3: The diagram shows the relationships between the file node and the simulation node.

In the Neo4j graph database, the CQL (Cypher Query Language) [18] is introduced. It is used to communicate and to perform operations on the

database. The basic operations include inserting, searching, updating, and deleting (CRUD) records in the database. CQL works like SQL (Structured Query Language) [16] for graphs, while SQL is used in traditional relational databases. It is faster and easier to use CQL to retrieve records that are related to other records because the relationships are already stored in the graph database; while for SQL we need to perform different "join" operations to obtain all the related data. In MdMp, CQL is used to perform the CRUD operations with the help of a .NET client library, Neo4jClient [14], to translate F# [7] code into Cypher.

2.3.2 Parser

To obtain the input and output files in a completed simulation, we need to read the content of the configuration file. We created our own parser for parsing the configuration file of FVCOM. It is built as a standalone library to be plugged into the MdMP. If we want to use another simulation model such as OpenDrift [11], we can add the parsing logic implementation to the parser library. The advantages of creating the parser as a standalone library are:

1. **Loose Coupling:** We do not need to recompile the main program if there are any new updates or changes in the parser, vice versa.
2. **Parallel development:** The main program MdMp and the parser can be developed in parallel since the implementation of the parser is independent of the main program.

The input of the parser is the configuration file for a simulation. The parser separates the content of the configuration file into different sections so that we can extract the metadata from each section later. The output of the parser is an "Result" discriminated union type [5]. The discriminated unions type contains cases with different values and types, the value can be one of named cases. If any errors occur during the parsing process, the "Result" will be an "Error" case that contains the error message. Otherwise, if the parsing succeeds, the "Result" will be an "Ok" case which contains a string array representing the sections.

2.4 Implementation

2.4.1 Knowledge Graph

To construct a knowledge graph in Neo4j graph database, MdMp needs to set up a connection to Neo4j. We use Neo4jClient, a .NET client library, to set up the connection. It also helps to translate F# code into CQL to perform CRUD operations on the records. We created two "Create" operations in MdMp, one for creating the file nodes and one for creating the relationships between the file nodes and the simulation node. MdMp also provides several "Read" operations to query for the data, and a "Delete" operation to delete the nodes.

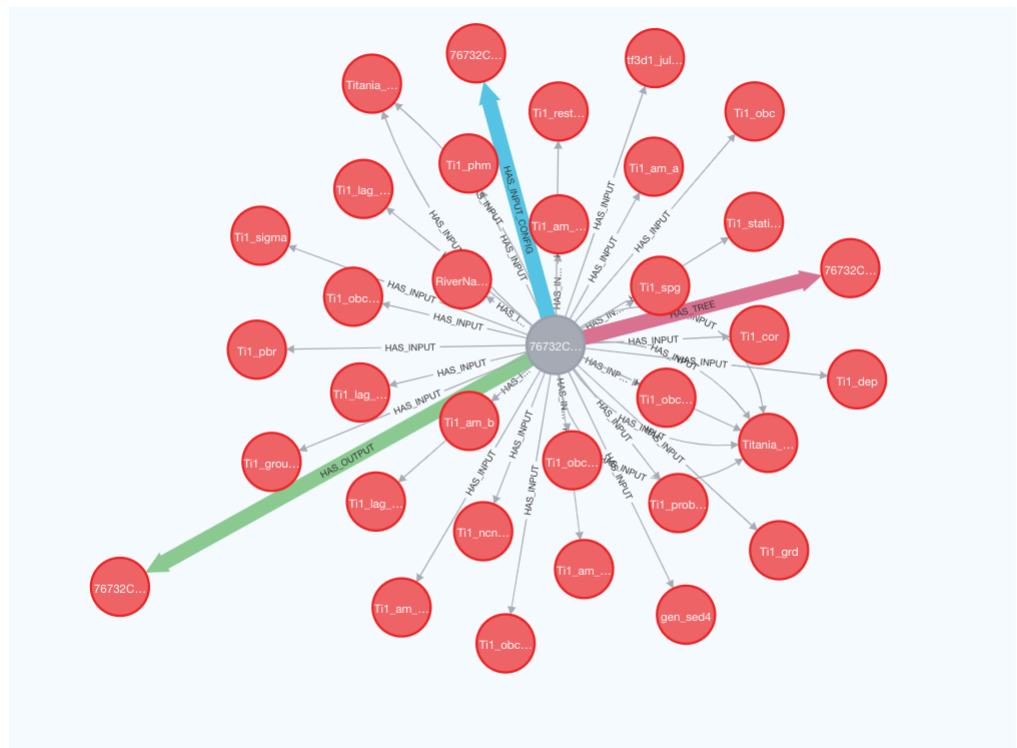


Figure 2.4: An example showing the knowledge graph of one simulation created in Neo4j. The node with grey color represents the simulation, and the nodes with red color represents the file node. The arrows with the color black, green, blue and pink indicate the "HAS_Input", "HAS_OUTPUT", "HAS_INPUT_CONFIG" and "HAS_Tree" relationship between the simulation node and the file node respectively.

When two simulations share some of the same inputs, they are related by

current directories of the input and output files, and the filenames of these files. The title of the simulation is denoted by the key "CASE_TITLE" under the "&NML_CASE" section in the configuration file. The directories of the input and output files are specified in the section "&NML_IO". The input file directory is denoted by the key "INPUT_DIR", while the output file directory is indicated by the "OUTPUT_DIR" (Figure 2.6).

```

=====
|                   |
|   F V C O M       |
|   P W L W I N G   |
|   -- Beta Release  |
|                   |
|=====
|
|=====DOMAIN DECOMPOSITION USING: METIS 4.0.1 =====
|=====Copyright 1998, Regents of University of Minnesota=====
|
|&NML_CASE
|CASE_TITLE      = 'Titania',
|TIMEZONE        = 'UTC',
|DATE_FORMAT     = 'YMD',
|DATE_REFERENCE  = 'default',
|START_DATE      = '2017-10-01 00:00:00',
|END_DATE        = '2018-09-26 00:00:00',
|/
|
|&NML_STARTUP
|STARTUP_TYPE    = 'hotstart',
|STARTUP_FILE    = 'T11_restart_1001.nc',
|STARTUP_UV_TYPE = 'set values',
|STARTUP_TURB_TYPE = 'set values',
|STARTUP_TS_TYPE = 'set values',
|STARTUP_T_VALS  = 0.0,
|STARTUP_S_VALS  = 34.5,
|STARTUP_U_VALS  = 0.00000,
|STARTUP_V_VALS  = 0.00000,
|STARTUP_DMAX    = 0.00000,
|/
|
|&NML_STARTUPX
|STARTUP_TYPE    = 'coldstart',
|STARTUP_FILE    = 'none',
|STARTUP_UV_TYPE = 'default',
|STARTUP_TURB_TYPE = 'default',
|STARTUP_TS_TYPE = 'constant',
|STARTUP_T_VALS  = 6,
|STARTUP_S_VALS  = 35,
|STARTUP_U_VALS  = 0.00000,
|STARTUP_V_VALS  = 0.00000,
|STARTUP_DMAX    = -600.00000,
|/
|
|&NML_IO
|INPUT_DIR       = './input/',
|OUTPUT_DIR      = './output/',
|IREPORT         = 100,
|VISIT_ALL_VARS  = F,
|WAIT_FOR_VISIT = F,
|USE_MPI_IO_MODE = F,
|/

```

Figure 2.6: An example showing the metadata we need to extract from the configuration file, including the title, input and output file path, and various input files. Note that the 'none' filename indicated that section is not being used in the simulation.

There are different types of input used in FVCOM, such as wind, heating, river, etc. The filenames of the input files are specified in their corresponding sections. For instance, to get the input file for wind, we check the key "WIND_FILE" in the "&NML_SURFACE_FORCING" section (Figure ??).

```
,
&NML_SURFACE_FORCING
WIND_ON = T,
WIND_TYPE = 'speed'
WIND_FILE = 'Titania_wnd_all.nc'
WIND_KIND = 'variable'
WIND_X = 0.000000E+00,
WIND_Y = 0.000000E+00,
HEATING_ON = F,
HEATING_TYPE = 'flux'
HEATING_KIND = 'variable'
HEATING_FILE = 'Titania_nest_allT.nc'
HEATING_LONGWAVE_LENGTHSCALE = 1.400000
HEATING_LONGWAVE_PERCTAGE = 0.780000
HEATING_SHORTWAVE_LENGTHSCALE = 6.300000
HEATING_RADIATION = 0.000000E+00,
HEATING_NETFLUX = 0.000000E+00,
PRECIPITATION_ON = T
```




Figure 2.7: An example showing the "&NML_SURFACE_FORCING" section containing the wind filename.

To ensure that MdMp extracts the metadata correctly in each section according to its section header, the parser separates the sections into a string array by checking the content that begins with the format "NML_XXX" and ends with a newline with a "/" character followed by a empty line. If the parsing fails, an error message wrapped in "Result" type with "Error" case will be returned. Otherwise, if the parsing succeeds, a string array wrapped in "Result" type with "Ok" case will be returned. Each item in the string array contains one section of the configuration file (Figure 2.8).

```

[&NML_CASE
CASE_TITLE      = 'Titania',
TIMEZONE       = 'UTC',
DATE_FORMAT    = 'YMD',
DATE_REFERENCE  = 'default',
START_DATE     = '2017-10-01 00:00:00',
END_DATE       = '2018-09-26 00:00:00',
;

&NML_STARTUP
STARTUP_TYPE   = 'hotstart',
STARTUP_FILE   = 'Ti1_restart_1001.nc',
STARTUP_UV_TYPE = 'set values',
STARTUP_TURB_TYPE = 'set values',
STARTUP_TS_TYPE = 'set values',
STARTUP_T_VALS = 0.0,
STARTUP_S_VALS = 34.5,
STARTUP_U_VALS = 0.00000 ,
STARTUP_V_VALS = 0.00000 ,
STARTUP_DMAX   = 0.00000
;

&NML_STARTUPX
STARTUP_TYPE   = 'coldstart',
STARTUP_FILE   = 'none',
STARTUP_UV_TYPE = 'default',
STARTUP_TURB_TYPE = 'default',|
STARTUP_TS_TYPE = 'constant',
STARTUP_T_VALS = 6 ,
STARTUP_S_VALS = 35 ,
STARTUP_U_VALS = 0.00000 ,
STARTUP_V_VALS = 0.00000 ,
STARTUP_DMAX   = -600.00000
; ... ]

```

Figure 2.8: An example showing the successful result as a string array containing the metadata in different sections.

Once we have the directories and the filenames, we can combine the directory and filename to get the file path. Then we check whether the file exists in the file path. If not, we will prompt a message to the user about the invalid file path and stop the program. Occasionally, the filename can be marked as "none" in the configuration file, indicating that this input is not needed. In this case, we will filter the file marked with "none" and proceed to the next step, but we will still prompt a message about the unused input. An example of the combined source file path is "/Users/miclo/MetaServer/Cli/input/Titania_wnd_all.nc"

/3

Centralized Data Storage

3.1 Introduction

A new simulation can use some of the same inputs and/or outputs from previous simulations as inputs. In such case, the same copies of the input files can be found in different simulation folders. To reduce the disk spaces occupied by duplicate copies, we store all files in a centralized data storage, then users can create symbolic link [13] files pointing to the actual files in the centralized data storage instead of duplicating the same files. MdMp provides several commands for users to search for the related files to obtain the file paths in the centralized data storage. Moreover, to avoid having duplicated copies of the same file in the centralized data storage, we base the filenames on the content of the file by generating a SHA-1 checksum for each file.

To prevent losing the information of the original simulation folder, we create a snapshot of the simulation folder, which contains the configuration file, the input files and the output files. These files are also symbolic link files that point to the actual files stored in the centralized data storage.

This chapter starts with the design (Section 3.2) of storing files to the centralized data storage, and the folder structure of the centralized data storage. We then present how users can refer to the files in the centralized data storage instead of copying the same files to a new simulation folder (Section 3.3). Next, we discuss the snapshot of the simulation folder (Section 3.4).

3.2 Design

3.2.1 Identify the same files when storing files

There are three kinds of files that MdMp stores in the centralized data storage: **input file**, **output file** and the **configuration file**. To avoid storing the same file more than once in the centralized data storage, our initial approach is to generate a SHA-1 checksum of a file's content, then use it as the filename to identify the file, which is similar to how Git stores files. However, it takes a lot of time to compute the SHA-1 checksum of a file with large content. So, our solution is: First, for each of the input files, we generate a SHA-1 checksum of its file content. The size of input files are small, so we do not have the problem of the SHA-1 checksum computation time. Next, we generate a SHA-1 checksum from all the input file SHA-1 check-sums generated in the previous step. This set of input file SHA-1 check-sums uniquely identifies the simulation that produces the corresponding output files. Since the configuration file contains all the input file names, we replace every input file name with the corresponding SHA-1 checksum of each input file generated in the previous step (Figure 3.1), and then generate a SHA-1 checksum of this transformed configuration file. This SHA-1 checksum is used by the configuration file and all the output files. To distinguish between the configuration file and the output file, and between the output files, we standardized the filename format of each file to "[SHA-1 checksum]-[custom name]", where the custom name is the original filename (Figure 3.2).

Finally, the process of storing a file to the centralized data storage is: we first check whether the filename contains a SHA-1 checksum or not. If the filename **DOES NOT** contain a SHA-1 checksum, we generate a SHA-1 checksum of the file content (For the configuration file and the output file, we use the SHA-1 checksum of the transformed configuration file instead) and use the new filename format "[SHA-1 checksum]-[custom name]" to store the file. Otherwise, if the filename **DOES** contain a SHA-1 checksum, we extract the filename. Next, we look up the filename in the centralized data storage. If it already exists, we do not store the file again. Otherwise, we store the file with the new filename containing the checksum and the custom name.

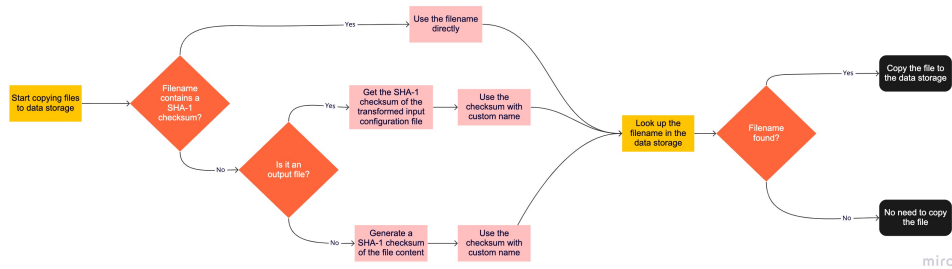


Figure 3.3: This diagram shows the decision flow of storing files to the data storage.

3.2.2 Data Storage Directory Structure

When we store files to the data storage, we create a directory to store all these files. Each file is stored under a directory with a directory name using the first two characters of the file's SHA-1 checksum, and the directory is stored under the the top-level directory (Figure 3.4). There are two main reasons why we extract the first two characters of the SHA-1 checksum as the directory:

1. **Provide faster file access time:** In some file systems such as the Ext* family, the structure of a directory is a linked list or a table of entries. To search for a file, the entire list is scanned until the matching file name is found, which is undesirable for performance. To provide faster access time, keeping each directory small is important.
2. **Keep the directory small:** Some file systems are only limited to 32000 entries in a single directory. In Linux kernel, the number of commits is within the order of that magnitude. By subdividing the commits using the first two hex digits, the top-level size is limited to 256 entries.

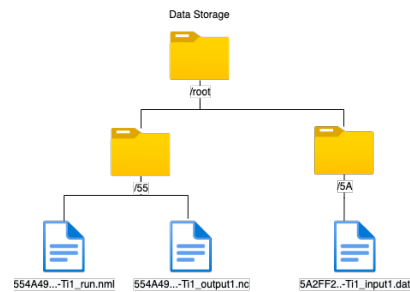


Figure 3.4: This diagram illustrate the folder structure in the data storage.

The design of the data storing structure is similar to the one implemented in Git. The only difference is that we have the full SHA-1 checksum and a custom name as the filename, instead of just using the rest of the SHA-1 checksum as a file name. There are two reasons behind the difference:

1. Both the output file and the transformed configuration file share the same SHA-1 checksum in a simulation, so we need the custom name to identify them.
2. We need to specify the filename of input files in the configuration file, so we need the full SHA-1 checksum in the filename. Also, it is possible that some output files from previous simulations are used as input files in the next simulation, so we also need the custom name to identify the output files.

3.3 Use files in centralized data storage

When users prepare for a new simulation, they may use some inputs and/or outputs from previous simulations as inputs for the new simulation. Without centralized data storage, they need to copy the same files to the new simulation folder to use them (Figure 3.5). To reduce disk spaces for storing multiple copies of the same files in different simulation folders, they can create symbolic link files instead. To create a symbolic link file, users need to know the file path that points to the actual files stored in the centralized data storage. To obtain the file path, MdMp provides several commands for users to search for the related file in the centralized data storage.

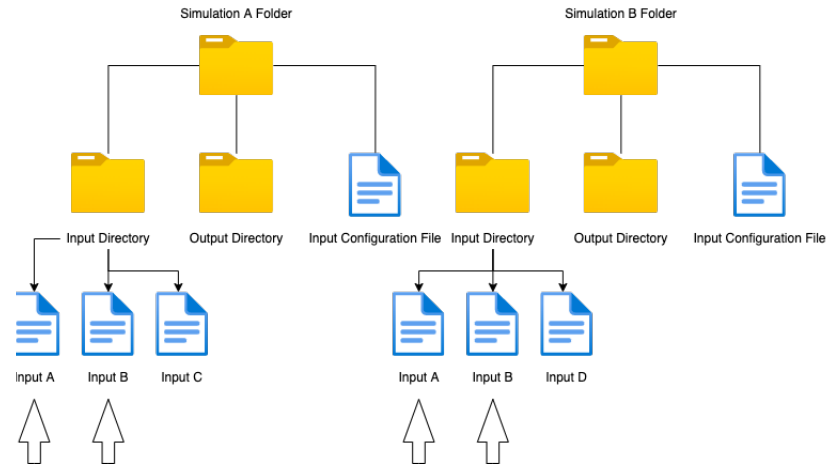


Figure 3.5: An example showing that user need to copy the same input file A and input file B from simulation A folder to simulation B folder.

After getting the file path in the centralized data storage, users can create the symbolic link file with it, then specify the filename in the configuration file in the new simulation folder for the simulation model to use it.

3.4 Snapshot of the Simulation Folder

We create a snapshot of the simulation folder as a backup to the origin simulation folder, which allows users to maintain the information about the simulation even if the files in the origin simulation folder are gone or corrupted. The information stored in the snapshot includes the configuration file, the input files and the output files. In order to save disk spaces, the files are symbolic link files that point to the actual files stored in the centralized data storage (Figure 3.6).

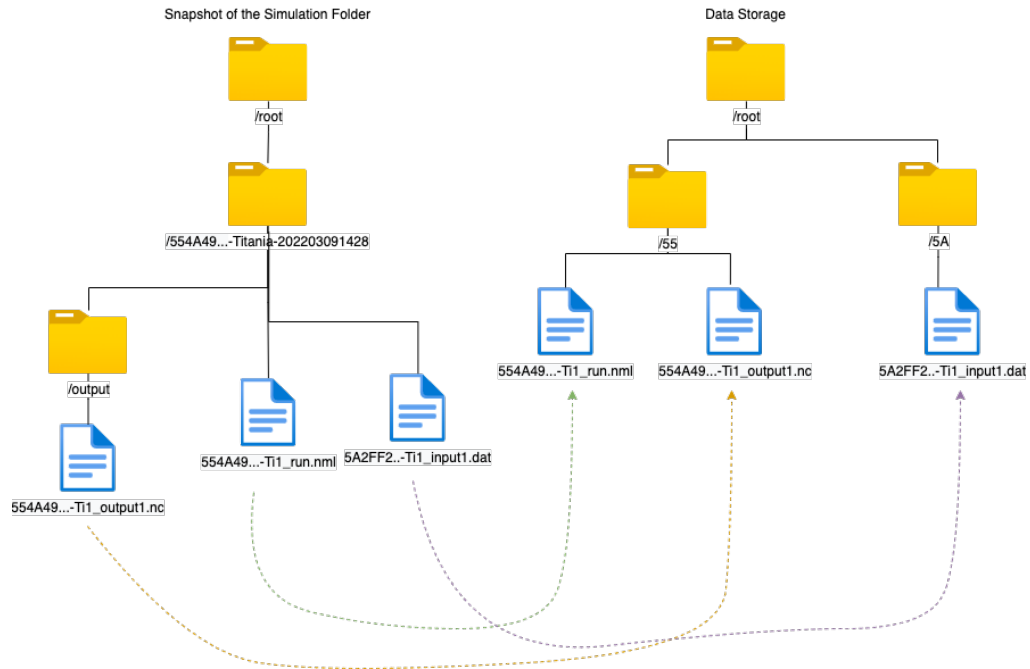


Figure 3.6: An example of showing the symbolic link files in the snapshot simulation directory pointing to the actual files in the data storage.

The name format of the snapshot of the simulation directory is [simulation SHA-1 checksum]-[case title]-[timestamp]. The simulation SHA-1 checksum is the same as the SHA-1 checksum of the transformed configuration file. The case title is the title of the simulation extracted from the configuration file. The timestamp is the current timestamp when creating the simulation directory (Figure 3.6).

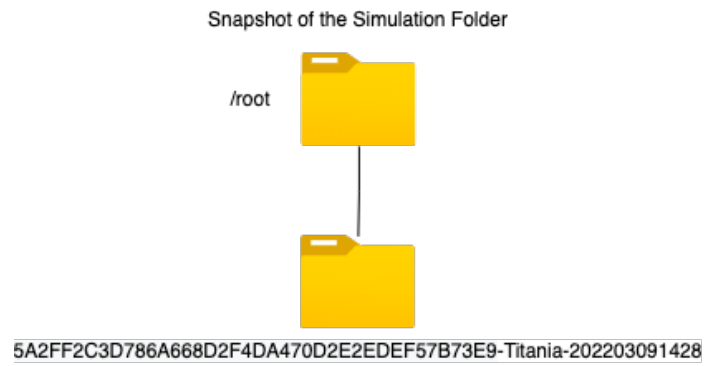


Figure 3.7: An example of the name format of the snapshot of the simulation directory.

/4

Domain Driven Design in MdMP

4.1 Introduction

In this chapter, we discuss the Domain-driven design (DDD) implemented in MdMp. DDD is a software design approach that focus on modelling the software to match the input from experts in each domain related to the project. A domain in DDD refers to an area of knowledge where people work and try to solve problems. There are several domains we need to understand to implement DDD in MdMp, including the simulation using FVCOM, the graph database to create knowledge graph. To provide solutions after understanding the problems in the domains, we build a solution space (domain model) that contains the relevant domains, and these domains are mapped to the DDD terminology - bounded contexts. Each bounded context defines its own data objects to represent the data in its own domain, and they are called domain objects in DDD. These data objects are passed around bounded contexts when they are needed in different bounded contexts. The data objects that are transferred between bounded contexts have different formats and they are called Data Transfer Objects (DTOs).

This chapter begins with the motivation of using DDD in MdMp (Section 4.2). We then discuss the design (Section 4.3) and our implementation on DDD with DTO and Domain Object Type (Section 4.4).

4.2 Motivation

When users prepare for a new simulation and use MdMp to look for information in previous simulations, it is possible that MdMp returns information that is no longer valid due to errors in the graph database or the centralized data storage, but users are not aware of that and still use the information, which can lead to a re-run of the simulation. The cost of re-running the simulation is high since the execution time can be up to a few weeks. DDD is implemented in MdMp to ensure data validity by providing a domain model.

There are two major advantages to implementing DDD in MdMp. The first one is that it creates a "Trust Boundary" between MdMp and everything outside MdMp, such as the database and the centralized data storage. The "Trust Boundary" ensures the data validity inside MdMp by implementing the validation at the boundary of the program to validate the data before it enters into MdMp. Therefore, when users search for files to use as inputs for a new simulation using MdMP, they can trust and use the information provided by MdMp. The second advantage is that DD makes MdMp self-documenting. DDD provides a "ubiquitous language" such that every party participating in the project, including the simulation users and the developers of the MdMp, can understand the logic and flow of the program better, which can save users' time to communicate with the developers when they need some adjustments or changes in the future. The data is wrapped into domain types defined by the users and the strict type system in F# helps to protrude the type label and to strengthen the type checking. The trade-off of using DDD is that it requires developers more effort to get the value when they work on the data because the data is wrapped in Domain Object types, so they need to unwrap the Domain object to get the value. However, we believe implementing DDD is worth the effort because the cost of re-running simulation is much higher.

4.3 Design

4.3.1 Data Transfer Object (DTO)

There are three scenarios when we convert the data into DTO type:

1. **When extracting data from the configuration file:** After parsing the configuration file, we get an array of string wrapped in "Result" union type. They are wrapped by the "Result" type because the parsing can fail due to invalid parameters. Once we have the valid result, we convert the result into our DTO type (Figure 4.1).

2. **When transferring data to the database:** When saving data to the database, we convert our data in MdMp from Domain Object Type to DTO type, and further serialize the DTO type into JSON string because the database receives data as JSON string (Figure 4.2).
3. **When transferring data from the database:** When we query data from the database, we de-serialize the JSON string retrieved from the database and then convert it into DTO type. The de-serialization may fail due to errors return from the database, so we wrap the DTO type in "Result" union type (Figure 4.3).

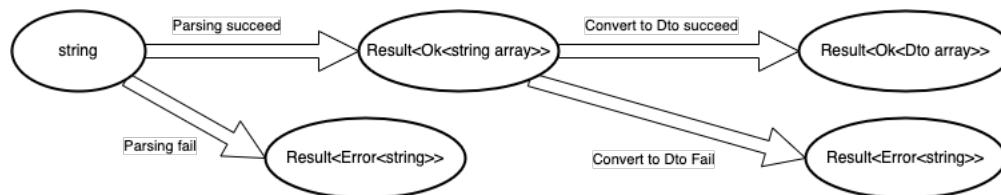


Figure 4.1: The diagram shows the conversion from the content in the configuration file to the DTO type.

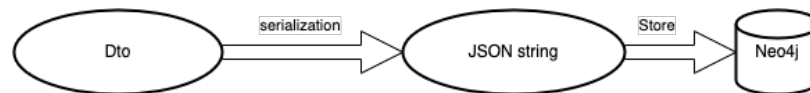


Figure 4.2: The diagram shows the process of saving data from MdmP to the database.

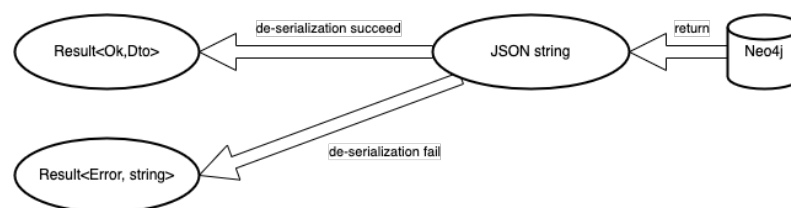


Figure 4.3: The diagram shows the process of converting data from the database to our DTO type.

4.3.2 Domain Object Type

In MdMp, we do not directly use the DTO type as the format to represent the data in our domains, we only use the DTO type to work as a bridge between our domain and the outside world. So, after we convert the data into DTO type, we further convert it into Domain Object Type.

During the conversion from DTO type to Domain Object Type, we put our validation rules and logic in when creating the Domain Object Type data. For example, when we create our "File" Domain Object Type data, we forbid the filename to be null.

Since the conversion from DTO type to our Domain Object type can fail due to the validation rules, we wrap the Domain Object type into "Result" union type. As a result, data that does not pass our validation rules is not created. Therefore, we ensure the data that is created in our domain is valid.

After we get the valid data, we wrap the data into our Domain Object type. For instance, we have a "File" Domain Object type, with three attributes - checksum, name, format. These attributes are all in the string type, but we create an extra new type for each of the attributes and a model for that type such that we can define our validation rules in the model. The extra type layer provides strict checking when passing the data around. When we manipulate the data, we pass the data around to different functions. With the explicit type, we do not need to worry about passing invalid data to the function. For example, if we have a function which accepts a "Filename" type data as the parameter to change the filename, we know the "Filename" type data already pass our domain logic and validation rules. But if the input parameter of the function is just a "string" type, we may receive a random string which does not pass our validation rule as a valid filename, then we need to check the validity of the data in every function that uses the data. When we convert the data from our Domain Object type to DTO type, we no longer need to wrap the DTO type in "Result" type because we know that every data in our Domain Object type is valid, so we can perform the conversion directly.

4.4 Implementation

4.4.1 Conversion to DTO

If the configuration file is valid, we have an array of strings. Each string in the array contains the information about a group of parameters for a setting. For example, the string marked with "&NML_IO" contains the information

about the input file path and the output file path, and the string marked with "&NML_SURFACE_FORCING" contains information about the wind input file along with other parameters such as wind type. To convert the string array into our DTO types, we create a module for each of the sections that we want to extract (Figure 4.4). In the module, there is a "toDto" function to convert the string to DTO type (Figure 4.5).

```
RegexTitle "&NML_IO\s" str ->
str: string
|> IOInput.toDto : Dto.Dto<Dto.NodeDto>
|> getResultArrayFromNodeDto
```

Figure 4.4: The figure showing the conversion from string to DTO type by calling the "toDto" function.

```
module IOInput =
    string -> Dto<NodeDto> & miclau13
    let toDto (str: string) =
        let InputDirectory = getProperty str "INPUT_DIR"
        let OutputDirectory = getProperty str "OUTPUT_DIR"
        let ConfigType = sprintf "%s-%s" "NML_IO" "IOInput"
        let result = Dto.IOInputDto {
            InputDirectory = InputDirectory
            OutputDirectory = OutputDirectory
            ConfigType = ConfigType
        }
        let dto: Dto.Dto<Dto.NodeDto> = {
            data = result
        }
        dto
```

Figure 4.5: The figure showing the "toDto" function in the "IOInput" module.

We define a discriminated union type named "NodeDto" that contains 5 cases of DTO records (Figure 4.6). With the help of the "NodeDto" union type, it is easier to pass the DTO records around to different functions.

```
type NodeDto =  
  | ConfigFileInputDto of ConfigFileInputDto  
  | FileDto of FileDto  
  | FVCOMInputDto of FVCOMInputDto  
  | IOInputDto of IOInputDto  
  | SimulationDto of SimulationDto
```

Figure 4.6: The figure shows the "NodeDto" union type with the 5 cases.

4.4.2 Conversion to Domain Object Type

After converting the string to DTO type, we further convert it into our Domain Object Type. We create a "toDomain" function to handle all the 5 DTO types conversion (Figure 4.7), and the validation rules for the conversion are implemented in the function (Figure 4.8). Then the result is wrapped in "Result" because the conversion may fail due to the invalid data defined by our Domain logic.

```
let toDomain (dto: Dto<NodeDto>) :Result<Node,string> =  
  let nodeDto = dto |> fromDto  
  match nodeDto with  
  | FileDto data ->  
    result {  
      // get each (validated) simple type from the DTO as a success or failure  
      let! checksum = data.Checksum |> Checksum.create "Checksum"  
      let! name = data.Name |> Name.create "Name"  
      let! path = data.Path |> Path.create "Path"  
      let! format = data.Format |> Format.create "Format"  
      // combine the components to create the domain object  
      return File {  
        Checksum = checksum  
        Name = name  
        Path = path  
        Format = format  
      }  
    }  
  | ConfigFileInputDto data ->  
    result {  
      let! configType = data.ConfigType |> FileType.create "ConfigType"  
      let! file = data.File |> InputFile.create "File"  
      return ConfigFileInput {  
        ConfigType = configType  
        File = file  
      }  
    }
```

Figure 4.7: The figure shows the "toDomain" function.

```
type InputFile = InputFile of string
module InputFile =
    string -> string -> Result<InputFile,string>
    let create fieldName str :Result<InputFile, string> =
        if (String.IsNullOrEmpty(str)) then
            Error (fieldName + " must be non-empty")
        else
            Ok (InputFile str)
    InputFile -> string
    let value d = match d with | InputFile d -> d
```

Figure 4.8: The figure shows an example of the validation when creating the Domain Object.

/5

Dependency Management in MdMP

5.1 Introduction

In this chapter, we discuss the dependency management implemented in MdMp. In an application, when a function calls another function, the former function then has a dependency on the latter one. It is very common to have dependencies inside an application. For instance, we need to connect to a database engine to store and manage our data, or to connect to some third party services such as payment services and authentication services. These dependencies can be "pure" or "impure". The meaning of "impure" here is defined under the context of referential transparency [15]. Thus, we consider every dependency that produces side effects as "impure". On the contrary, the dependency that always produces the same result with the same input is considered as "pure". Since we adopt the functional programming style throughout MdMp, we separate the impure and pure actions, and push the impure actions to the boundary of the program. The impure actions in MdMp include the communication between the database, the logging service and the file IO between the data storage.

When the application becomes larger and more complicated, the number of dependencies could grow rapidly. If the dependencies are not managed properly, we will face the following problems when the application starts to grow:

1. **Difficult to replace parts of the application:** During the development of a project, the business requirements could change and we may replace parts of the application with new services. Without proper management of the dependencies, it is hard to perform the replacement and we may need to rewrite the whole code base.
2. **Hard to write unit tests:** Unit testing is used to ensure the outcome of a function is expected and valid. If the dependencies are not handled properly, many unexpected outcomes could be generated from different dependencies and hence it would be difficult to do unit testing.

In Object-Oriented programming (OOP) [9], such dependency issues can be managed by dependency injection [4]. In functional programming, a similar approach is to pass the dependencies as parameters in a function, using partial application [7] instead of injecting the dependencies into the constructor or into a container. However, this approach is not totally functional because the function still has impure actions that would produce side effects.

In order to make it functional, we can use the dependency rejection [17] technique to first separate the pure and impure codes, and then combine these actions inside a top level function. The function that mixes pure and impure codes is called composition root. However, since we plan to make MdMp into an interactive program in the future, the dependency rejection approach is no longer applicable in an interactive program. Hence, we proceed to use dependency interpretation [17], known as Free Monad [6] in Functional Programming.

This chapter begins with the motivation to use Free Monad in MdMp (Section 5.2). We then discuss the design (Section 5.3) and the implementation of Free Monad in MdMp (Section 5.4).

5.2 Motivation

MdMp contains impure dependencies with different services, including the database service, the file I/O service and the logging service. By applying the Free Monad approach to manage these dependencies, we gain several benefits. The first one is that it allows late binding. We can delay the dependency using partial application, enabling us to switch out the interpreter with a different infrastructure. For example, we could replace our logging interpreter with another logger interpreter that shares the same interfaces without modifying the top-level composition root. The second benefit is that it is easier and faster for developers to write unit tests. By separating the impure and pure

dependencies, they can write unit tests for the pure parts and then write the integration tests in the composition root. Finally, it allows MdMp to implement different services in parallel because it enables loose coupling by creating separate model for each service.

However, the disadvantage of Free Monad approach is that it can be hard to understand for those who do not have knowledge about functional programming concepts like Monad. Other approaches like "dependency rejection" are more direct and do not require any special knowledge.

5.3 Design

To implement the Free Monad approach in MdMp, we first need to identify the impure dependencies and separate them from pure dependencies. There are three impure dependencies in MdMp:

1. **Database:** Our program needs to communicate with the database, such as querying and creating data in the database, which are impure actions. The database we use in our program is the graph database Neo4j.
2. **File IO:** The actions of File IO such as creating files and directories in the centralized data storage are impure.
3. **Logging:** There is logging in MdMp to keep track of the progress and errors throughout the program, and the logging actions are impure.

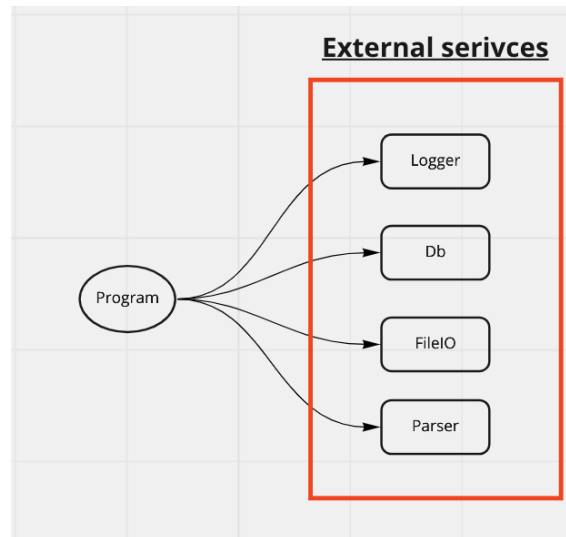


Figure 5.1: The figure shows the impure dependencies in our program.

After identifying the impure dependencies, we define a data structure which acts as instructions (Figure 5.2). In order to delay performing the impure actions, we change the type signature of the function from `'a -> 'b` to `'a * ('b -> 'c)`. That is, a tuple which contains the original input and a "next" function (Figure 5.3).


```
type DbInstruction<'a> =  
  | CreateNodes of [(string -> unit)]  
  | GetAllNodes of (unit -> string)
```

Figure 5.2: The figure shows an example of normal functions without creating instructions to delay the impure actions.

```
type DbInstruction<'a> =  
  | CreateNodes of string list * next:(unit -> DbInstruction<'a>)  
  | GetAllNodes of unit * next:(string -> DbInstruction<'a>)
```

Figure 5.3: The figure shows an example of how we create the instructions with a "next" function to delay the impure actions.

With this approach, we can apply it in a recursive interactive program because the "next" function delays the return of results until another instruction is received. To end the recursion, we add a "stop" instruction to the instruction set (Figure 5.4). The "stop" instruction returns the value received from the

previous instruction after being executed by the interpreter.

```
type DbInstruction<'a> =  
  | CreateNodes of string list * next:(unit -> DbInstruction<'a>)  
  | GetAllNodes of unit * next:(string -> DbInstruction<'a>)  
  | Stop of 'a
```

Figure 5.4: The diagram shows an example of adding the "stop" instruction to the instruction set.

Next, an interpreter is created to execute the instructions. The "next" function in the instruction is called after the execution, and the result of the execution is used as input in the "next" function (Figure 5.5).

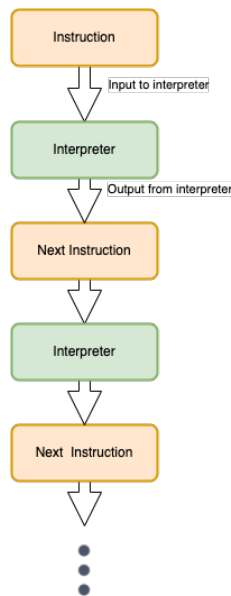


Figure 5.5: The diagram shows an example of the approach of using an interpreter to execute the instructions.

The interpreter also acts as the composition root for that service, which contains a mix of pure and impure actions. In the implementation, we first obtain the instructions that we need to execute. Note that these instructions are all pure, so we are able to reap the benefits of loose coupling and easier unit testing. After we receive the pure instructions, we use the interpreter to execute these instructions and then perform the impure actions (Figure 5.6). This is how we delay impure actions.

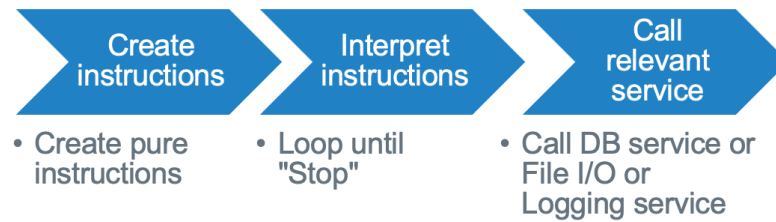


Figure 5.6: The diagram shows an example of the approach of using an interpreter to execute the instructions.

5.4 Implementation

The first step of implementing a Free Monad is to create a functor. After that, we built a monad from that functor. The monad is a new type containing the functor, and the monad has two types of value - the functor as the "Free" value and the normal value as the "Pure" value. To "Free" the monad, we turn the monad into recursive, whereas the "Free" values are the trees and the "Pure" values are the leafs.

5.4.1 Instruction as Functor

There are three impure dependencies in our program, we create an instruction type as functor for each of the impure dependencies to extract the pure part. We further wrap these three functors in a generic instruction type to allow interpreting different instructions in the program.

```
Node list -> Program<unit> 2 miclau13
let createNodes nodes =
  Instruction (CreateNodes(nodes, Stop))
```

Figure 5.7: The figure shows an example of the "createNodes" instruction to create nodes in the database.

We add an extra optional "Decision" layer to extract the corresponding instructions. The reason we add this layer is that we want to group the instructions into a decision that is defined by our domain logic. For example, when we copy directory to the data storage, we first have the instruction to create directory in the data storage, then we have the instruction to copy the directory to the destination. So we group these two instructions into a decision called "CopyDirectoryDecision" (Figure 5.8).

```
module Shell =
  Decision -> Program<unit> 2 miclau13
  let handleDecision (decision:Decision) :Program<unit> =
    match decision with
    | CopyDirectoryDecision copyDirectoryInput ->
      let _, destDirPath, _ = copyDirectoryInput
      program {
        do! createDirectory destDirPath
        do! copyDirectory copyDirectoryInput
      }
```

Figure 5.8: The figure shows an example of the "CopyDirectoryDecision" decision that contains two instructions to be execute.

5.4.2 Program as Free Monad

To create a Free Monad, we build a Monad from the functor, and the Monad is a union type named "Program" with two cases:

1. **Free:** The Free case contains the "Program" type itself wrapped in the instruction functor, which is a recursive type.
2. **Pure:** The Pure case contains the union's generic type. We use the type name "Stop" to represent the Pure case.

```
type Program<'a> =  
  | Instruction of IInstruction<Program<'a>>  
  | Stop of 'a
```

Figure 5.9: The diagram shows the "Program" with the "Free" part and the "Pure" part.

5.4.3 Interpreter

Once we have our "Program" as Free Monad, we create an interpreter function to recursively read the tree (Free case value) until it encounters a leaf (Pure case value) (Figure 5.10).

```
program<a>-> AsyncResult<a,b>> miclau13
let interpret program =
  let rec loop programAS =
    asyncResult {
      let! program = programAS
      return!
      match program with
      | Instruction inst ->
        match inst with
        | :? LoggerInstruction<Program<_>> as inst -> interpretLogger loop inst
        | :? DbInstruction<Program<_>> as inst -> interpretDbInstruction loop inst
        | :? FileIOInstruction<Program<_>> as inst -> interpretFileIOInstruction loop inst
        | _ -> failwithf $"unknown instruction type {inst.GetType()}"
      | NotYetDone p ->
        loop (p() |> asyncResult.Return)
      | Stop value ->
        value |> asyncResult.Return
    }
  // 3. start the loop
  let initialProgram = program |> asyncResult.Return
  loop initialProgram
```

Figure 5.10: The diagram shows the interpreter. It recursively execute the instructions until a "stop" instruction

There are three sets of instructions wrapped in the "Program" type corresponding to the three impure dependencies: the database service, the logging service and the file I/O service. We can easily add a new set of instructions to the interpreter if we have new impure dependencies in the future. Also, it is easy to switch out the current services, as long as the new one implements the same interface.

/6

Evaluation

6.1 Evaluation

Before MdMp was implemented, when users prepare input files for a new simulation, they may not know that other copies of the file are already in other simulation folders. Even if they do, they need to browse the other simulation folders and copy the files to the new simulation folder. MdMp helps to gather the files and metadata of the files in previous simulations such that users can have the knowledge of the files that were used. Then, users can search for particle files to use for their next simulation through MdMp.

There are around 20 simulation folders created on our HPC cluster. These simulations target different areas of the ocean. There are 9 input files inside each simulation folder on average, and 1 out of 9 of the files are the same on average. So, approximately 11% of the files are redundant. Therefore, the total number of files that need to be stored can be reduced by 9% by not having duplicated files. We believe the duplication rate will grow when we execute more simulations in future, especially when the new simulations target a part of the area that exists in the previous simulations, or an area that overlaps two areas, because they may share some of the same input files such as the wind file.

After adopting DDD in MdMp, we spent less development time on writing codes to check the validity of data coming from external services like the database because data was validated before we created the data into the domain objects

in MdMp, and we do not need to add extra codes to check the validate of the data in the functions that use the domain objects.

When using the impure dependencies in MdMp, including the database service, the file I/O service and the logging service, we found out that it was hard for us to debug the problems when the pure and impure dependencies are mixed together in the top-level composition root. After using the Free Monad approach to manage the dependencies, the debugging process becomes easier and less time consuming because the impure actions are separated according to the services, making it clearer to display where the errors occur.



Conclusion

7.1 Lesson Learned

We thought that it was easy to identify files with the same content by generating a SHA-1 checksum of the file content. The process was smooth for input files and the configuration file until it came to the output files. It took a long time to generate a SHA-1 checksum of the content of an output file. So we need to develop another approach to identify the output files with large sizes using a SHA-1 checksum but not generated by the file's content.

To ensure data validity in MdMp, we used the DDD approach when developing MdMP. It took time to create different domain object types and set up the validation rules when creating the domain objects. But with the help of the domain object types, we instantly know if we commit any mistakes or not when we pass the data around to different functions to perform different operations on the data. Also, we do not need to worry about the data received from other services being corrupted or invalid because of the validation rules.

To manage the dependencies in MdMp, we used the Free Monad approach. At the beginning, it was complex and it took time to build up different interpreters and the instructs inside the interpreters. However, we believe it is worth our effort to implement it because the impure actions of each service are isolated from each other. So, it was easy to add a new interpreter for logging and replace our logging interpreter with another logger interpreter without affecting the other services.

7.2 Limitation

MdMp only supports parsing the content of the configuration file in FVCOM now. However, the parser was built as a standalone library. So, to support another simulation model with different formats of content in the configuration file, the parsing logic can be added to the parser library without affecting MdMp.

7.3 Future Work

Currently, MdMp only provides commands on the command line tools for users to query for the files that are stored in the knowledge graph and the centralized data storage. We plan to provide a GUI (Graphical User Interface) on a web server that allows users to query data and provide more searching criteria. Also, we plan to provide a retention policy to filter the old data in the centralized data storage that is not being used for a long period of time.

Bibliography

- [1] Jesús Barrasa, Amy E. Hodler, and Jim Webber. *Knowledge Graphs - Data in Context for Responsive Businesses*. 1st ed. O'Reilly Media, 2021.
- [2] Scott Chacon and Ben Straub. *Pro Git*. 2nd ed. Apress, 2014.
- [3] Nils Anders Danielsson. "Total Parser Combinators." In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. Association for Computing Machinery, 2010, pp. 285–296. ISBN: 9781605587943. DOI: 10.1145/1863543.1863585. URL: <https://doi-org.mime.uit.no/10.1145/1863543.1863585>.
- [4] STEVEN VAN DEURSEN and MARK SEEMANN. *Dependency Injection - Principles, Practices, and Patterns*. Manning Publications Co., 2019.
- [5] "Discriminated Unions." In: (2021). URL: <https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/discriminated-unions>.
- [6] "F# free monad recipe." In: (2017). URL: <https://blog.ploeh.dk/2017/08/07/f-free-monad-recipe/>.
- [7] Dave Fancher. *The Book of F#: Breaking Free with Managed Functional Programming*. 1st ed. Manning Publications Co., 2014.
- [8] "FVCOM." In: (2001-2022). URL: <http://fvcom.smast.umassd.edu/>.
- [9] Cay Horstmann. *Object-Oriented Design and Patterns*. 2nd ed. Wiley, 2005.
- [10] "Introducing the Knowledge Graph: things, not strings." In: (2012). URL: <https://blog.google/products/search/introducing-knowledge-graph-things-not/>.
- [11] "Introduction to OpenDrift." In: (2020). URL: <https://opendrift.github.io/>.
- [12] Einar Landre, Harald Wesenberg, and Jorn Olmheim. "Agile Enterprise Software Development Using Domain-Driven Design and Test First." In: *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*. OOPSLA '07. Association for Computing Machinery, 2007, pp. 983–993. ISBN: 9781595938657. DOI: 10.1145/1297846.1297967. URL: <https://doi-org.mime.uit.no/10.1145/1297846.1297967>.
- [13] "Ln Command in Linux (Create Symbolic Links)." In: (2019). URL: <https://linuxize.com/post/how-to-create-symbolic-links-in-linux-using-the-ln-command/>.

- [14] “Neo4jClient.” In: (2016). URL: <https://github.com/DotNet4Neo4j/Neo4jClient/wiki>.
- [15] “Referential transparency fits in your head.” In: (2021). URL: <https://blog.ploeh.dk/2021/07/28/referential-transparency-fits-in-your-head/>.
- [16] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. 7th ed. McGraw-Hill, 2019.
- [17] “Six approaches to dependency injection.” In: (2020). URL: <https://fsharpforfunandprofit.com/posts/dependencies/>.
- [18] Dr. Jim Webber and Rik Van Bruggen. *Graph Databases For Dummies, Neo4j Special Edition*. John Wiley & Sons, Inc., 2021.
- [19] Scott Wlaschin. *Domain Modeling Made Functional*. Pragmatic Bookshelf, 2018.

