Dataflow Analysis of the Earth System Model MITgcm

Bachelor's Thesis

Simon Ohlsen

September 28, 2022

Software Engineering Group
Department of Computer Science
Kiel University

Advised by
Prof. Dr. Wilhelm Hasselbring
Dr.-Ing. Reiner Jung

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Many of the earth system models (ESMs) used today, e.g., for climate forecasting and ocean modeling, are originally from the late 1990s. The long lifespan and still ongoing development makes them potentially vulnerable to architectural degradation and thus less maintainable and flexible. Therefore, there are efforts to better understand, modernize, and future-proof their software architecture. To achieve this, the software architecture of the ESM must be reconstructed.

With this thesis, we contribute to the OceanDSL project by reconstructing the architecture of the ESM *Massachusetts Institute of Technology General Circulation Model* (MITgcm). Since ESMs are data-centric applications, we develop a static dataflow analysis procedure based on an abstract syntax tree of the ESM's Fortran source code to extract dataflow information and reconstruct the dataflow-based architecture. This requires the identification of dataflow relevant structures in Fortran, enhancing the given architectural meta-model and the tools provided by OceanDSL, as well as the evaluation of the tooling based on different variants of the MITgcm. We assess the reconstructed architectures by comparing them to the architecture description provided by the MITgcm documentation and using various metrics. As a result, we identified `COMMON` blocks as key components in the interprocedural dataflow of the MITgcm and were able to recover selected design decisions of the descriptive architecture.

Contents

Chapter 1

Introduction

Accurate climate forecasts have become increasingly important over the last decades. As the human impact on climate change got widely recognized, extreme weather events became more common, and the threat of rising sea levels to cities and coastlines became clearer, the need for valid climate predictions increased drastically. Many of today's used Earth System Models (ESMs) are originally from the last century. Therefore, there are efforts to modernize and future-proof their software architecture, but that requires recovering the architecture beforehand.

A software system's architecture is a high-level representation of the structure, behavior, and nonfunctional properties of the software's components and their interactions [Zhang and Goddard 2005]. Recovering a software architecture consists of the identification and extraction of higher-level representations of an existing software system and the design decisions made during its development [Rasool and Asif 2007]. There is a huge variety of different approaches to architecture recovery, with many of them focussing on control flow structures, calling hierarchies, and dataflow, but also on dynamically obtained data like metrics based on monitoring. However, none of the mentioned approaches manages to capture the entirety of design decisions and structural details of a software, at least not for larger software systems. Therefore, a combination of different approaches is necessary, depending on the use case [Erdos and Sneed 1998]. A comprehensive software architecture enables further analysis and evaluation, for example of modularization. Modularization is a key metric in identifying architectural degradation and comparing architectures [Sant'Anna et al. 2007], as well as an indicator for quality attributes like maintainability, reusability, or flexibility [Ghasemi et al. 2015].

The OceanDSL[1] project has the aim to develop a domain-specific language (DSL) for ocean modeling and simulation. Therefore, it is essential to understand the design and composition of ESMs and ocean models. Furthermore, the design of existing models can be evaluated and potentially improved, for example, in regard to modularization, to improve their maintainability, usability, and relevance for the future. In this context, we developed a tool to analyze the dataflow in the *Massachusetts Institute of Technology General Circulation Model* (MITgcm) and *University of Victoria Earth System Climate Model* (UVic ESCM) with the aim to reconstruct and evaluate their dataflow-based architecture by adapting the provided tools from OceanDSL. The results can be combined with, for example, call-graph-based

[1] `https://oceandsl.uni-kiel.de/`

approaches to architecture recovery, among other things, to get a more complete picture of the actual architecture of the ESMs.

This work is a joint project with Yannick Illmann, who focuses in his thesis *Data Flow Analysis of the University of Victoria Earth System Climate Model* on the architecture reconstruction process and analysis of the UVic ESCM, while this thesis' emphasis is primarily on the data flow analysis and application to the MITgcm. Our project has the following overall goals:

G1: Implementation of Dataflow Analysis on an Abstract Syntax Tree

The first goal is to extract the dataflow from the Fortran code of the ESMs via static source code analysis. Therefore, we develop a new Python tool *ESM Dataflow Analysis*, which is partially based on the existing tool *ESM Coupling Analysis* and utilizes *fparser* to obtain an abstract syntax tree (AST) of the source code. Additionally, we define a way to store the extracted information in a way that allows other tools to reconstruct an architecture from it.

G2: Enhance Architecture Meta-Model and Visualize Architecture

The next step is to extend the given architecture recovery tools from OceanDSL to handle the extracted dataflow. Therefore, we need to enhance the given architecture meta-model provided by *Kieker*[2]. We also utilize the *Kieker Development Tools*[3] to visualize the extracted architecture, which then allows further analysis. The majority of this work is done in Java.

G3: Application to MITgcm and Evaluation

After developing and extending the tools for dataflow analysis and architecture recovery, the aim is to evaluate our implementations by applying them to the ESM MITgcm and analyze its architecture. Application and analysis also cover different variants of the model to point out changes in dataflow depending on the model's configuration.

We begin by describing the necessary foundations and tools or frameworks the work is based on in Chapter 2. In Chapter 3, we introduce the analysis concept and how we applied it to Fortran source code. Additionally, we provide insides on the implementation and data structures used to store the extracted dataflow information. Chapter 4 outlines the changes made to the architecture meta-model and the reconstruction of the architecture. In Chapter 5, we apply the tooling to the tutorial experiments of the MITgcm to evaluate the reconstructed dataflow-based architectures. In Chapter 6, we discuss the analysis results and limits of the implementations along with suggestions for improvement. Chapter 7 is about related work in the fields of dataflow analysis and architecture recovery. Finally, Chapter 8 summarizes our findings and provides an outlook on future work.

[2] `https://kieker-monitoring.net/`

[3] `https://github.com/kieker-monitoring/instrumentation-languages`

Chapter 2

Foundations and Technologies

In the following, we provide an overview of the tools, frameworks, and concepts that lay the foundation for our implementations. First, in Section 2.1, we introduce the MIT General Circulation Model and it's general software architecture. Second, in Section 2.2, outline the basics of Fortran. In Section 2.3, we explain the theoretical foundation for the analysis and modeling of dataflow. Finally, in Section 2.4, we introduce the software tools and frameworks that enabled the implementation of the analysis and visualization of the results.

2.1 MIT General Circulation Model

The *Massachusetts Institute of Technology General Circulation Model* (MITgcm) is a numerical earth system model developed and maintained by the *Massachusetts Institute of Technology* (MIT) since the late 1990s. It is designed around a non-hydrostatic dynamical kernel that drives an atmospheric and oceanic model each with its associated physics (cf. Figure 2.1). These base models can either run separately or combined enabling study from, e.g. small-scale local convection in the atmosphere or ocean, up to global circulation patterns and climate [MIT 2022]. The MITgcm is hosted on GitHub[1] and publicly available. Most of the models source code is written in Fortran 77 and Fortran 90.

Software Architecture

The MITgcm is designed highly modular and flexible to be able to "study a broad range of problems [...] [on] a wide range of platforms" [MIT 2022, p. 341], i.e., from running simulations on a single computer to large-scale multiprocessing clusters and from small process studies to huge coupled climate experiments. It consists of "a core set of numerical and support code" [MIT 2022, p. 341] as well as "a scheme for supporting optional 'pluggable' packages" [MIT 2022, p. 341] extending the core numerical code with alternate dynamics and more specialized physics. Together core code and packages form the *numerical model*. The code of the numerical model is written to fit inside a special software support infrastructure called *Wrapper* (Wrappable Application Parallel Programming Environment Resource). The *Wrapper* is an additional abstraction layer between the numerical model and

[1] `https://github.com/MITgcm/MITgcm`

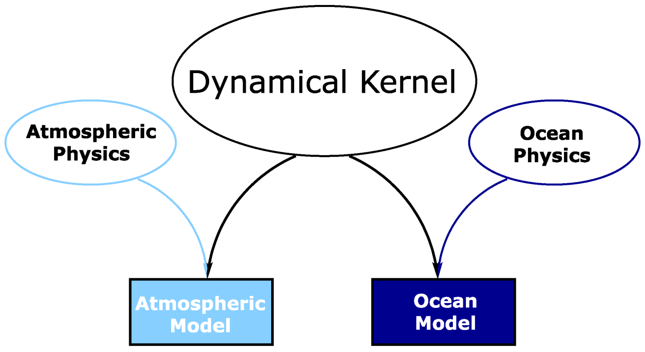

Figure 2.1. The MITgcm has a dynamical kernel that can drive either oceanic or atmospheric simulations with associated physics. From [MIT 2022, p. 1].

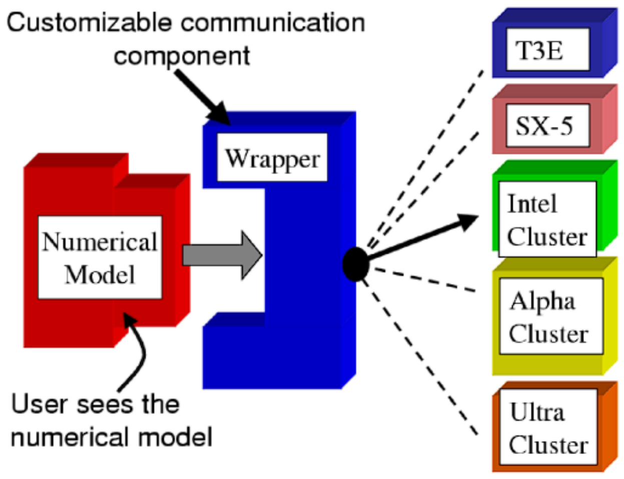

Figure 2.2. The *Wrapper* functions as an insulation and adaptation layer between the numerical model and the target hardware and operating systems to make the MITgcm easily runnable for a variety of target platforms. Adapted from [MIT 2022, p. 343].

target operating system and hardware. Its purpose is to isolate the model from architectural differences of hardware platforms and runtime environments to make it easily deployable on a wide range of platforms [MIT 2022]. In Figure 2.2 we can see how the *Wrapper* can be configured to run the numerical model on a specific target platform without changes to the model itself.

For the highest compatibility, the *Wrapper* is written in Fortran 77. In the architectural models, packages that are from the core model are denoted with just the package name, while packages that extend the core models functionality have the prefix `pkg/`. This naming scheme is derived from the MITgcm directory structure. The *Wrapper* environment is represented by the package named `eesupp`. Within `eesupp` `main.f` is the starting point for the execution of the whole model including the *Wrapper*. `main.f` starts the initialization of the *Wrapper* by calling `eeboot`, which then calls additional components to setup the execution environment. Additionally, `main.f` starts the execution of the numerical model by calling `the_model_main.f`, after the initialization is complete. Listing 2.1 shows an excerpt of the startup calling sequence.

Listing 2.1. Shortened version of the MITgcm startup sequence. Adapted from [MIT 2022, pp. 355-56].

```
MAIN
|
|--EEBOOT               :: WRAPPER initialization
|  |
|  |-- EEBOOT_MINMAL    :: Minimal startup. Just enough to
|  |                       allow basic I/O.
|  |-- EEINTRO_MSG      :: Write startup greeting.
|  |
|  |-- EESET_PARMS      :: Set WRAPPER parameters
|
|
|--CHECK_THREADS    :: Validate multiple thread start up.
|
|--THE_MODEL_MAIN   :: Numerical code top-level driver routine
```

2.2 Fortran

Fortran is a widely used scientific programming language. Its imperative, procedural, and high-performance design makes it ideally suited for numeric computation and simulations. Since its first version was released in 1957, several new releases added increasing functionality up to object-oriented and concurrent programming capabilities [Backus and Heising 1964]. The latest version available is Fortran 2018. In the following, we focus on the versions Fortran 77 and Fortran 90 because these are the ones used within MITgcm.

A Fortran 77/90 program consists of a main executive program in a `PROGRAM` block that can be possibly followed by multiple subprograms, which are either a `SUBROUTINE`, `FUNCTION`, or `BLOCK DATA`. A subprogram can be referenced by other subprograms or the executive program [Janni et al. 1986]. Another important structure is the `COMMON` block, which is a part of memory shared between different program parts to exchange data without using arguments [FORTRAN 2010]. The use of modules allows organizing and structuring large programs, although MITgcm uses the C preprocessor to include different packages and functionality during compilation of the model.

2.3 Dataflow Analysis

Execution of a computer program can be generally broken down into three steps: input of data, operations on the data, and output of the results of the performed operations. The program determines the sequence of operations the data is subject to and thus the flow of data through the program [Fosdick and Osterweil 1976].

A basic technique to analyze and model dataflow is the construction of flow graphs. First, we need a formal representation of a program given by a *control flow graph* (CFG). Formally, a CFG is a graph $G_F(N, E, n_0)$, with a set of nodes N and a set of directed edges E as well as a unique entry node $n_0 \in N$ [Fosdick and Osterweil 1976]. Each node represents an executable statement of a program. Multiple statements that form a *basic block* can be combined into a single node to reduce the complexity of the graph. A basic block is a "linear sequence of [...] [statements] with [exactly] one entry and one exit point" [Söderberg et al. 2013, p. 1810], i.e., parts of a program that are executed sequentially together in every possible case without branches or cycles. The edges link nodes in the order of execution, starting from the unique entry node n_0. Every node in the CFG has to be reachable from the entry node, so there has to be at least one path from the entry node to every other node. While there is only one entry node, the graph may have more than one exit node [Fosdick and Osterweil 1976]. The CFG is used to guide the analysis of dataflow through the program at statement level. One simple way to model dataflow with a CFG is to annotate nodes with in- and out-sets, with an in-set containing all variables that are not defined but referenced inside the basic block corresponding to the node. Vice versa the out-set contains all variables from a basic block that are referenced in a following basic block. For example, in the Fortran statement `A = B + C` the variable `A` is defined, while `B` and `C` are referenced.

Figure 2.3 shows the result of an exemplary dataflow analysis using the *worklist algorithm* described by Cooper et al. [2004]. In short, the analysis starts at one exit point and adds all variables that are not defined within this block to the in-set of this block. The out-set of an exit node is empty because all local variables become undefined when a `return` is executed [Fosdick and Osterweil 1976]. All predecessor nodes are added to the worklist. For every node in that list, the in-set is computed such as one of the exit nodes. The out-set of a node is the union of the successor's in-sets. The node is then removed from the worklist. If any of the sets changed, all successors are added to the worklist and these steps are repeated until the worklist is empty.

We do not use this specific approach in our implemented analysis because it is CFG-based, but it resembles our understanding of dataflow and lays the foundation for our own approach described in Chapter 3.

```
# B1:
a = 42
b = 2
c = 0
x = 1
if a = b then

# B2:
    c = a + b

# B3:
else:
    x = a - b

# B4:
return c * x
```

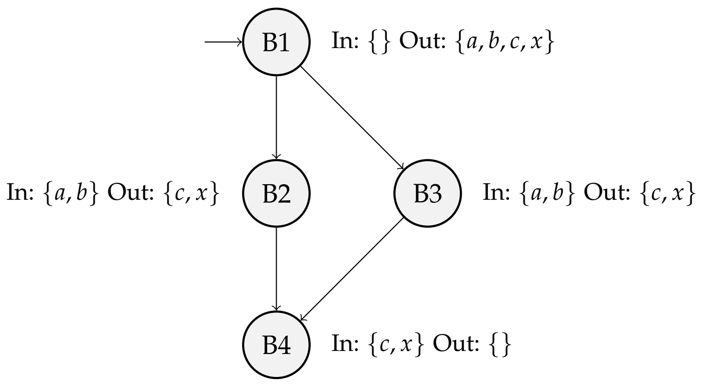

Figure 2.3. A simple algorithm on the left with its corresponding CFG on the right hand side. Basic blocks are marked by the comments in the code. Annotations of nodes represent the respective in- and out-sets.

2.4 Analysis Tools

This section contains all software tools and frameworks we used to implement the dataflow analysis, reconstruct the architecture and visualize the results. We give a brief introduction to each and explain in which way we use them.

fparser

fparser is a Python package implementing a parser for the programming language Fortran from Fortran 66 up to Fortran 2008. It includes *fparser1* and *fparser2*. While *fparser1* is deprecated and only supports parsing down to line level of Fortran 66/70/90, *fparser2* enables fully parsing Fortran code with additional support for Fortran 2003 and some Fortran 2008. The tool originated from the F2PY Project [Peterson 2009], which had the aim to connect Fortran and Python in an easy way by automatically generating Python wrappers to Fortran libraries. Initial development was started by Pearu Peterson. Since 2017 the package is publicly available on GitHub[2] or the Python Package Index. We use *fparser2* to obtain an abstract syntax tree (AST) from the Fortran code of the MITgcm and the build-in functions to navigate the AST.

[2]`https://github.com/stfc/fparser`

OceanDSL Tools

We use a suite of software tools[3] developed in the context of *OceanDSL*[4] to generate an architectural model from the analyzed dataflow and to calculated metrics from the generated model. In more detail, we use the tool *static architecture reconstruction* (SAR) to reconstruct the architecture of the MITgcm with the dataflow provided by the dataflow analysis. Additionally, we use *model visualization* (MVIS) to calculate metrics on the architectural model such as model complexity and coupling.

Eclipse Modeling Framework

The *Eclipse Modeling Framework* (EMF) "is a modeling framework and code generation facility for building tools and other applications based on a structured data model" [EMF Website 2022]. The framework is build around the *Essential Meta Object Facility* (EMOF) specification of the *Object Management Group* (OMG). It is able to generate classes and adapter classes from a model described in XMI to enable viewing and editing of the model. The EMF is the foundation for the dataflow model generation.

Kieker Monitoring Framework

Kieker is a framework for dynamic software analysis and monitoring. In contrast to static analysis, it collects and analyzes monitoring data at runtime, allowing performance monitoring as well as architecture discovery [Hasselbring and van Hoorn 2020; van Hoorn et al. 2012]. Kieker includes an architecture meta-model for its dynamic architecture recovery. We use this meta-model and extend it so support dataflow as well. Furthermore, we use the *Kieker Development Tools* (KDT) to visualize the architecture models.

The architecture meta-model is based on the EMOF specification. Altogether, it describes seven sub-models that imply different visualization possibilities. Four of the models are directly related by cross-referencing classes (cf. Figure 2.4). As a result, a top to bottom design approach is necessary for implementation. This means that visualizing an execution model requires referenced *DeploymentComponents* defined in the *Deployment Model*. However, the *DeploymentComponents* store *AssemblyComponents* defined in the *Assembly Model*. Further, the *ComponentType* is defined in the *Type Model* and stored in an *AssemblyComponent*.

The architecture visualization takes place in *Eclipse* using the Kieker architecture visualization plugin. It can create different diagrams - depending on the model type - from imported meta-models given in XMI-format (XML Metadata Interchange). Kieker and all of its components are available on GitHub[5].

[3] `https://github.com/cau-se/oceandsl-tools`

[4] `https://oceandsl.uni-kiel.de/`

[5] `https://github.com/kieker-monitoring`

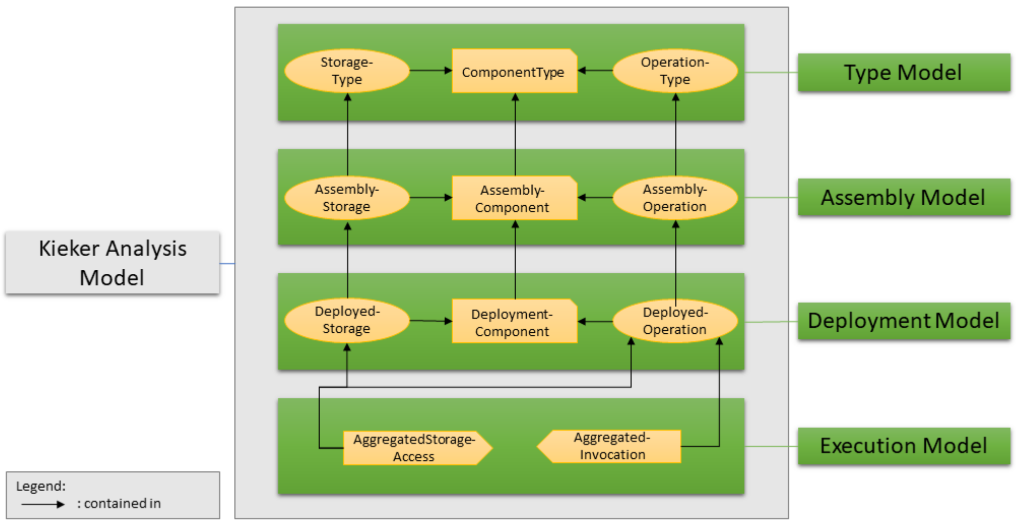

Figure 2.4. Kieker Architecture Meta-Model concept diagram.

TeeTime Framework

The pipe-and-filter framework *TeeTime* enables the modeling and execution of arbitrary pipe-and-filter architectures sequentially or in parallel. For this work, we use TeeTime to implement several sequential stages connected by pipes that convert from the ESMs extracted dataflow into an architectural model. A description of other features and application examples can be found in [Wulf et al. 2017].

Chapter 3

Dataflow Analysis on an Abstract Syntax Tree

For our goal of reconstructing the architecture of an ESM like MITgcm, the level of detail an intraprocedural dataflow analysis offers, such as the one described in Section 2.3, is not necessarily required. Instead, we can permit a loss of information on dataflow inside operations to focus on dataflow between operations and components. In the context of Fortran, operations correspond to a `SUBROUTINE`, `FUNCTION`, or `PROGRAM`. `COMMON` blocks are modeled as storages and files or directories are represented by components. The higher abstraction level of interprocedural dataflow allows us to leave out the construction of a control flow graph (CFG) before the analysis and directly extract dataflow from the abstract syntax tree (AST). In the following section we describe the general concept behind this approach to dataflow analysis. In Section 3.2, we explain the details of the implemented analysis.

3.1 Concept

In contrast to many approaches to dataflow analysis, this approach to dataflow is not focused on the dataflow of individual variables inside operations, such as functions and subroutines, but rather on a more general level between operations and components. This means, we are not interested in precisely tracking which variables and data are computed and transferred. Instead, we look at whether data is transferred between operations and in which direction. Therefore, we base the analysis of dataflow on a call graph instead of a CFG, because we do not need the level of detail on statement level that a CFG provides, but rather the description of invocations of program components on a operation level that is modeled by a call graph.

We distinguish between two types or directions of dataflow that we denote with *read* and *write*, with *read* being the retrieval of data from another operation, e.g., assigning the return value of an operation to a variable and *write* being the transfer of data to a different operation, e.g. via a function call.

The structure of a call graph $G_C(N, E, n_0)$ is formally identical to a CFG, but with different meanings for nodes and edges. A node in a call graph corresponds to an operation with edges (n_i, n_j) between nodes, indicating that execution of operation n_i invokes execution of operation n_j [Fosdick and Osterweil 1976]. To model dataflow, we take the call graph as a basis and construct a new graph $G_D(N_D, E_D)$ with $N_D = N_C$ in the following way (cf. Listing 3.1): for every node $n_i \in N_C$ of the call graph $G_C(N_C, E_C, n_0^C)$, we take the set of all outgoing edges of a node $out(n_i) = \{(n_i, n_j) \in E_C \mid n_j \in N_C\}$. Then, for each of the outgoing edges, we analyze whether the invocation of another operation involves handing over data, retrieving data, or both. If in operation n_i data is transferred to another operation n_j, we add the edge (n_i, n_j) to the set of edges E_D of the dataflow graph G_D. In case data from n_j is retrieved, we add the edge (n_j, n_i) to G_D. If data is both read and written, we add both edges to the dataflow graph.

Listing 3.1. Algorithm to construct a dataflow graph based on a call graph.

```
Input: $G_C(N_C, E_C, n_0)$
Output: $G_D(N_D, E_D)$

$N_D = N_C$
for each $n_i \in N_C$ do:
    for each $(n_i, n_j) \in out(n_i)$ do:
        if data is sent from $n_i$ to $n_j$:
            $E_D = E_D \cup \{(n_i, n_j)\}$
        fi
        if data is retrieved from $n_j$ by $n_i$:
            $E_D = E_D \cup \{(n_j, n_i)\}$
        fi
    done
done
```

However, this approach requires the construction of a call graph beforehand, which means we have to walk the AST multiple times. First, completely to construct the call graph and later partly for each check whether data is read or written to or from an operation. We can reduce this effort drastically, if we combine finding invocations of other operations with checks for read and write relations along with the construction of a dataflow graph. Therefore, we adapted the call-graph-based algorithm in Listing 3.1 as depicted in Listing 3.2: Assuming an AST given as an acyclic directed graph $G_T(N_T, E_T, r)$ with a set of nodes N_T, a set of edges E_T, and a unique node r with no incoming edges, we are interested in all nodes $n_i \in N_T$ that define an operation. For each node n_i, we add it to the dataflow graph, then look at all its successor, i.e. all statements inside that operation, and check whether it involves the execution of a different operation. If so, we add that operation to the dataflow graph as well and check whether it stands in a *read* or *write* relation or both of them. If one is the case, we add the respective edge to the dataflow graph analog to the call-graph-based algorithm.

The key points of this algorithm are the two innermost if-conditions that check whether a *read* or *write* relation exists. Concrete procedures for this problem are highly dependent on the programming language involved. In the following section we describe how we adapt the described concepts to Fortran.

Listing 3.2. Algorithm to construct a dataflow graph based on an AST.

```
Input: $G_T(N_T, E_T, r)$
Output: $G_D(N_D, E_D)$

for each $n_i \in N_T$ defining an operation do:
    $N_D = N_D \cup \{n_i\}$
    for each successor $n_j$ of $n_i$ do:
        if $n_j$ involves execution of another operation $n_x$:
            $N_D = N_D \cup \{n_x\}$
            if data is sent to $n_x$:
                $E_D = E_D \cup \{(n_i, n_x)\}$
            fi
            if data is retrieved from $n_x$:
                $E_D = E_D \cup \{(n_x, n_i)\}$
            fi
        fi
    done
done
```

3.2 Application to Fortran

Before we can begin the dataflow analysis itself, we need to obtain the AST of the Fortran code. Because the ESM's code is divided into several files, we construct an AST of each file and analyze the dataflow on a per-file basis. Later, we combine the results of individual files into a global result representing the dataflow of the whole model. In the following, we assume that the names of operations are unique. That is indeed the case for MITgcm, but may not apply to every Fortran program.

In Section 3.2.1 we explain how we model the information extracted by the analysis. In Section 3.2.2, we discuss our specific dataflow analysis procedure. Finally, in Section 3.2.3, we outline our Python implementation of the analysis.

3.2.1 Data Model

Since the Fortran code is during compilation first processed by a C or Fortran preprocessor that configures dependencies using `#include`, we do not know from which file operations, that are not defined in the component we are currently analyzing, come. To circumvent this problem, we create two tables (cf. Table 3.1 and Table 3.2) during the dataflow analysis,

Table 3.1. Exemplary table *file_contents* which maps operations to the file they are defined in.

File	Identifier	Type
c_one.f	INC	FUNCTION
c_two.f	SUB	SUBROUTINE
c_two.f	RUN	PROGRAM

Table 3.2. Exemplary table *dataflow* which saves all read and write relations (or both) between two operations or an operation and a storage.

File caller	Identifier caller	Flow type	Identifier callee
c_two.f	RUN	WRITE	SUB
c_two.f	RUN	READ	/vars/
c_two.f	SUB	BOTH	INC

which will be the basis for the architecture reconstruction. In the first table called *file_-contents* (cf. Table 3.1), we map all operations to the file (component) they are defined in, except `COMMON` blocks because these can be defined and referenced in multiple operations, so we cannot make a clear assignment. We deal with `COMMON` blocks during architecture reconstruction and group them as storages in a separate component. The dataflow information itself is stored in a separate table named *dataflow* (cf. Table 3.2). It includes the file name and identifier of the operation the other operation is referenced in, as well as the name of the referenced operation and whether data flows from caller to callee (WRITE), from callee to caller (READ), or in both directions (BOTH). A special case is `COMMON` blocks. Because we cannot map them in the *file_contents*, we need a way to distinguish them from operation calls. As a result, we surround their name with two slashes, similar to how they are declared in Fortran: `COMMON /example_block/`. Table 3.2 shows an example for the *dataflow* table, while Table 3.1 represents the table *file_contents*.

3.2.2 Analysis Procedure

The key structures in the ESMs Fortran 77/90 parts regarding interprocedural dataflow are `PROGRAM`, `SUBROUTINE`, `FUNCTION`, and `COMMON` block. These are the ones that define how data flows through the program and its operations. `COMMON` blocks are referenced within one of the three other operations, so we analyze `COMMON` blocks by analyzing the operation it is referenced within. Figure 3.1 outlines the general course of the analysis. Note that the order in which operations are analyzed is arbitrary and without any loss of generality. We explain the analysis of operations using Listing 3.3 and Listing 3.4 as an example. The steps performed are the same for all operations within the AST.

First, we construct a *blacklist* of all variables and arrays assigned inside the operation we want to analyze. We have to do this because fparser - the tool we utilize to construct the AST - is in some circumstances not able to distinguish between arrays and function calls.

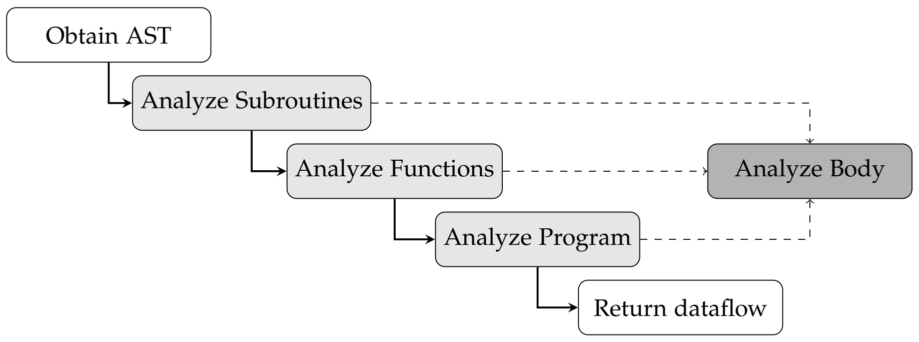

Figure 3.1. Course of the dataflow analysis of a single file. For each `SUBROUTINE`, `FUNCTION`, or `PROGRAM` defined within, the file the dataflow of all statements, e.g. assign statements, in scope of the operation are analyzed.

As a result, we work with the blacklist to verify that a function call is indeed a function call and not an array that is wrongly labeled. Each function call that matches an entry in the blacklist is discarded. After this step, we create a list that stores all the `COMMON` blocks used inside the current operation along with a list of their contained variables or arrays. We use this list later to check whether we read from or write to a `COMMON` block when, for instance, a value gets assigned to a variable. With these preliminary step complete, we begin with the analysis of individual statements. We differentiate between five types of statements that we will cover in this order: call statements, `IF` statements, loop control statements, select case statements, and assign statements.

Call statements are used to branch to a subroutine and return to the calling program after finishing [FORTRAN 2010]. The call of a subroutine can include arguments but does not have to. If at least one argument is included, we save the caller and callee together with the flow type *WRITE* in the table *dataflow* as we describe in Section 3.2.1. For example, the subroutine call `CALL SUB(X, Y)` in `PROGRAM RUN` of Listing 3.3 includes two local variables and thus is a *write* from `RUN` to `SUB`.

Next, we look at `IF` (or `ELSE IF`) statements and select case statements. Note that we only analyze the statements themselves, not the whole block, e.g., `SELECT CASE (x) ... CASE(Y)` and `IF (X .LT. Y) THEN`. These are read-only statements because each only checks a condition and is not assigning any value to the referenced variables. We check whether the referenced variables are defined in a `COMMON` block. Otherwise, they are defined locally inside the operation and covered by the assign statement analysis. Loop control statements are generally read-only too and handled like `IF` and select case statements, except in a special case where the iterator variable is declared in a `COMMON` block. In this case, we have a potential *write* relation to that block that has to be included in the dataflow table.

Finally, we analyze all assign statements inside the operation. First, we split the statement into two parts: the value or data that is assigned to a variable, called assigned part, and the part that the assignment is made to, called defined part. For instance, the assign statement `Y = INC(X)` consists of the defined part `Y` and the assigned part `INC(X)`. In the next step, we differentiate on the assigned part side between the call of a function with arguments or an array (because fparser cannot distinguish them in every case), a function call without arguments, and a variable or value. In the first case, we check whether it is a function or an array using the blacklist. Then, if it is an array, we check whether it is from a `COMMON` block. In case it is, we add a *read* relation between the block and the current operation. If we look at a function and not an array, we check whether or not one of the arguments is referenced from a `COMMON` block. If the former is the case, we add a *read* relation to the block and a *write* relation to the called function. Otherwise, we only add a *write* relation to the called function or a both relation if the assigned part is not from a `COMMON` block. For example, the assign statement `Y = INC(X)` from Listing 3.3 consists of a defined part with a local variable `Y` and an assigned part `INC(X)` with a function call including an argument, which is also a local variable and not from a `COMMON` block. Thus, we add the flow type *BOTH* to the table *dataflow*, as data is sent to `INC` via the argument `X` and retrieved by saving the return value of `INC` in `Y`. The cases for functions without arguments and variables or values can be handled more easily. For functions without arguments, we have a *read* relation to the called function and additionally the possibility for a *write* relation to a `COMMON` block, if the assigned part is declared in a `COMMON` block.

If the assigned part consists of a single value, we only have to check whether the assigned part is defined in a `COMMON` block, because in this case we write to it. Otherwise, we assign a value to a local variable which is intraprocedural dataflow and thus not relevant. In case another variable is assigned, we check whether this variable is from a `COMMON` block. If so, we add a *read* relation to that block. For instance, the statement `Y = A` in `PROGRAM RUN` from Listing 3.3 is an assignment of a variable (`A`) from the `COMMON` block `/vars/` to a local variable (`Y`) of the operation `RUN`, so `RUN` reads data from `/vars/`.

Additionally, we add each operation we analyze to the table *file_contents* (cf. Section 3.2.1). Table 3.1 and Table 3.2 show the resulting tables for the dataflow analysis of both files `c_one.f` and `c_two.f` given by Listing 3.3 and Listing 3.4 respectively.

Listing 3.3. Function INC in c_one.f

```
      FUNCTION INC(A)
      IMPLICIT NONE
      INTEGER :: A

      INC = A + 1
      RETURN
      END FUNCTION
```

Listing 3.4. Subroutine SUB and program RUN in c_two.f

```fortran
      SUBROUTINE SUB(X, Y)
      IMPLICIT NONE
      INTEGER :: X
      INTEGER :: Y

      Y = INC(X)
      END SUBROUTINE

      PROGRAM RUN
      IMPLICIT NONE
      INTEGER :: A, X, Y
      COMMON /vars/ A, B, C

      Y = A
      X = 1
      CALL SUB(X, Y)
      END PROGRAM
```

3.2.3 Implementation

For the implementation of the dataflow analysis with the goal to analyze the dataflow of ESMs, such as MITgcm, we want to exploit the inherent concurrency of analyzing several hundred Fortran files individually. We opted for a master-worker model, similar to Figure 3.2. The dataflow analysis of a single file of the model is one task with the result consisting of the two tables *file_contents* and *dataflow*. The master assigns the first *n* files to *n* threads. For each finished file a new worker with a new file is initialized. This allows for easy adaptation to the target execution environment in regard to thread count and thus more efficiently utilizing available processors while reducing execution time. After every file got processed, the individual results are merged into a single global result by the master and saved into a CSV file, which we then hand over to the *SAR* tool.

We implement the dataflow analysis itself completely in Python 3.10 because fparser is also a Python implementation, which provides the Fortran AST and methods to navigate it that are essential for the analysis on the AST. The module `main.py` implements the master thread using the multiprocessing library and additionally includes methods to construct the list of tasks and to combine along with writing the results into a file. All functions related to the AST, dataflow analysis, and the implementation of the worker itself are located in the worker package. In the module `ast_functions.py`, we implement functions to obtain the AST from a file, construct the blacklist and list of `COMMON` blocks inside the parse tree, and several get and convenience methods to abstract the direct access to the AST for the analysis. `dataflow.py` provides the whole analysis procedure along with several

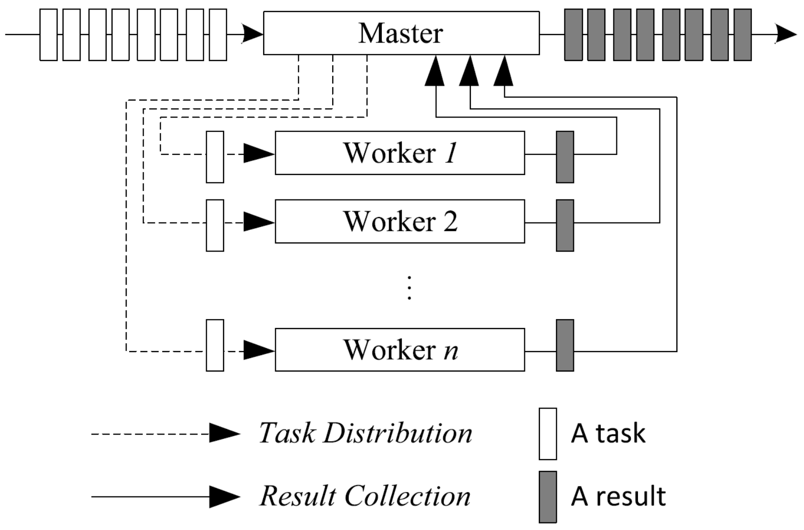

Figure 3.2. Master-worker model corresponding to the implementation of the dataflow analysis. From [Lin and Chung 2014, p. 20].

related functions. The analysis is based on functions offered by `ast_functions.py`, so there is no direct access to the AST by `dataflow.py`. The worker module provides error handling and initializes parsing and dataflow analysis procedures after being called by the main module with a path to the to-be-analyzed file.

Chapter 4

Architecture Reconstruction

In this chapter we explain the enhancements made to the Kieker architecture meta-model (Kieker Analysis Model) to model dataflow and how we construct an architecture from the dataflow information provided by the analysis tool. More information on specific details of the implementation can be found in the thesis of Yannick Illmann.

4.1 Enhancement of the Architecture Meta-Model

This section describes the parts of the Kieker architecture meta-model that are involved in modeling a dataflow-based architecture. Figure 4.1 shows these parts and their relationships to each other. In the following, we explain these relationships and the corresponding model level in more detail.

Type Model

The type model represents the highest referenced level (cf. Figure 2.4). Its classes store all the semantic information. The relevant classes in the scope of our work are *ComponentType*, *OperationType*, and *StorageType*, along side the mapping classes. To visualize files and packages we use the ComponentType class, because it allows us to link classes of the ComponentType with other classes of the ComponentType. Consequently, we are not only able to model relations between program components, i.e. individual files, but also between packages of program components, given the additional information is available. We do not have to modify any of the provided classes of the type model to model the dataflow of Fortran-written ESMs. In the context of Fortran, the classes of the type model have the following meaning: A *ComponentType* represents a Fortran file or a package of files corresponding to the directory they are stored in. Therefore, we store every file component in a package component. We declare every operation, e.g. a `SUBROUTINE`, as *ComponentOperation*. A `COMMON` block is stored as a *ComponentStorage*. Both, *ComponentOperation* and *ComponentStorage*, are stored in a related file component.

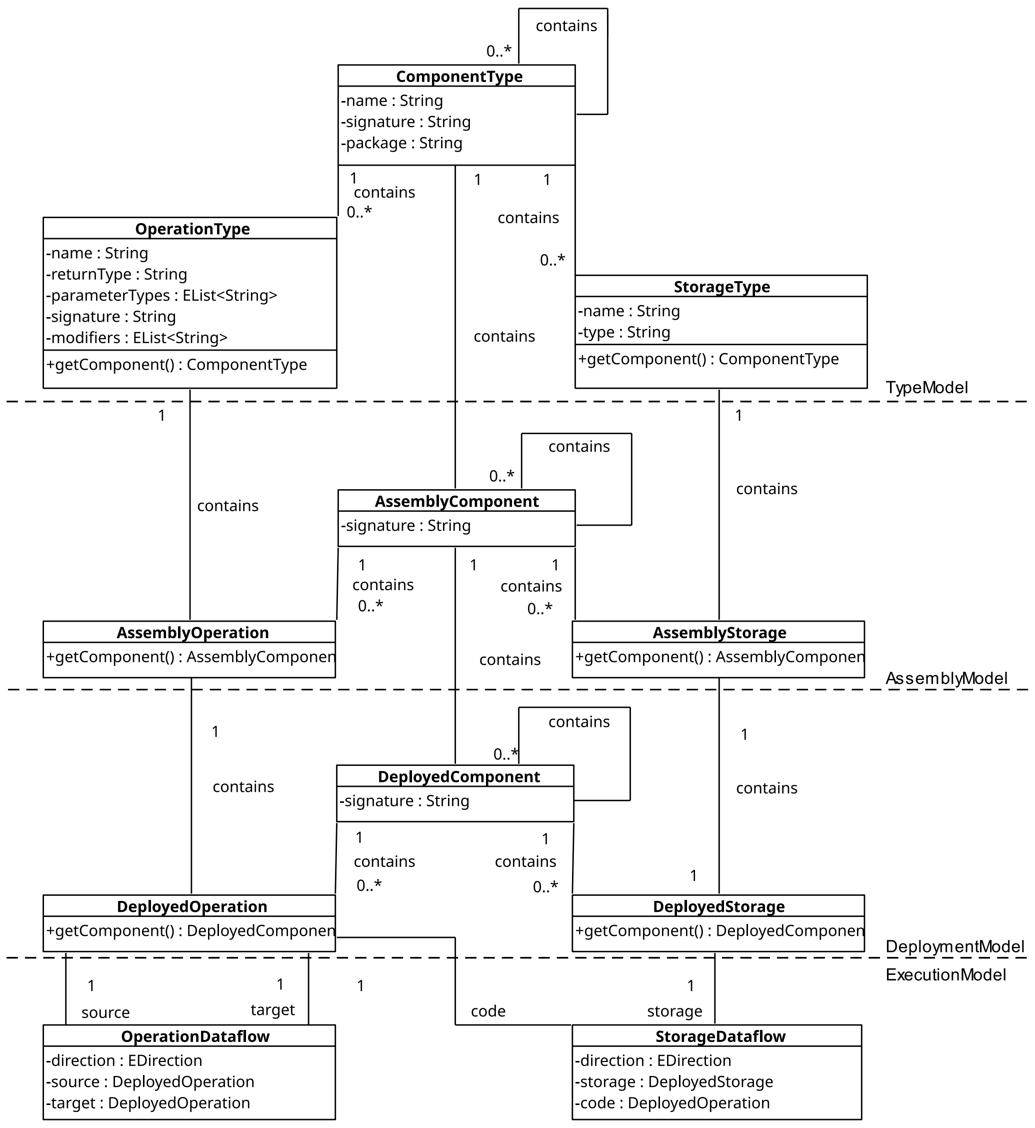

Figure 4.1. Kieker architecture meta model classes in UML relevant to model dataflow.

Assembly Model

The assembly model represents the second-highest level of model classes, containing classes of the type model level. In detail, the assembly model is contained within the type model, but stores by itself only minimal information. For each type model class, there is also a class on the assembly level, as we can see in Figure 4.1. More information on the assembly level classes can be obtained by referencing to the corresponding type model classes. Due to the highly connected classes, we made no changes on the assembly model level.

Deployment Model

The deployment model represents the next-lowest level after the assembly level. It is divided into different *DeploymentContexts*, but in our case we only handle a single context. The defined classes are syntactically similar to the classes of the assembly type. We specify *DeployedComponent*, *DeployedOperation*, and *DeployedStorage* with just basic attributes, because of the cross-referencing to classes of a higher level. We use the original definition of the deployment model and reference all Fortran packages, components, operations, and storages as shown in Figure 4.1.

Execution Model

The execution model represents the lowest and most essential level of the meta-model for dataflow modeling. We define all dataflow interactions between operations and storages at this stage. In contrast to the other levels, we had to add to and rework classes of the original model. Besides the required mapping classes, we add the class *OperationDataflow*. It contains an attribute *direction* of type *EDirection* and two references *source* and *target* of type *DeployedOperation*. The *EDirection* is a Java enumeration type declaring the dataflow interaction, i.e. read, write, or bidirectional dataflow. Therefore, the *OperationDataflow* class declares which operation accesses what operation and with what kind of interaction. The access is related to the deployment model by storing *DeployedOperation* classes (cf. Figure 4.1). Kieker's execution model already defines a storage dataflow class called *AggregatedStorageAccess*. It contains an attribute *direction* of type *EDirection* along with two references *code* of type *DeployedOperation* and *storage* of type *DeployedStorage*. To make the execution model more uniform, we refactored the class to *StorageDataflow*, but it still contains the same attributes and references.

The described model enables us to generate Java code using the EMF to create a fully-qualified dataflow-based meta-model of theoretically any Fortran application. From the semantics defined on type level down to actual data flow connections on execution level, the model offers a concept for storing and modeling data flow between Fortran program operations.

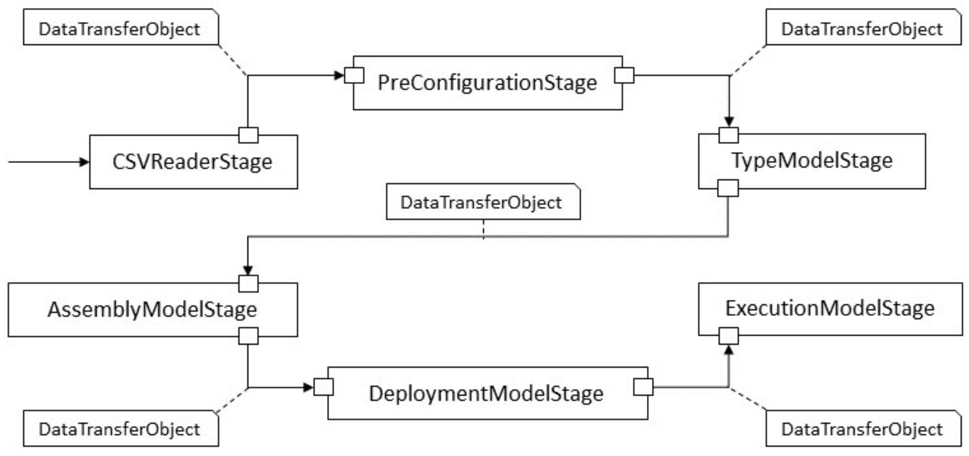

Figure 4.2. TeeTime Stage Concept.

4.2 Model Construction and Visualization

To construct an architecture from the extracted dataflow information of an ESM, we have to create instances of the specified classes of the architecture meta-model. Therefore, we use several TeeTime stages, which for each stage create corresponding objects. We configured the stages and connections within the *SAR* tool as we picture in Figure 4.2. The *CSVReaderStage* extracts information line by line from the tables created by the dataflow analysis and stores the information into *DataTransferObjects* (DTOs), which are passed further through the configured pipeline. The *PreConfigurationStage* determines whether the DTO refers to a storage or operation access. The following stages only read from the created DTOs and construct the corresponding model instances from it. The *TypeModelStage* creates components containing storages or operations, the *AssemblyModelStage* links type model components, the *DeploymentModelStage* links to an assembly model component, and the execution model created by the *ExecutionModelStage* stores the final dataflow connections using deployment classes.

By the *SAR* tool created models are stored in form of XMI files. These can be interpreted by the Kieker architecture visualization. We use the Eclipse Plugin *Kieker Architecture Visualization* that is based on the automatic layout engine of *Kiel Integrated Environment for Layout Eclipse RichClient*[1] (KIELER) to render and export the models as graphs.

[1]https://www.rtsys.informatik.uni-kiel.de/en/archive/kieler

Chapter 5

Analysis and Tool Evaluation based on the Earth System Model MITgcm

ESMs, such as MITgcm, are data-centric applications, which means that the core value of the application lays on computations of input data and the output of resulting data. To analyze the flow of data through the program, we perform a dataflow analysis and reconstruct the architecture of the ESM based on the extracted dataflow information. The analysis and reconstruction process involves all procedures and enhancements to tools presented in this thesis. The evaluation of the results enables us to assess the accuracy of the tooling.

In this chapter we apply the presented tooling to twelve predefined tutorial variants of the MITgcm and present the results. First, in Section 5.1, we describe, which metrics we use for the architecture evaluation. Further, in Section 5.2, we outline the general package structure of the MITgcm along with details on the visualization. Following, we analyze the recovered architectures of twelve tutorial variants and compare the architectures of two specific variants to each other. To reproduce our results, we provide a replication package [Ohlsen and Illmann 2022].

5.1 Metrics

We employ several metrics for the evaluation of the reconstructed architectures. To assess cohesion and coupling, we calculate incoming and outgoing edges on the operation and component levels. Further, we compute the total number of nodes and edges along with in- and outgoing edges for each package, which allows us to compare the package composition and coupling of different MITgcm variants. Besides that, we determine the overall model size and complexity based on [Allen 2002]. This enables us to analyze the complexity distribution among the tutorial variants and compare our findings to previously obtained results for call-graph-based architectures of the MITgcm.

To obtain the metrics, we use *MVIS* from the OceanDSL tools (cf. Section 2.4), which reads the models' XMI files and writes the metrics in separate files that contain the in- and out-degree on module and component levels and the metrics described by Allen [2002]. Additionally, we compute statistics with a Python script, such as the total number of nodes, edges, or in- and outgoing edges on package level, by analyzing the *MVIS* output.

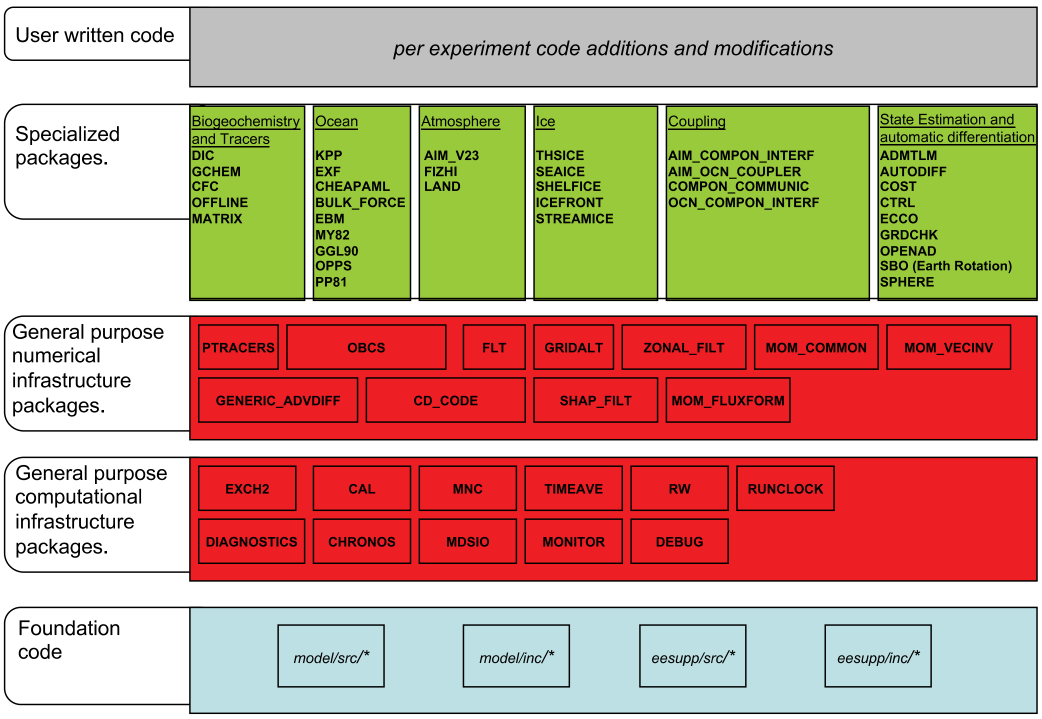

Figure 5.1. Package structure of the MITgcm. From [MIT 2022, p. 400].

5.2 Package and Component Visualization

In the visualization of the reconstructed architecture, we add the `COMMON-Component` to the model, which includes all `COMMON` blocks that are relevant to the dataflow. We model them this way, because the properties of `COMMON` blocks make it hard to assign them to a single existing component. The `COMMON` component does not correspond to any actual package or file within the model. Further, it is easy to understand and simple to implement. Alternative strategies for the modeling of `COMMON` blocks are, for example, that every block is a component by itself, but that could make the visualization rather confusing because of the high number of `COMMON` blocks. Another approach would be to group `COMMON` blocks in components by the components they are getting accessed by, or to assign `COMMON` blocks to a component if they are only used within the component.

Furthermore, we group components in the meta-model into other components that represent the different packages of the model. This has two benefits: First, the visualization is more structured and it is easier to differentiate between dataflow within components of the same functionality and dataflow between components of different functionality. Second, we are able to analyze incoming and outgoing edges on package level to analyze how packages integrate into the architectural model. To provide an overview of the purpose of the different packages, Figure 5.1 shows the hierarchy and use case of many of the MITgcm packages. The *numerical model* and *Wrapper* are located within the foundation code in the packages `model` and `eesupp`. Any additional functionality is added to the *numerical model* by different packages that can be individually configured, depending on the experiment.

5.3 Architecture Evaluation

In the following, we outline the results of the analysis of the reconstructed dataflow-based architectures. In Section 5.3.1, we present general findings, that apply to all analyzed variants. Additionally, in Section 5.3.2 and Section 5.3.3, we analyze the package composition and dataflow between packages of the example experiments *Barotropic Ocean Gyre* and *Held-Suarez Atmosphere* in more detail. Finally, in Section 5.3.4, we compare the example experiments and the results of their analysis to each other.

5.3.1 General Findings

The MITgcm includes a range of preconfigured model variants. We analyze twelve of the 14 tutorial variants, excluding `tutorial_tracer_adjsens` and `tutorial_global_oce_-optim`, because they are variations of `tutorial_global_oce_latlon`, which we include in the analysis. The reason for analyzing the tutorial variants is, that we expect a wide range of complexities, from relatively simple models to more complex simulations, but also many different application scenarios, from atmospheric circulations to modeling of biochemical processes in the ocean. Table 5.1 provides an overview of the analysis results, arranged in ascending order of complexity. The first immediate result is that we observe a correlation between complexity and size. And additionally, that complexity correlates more with the number of files, than the lines of code.

The complexity distribution, depicted in Figure 5.2, suggests that there is a tendency to lower complexity values, with an average complexity of 14095.623, but overall we can see a wide range of complexities. As the tutorial experiments should provide a comparatively simple introduction to the MITgcm, the tendency to lower complexities is expected.

Further, we set the complexity in relation to the size of each variant. Figure 5.3 shows the resulting graph along with two auxiliary lines that approximate the ratio between them. We can see that the model complexity grows linearly with the size with an average factor of above two. There is also the trend of increasing ratio with increasing model size, but we need more data points of more complex variants to verify that.

Figure 5.4 shows the results from the analysis of call-graph-based architectures of a number of predefined MITgcm variants, including some of the tutorial variants we analyzed. Although we do not have values for all tutorial variants, we can observe some general trends by comparison with the dataflow-based values. The complexity values and the ratios

Table 5.1. The table shows the results of the dataflow-based architecture analysis for each variant arranged in ascending order of complexity. It includes the number of files the variant consists of (NoF), the lines of code analyzed (LoC), the number of files that could not be analyzed due to parsing errors (PE), and the Allen-metrics size and complexity along with the ratio complexity to size.

Variant	Complexity	Size	Ratio	NoF	LoC	PE
`tutorial_barotropic_gyre`	11050.307	5829.528	1.896	573	1250049	2
`tutorial_global_oce_in_p`	11359.737	5937.735	1.913	575	1201556	3
`tutorial_cfc_offline`	11724.558	6560.581	1.787	608	1297817	2
`tutorial_global_oce_latlon`	13013.681	6776.793	1.920	652	1442637	4
`tutorial_plume_on_slope`	13240.058	6565.828	2.016	615	1358712	2
`tutorial_baroclinic_gyre`	13793.209	6762.359	2.040	626	1370341	2
`tutorial_rotating_tank`	14759.225	6926.631	2.130	627	1323800	3
`tutorial_advection_in_gyre`	14847.299	7140.617	2.079	657	1445661	2
`tutorial_deep_convection`	14985.395	7025.406	2.133	626	1369724	2
`tutorial_reentrant_channel`	15097.137	7510.071	2.010	670	1507343	2
`tutorial_global_oce_biogeo`	17447.907	8463.241	2.062	751	1712454	4
`tutorial_held_suarez_cs`	17828.967	8485.696	2.101	762	1542487	4

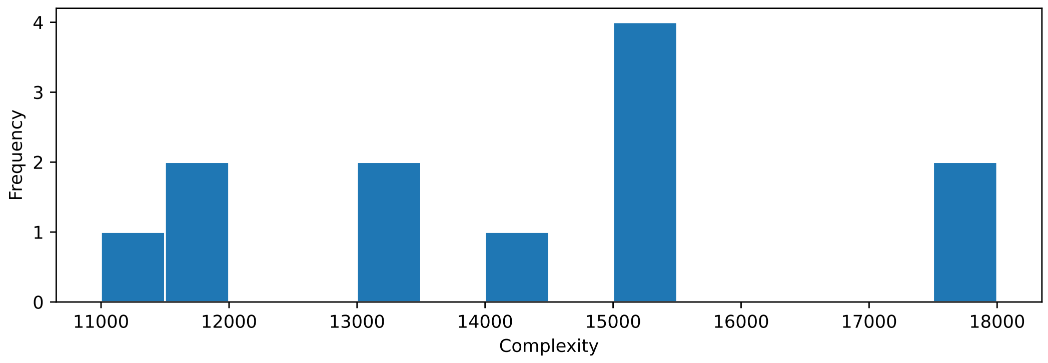

Figure 5.2. Complexity values rounded to 500 increments for the dataflow-based architectures of twelve MITgcm tutorial variants.

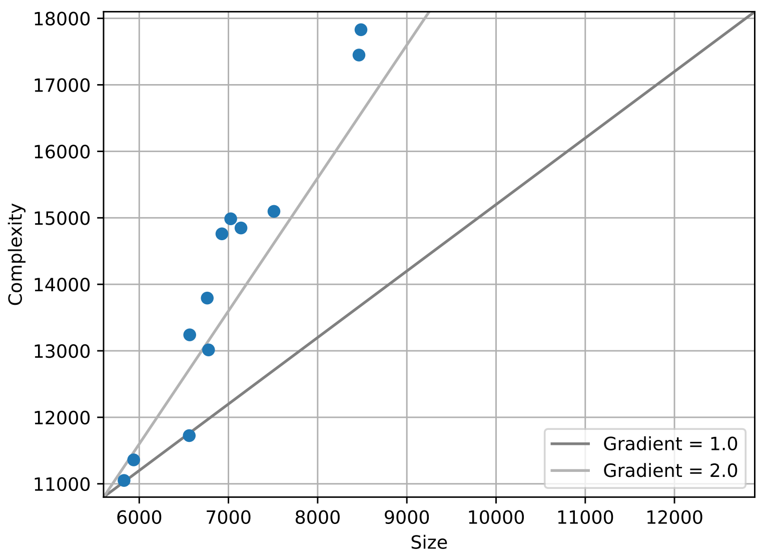

Figure 5.3. Relationship of complexity and size of the measurements for the dataflow-based architecture models of the MITgcm tutorial variants, including two auxiliary lines indicating the growth rate.

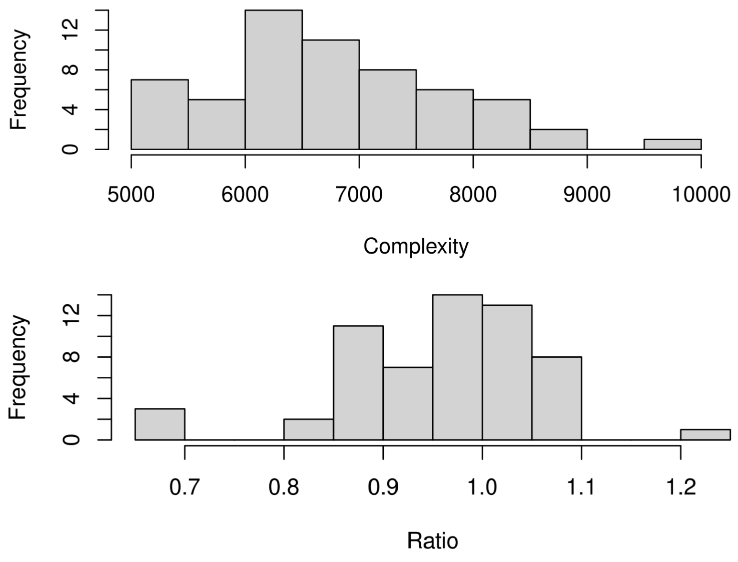

Figure 5.4. Complexity value distribution (top) and complexity to size ratio (bottom) for a number of MITgcm variants previously computed from recovered architectures based on a call graph by members of the Software Engineering Group at the CAU Kiel.

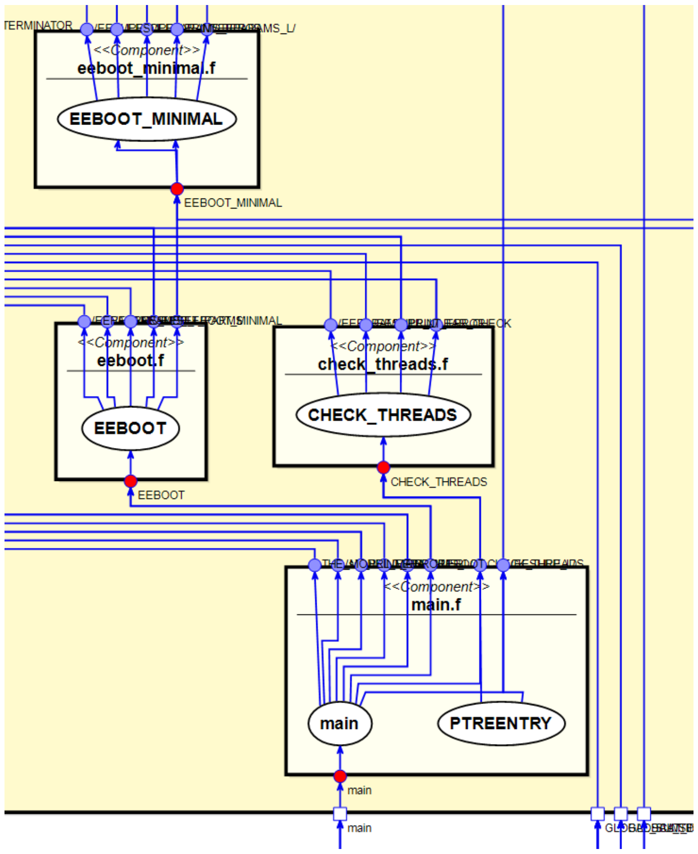

Figure 5.5. Excerpt of the `eesup` package from the visualized dataflow-based architecture of the MITgcm variant `tutorial_global_oce_biogeo` showing parts of the MITgcm startup sequence.

of complexity to size are about twice as high for the dataflow-based architecture. This means that a call-graph-based architecture has on average half the complexity compared to a dataflow-based architecture for the same size.

In the architectural model itself, we can observe parts of the MITgcm startup sequence (cf. Listing 2.1) for every variant. Figure 5.5 shows an exemplary excerpt of the `eesup` package from the architecture visualization of the MITgcm variant `tutorial_global_oce_biogeo`. The dataflow in the graph, starting from the component `main.f`, resembles the specified call sequence. Beyond invoking the files stated in Listing 2.1, `main.f` also writes data to the component `barrier.f`, which provides routines for multithreading. Other edges starting from `main.f` that leave the picture, lead to different `COMMON` blocks and `the_model_main.f`, which starts the numerical model.

5.3.2 Barotropic Ocean Gyre

The first variant of the MITgcm that we analyze in more detail is the example experiment *Barotropic Ocean Gyre* (`tutorial_barotropic_gyre`). It simulates a large wind-induced circulating ocean current in a scale of 1200 km^2 and a depth of 5 km. The experiment is the smallest among the analyzed variants and also has the lowest complexity.

Figure 5.6 depicts all ten MITgcm packages the variant consists of along with the number of associated edges divided into incoming and outgoing edges. The experiment only leverages packages of the core model along with some additional general purpose packages for computational and numerical infrastructure. Besides the foundation packages `model` and `eesup`, the numerical infrastructure package `pkggeneric_advdiff`, which provides subroutines for solving advection-diffusion equations, has the most dataflow connections. Not shown is the `COMMON-Component`, because it is no MITgcm package. The `COMMON-Component` of `tutorial_barotropic_gyre` has 213 incoming and 642 outgoing edges, which would rank it just below `eesup` in Figure 5.6. Parsing and thus the dataflow analysis failed on the files `find_alpha.f` from the `model` package and `gad_c4_adv_r.f` from `pkggeneric_advdiff`. This corresponds to 0.35% of all files that we could not analyze due to limitations of *fparser*.

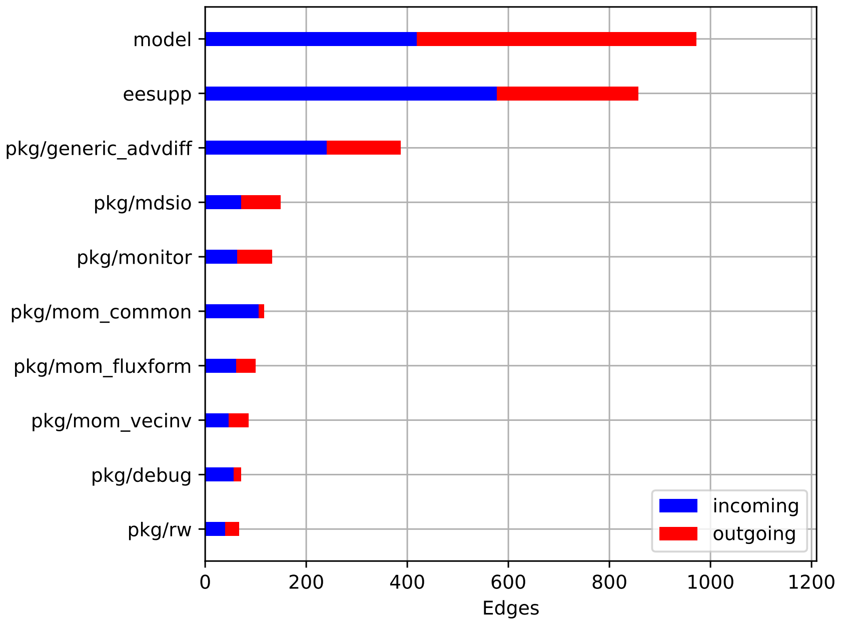

Figure 5.6. The number of incoming and outgoing edges related to dataflow per package of the MITgcm tutorial variant `tutorial_barotropic_gyre`.

5.3.3 Held-Suarez Atmosphere

The *Held-Suarez Atmosphere* (`tutorial_held_suarez_cs`) experiment simulates a simplified planetary atmospheric circulation. The simulation does not include elevation changes of the underlying land, uses simplified forcing and only models dry air processes. We include this experiment into the evaluation, because it is an atmospheric simulation, in contrast to the oceanic simulation of `tutorial_barotropic_gyre`, and additionally the most complex of the tutorial variants.

The variant comprises 13 packages, depicted in Figure 5.7 with their associated edges. The foundation packages `model` and `eesupp` make up the majority of dataflow edges. Followed by `pkgdiagnostics` and `pkggeneric_advdiff` with 445 and 409 edges respectively. The diagnostics package offers a wide range of predefined subroutines for diagnostic information that can be flexibly configured. All packages included are related to the hydrodynamical kernel of the MITgcm and thus require no additional physics packages, except numerical and computational expansions to the base model. The `COMMON-Component` of `tutorial_held_suarez_cs` has 306 incoming and 1127 outgoing edges, which makes it by over 280 edges the biggest component of the architecture. Parsing failed for two files from `model`, one file from `pkggeneric_advdiff`, and one experiment specific file `apply_forcing.F`. Two of the failed files also failed for `tutorial_barotropic_gyre`. Thus, we could not cover 0.52% of the total number of files for this variant.

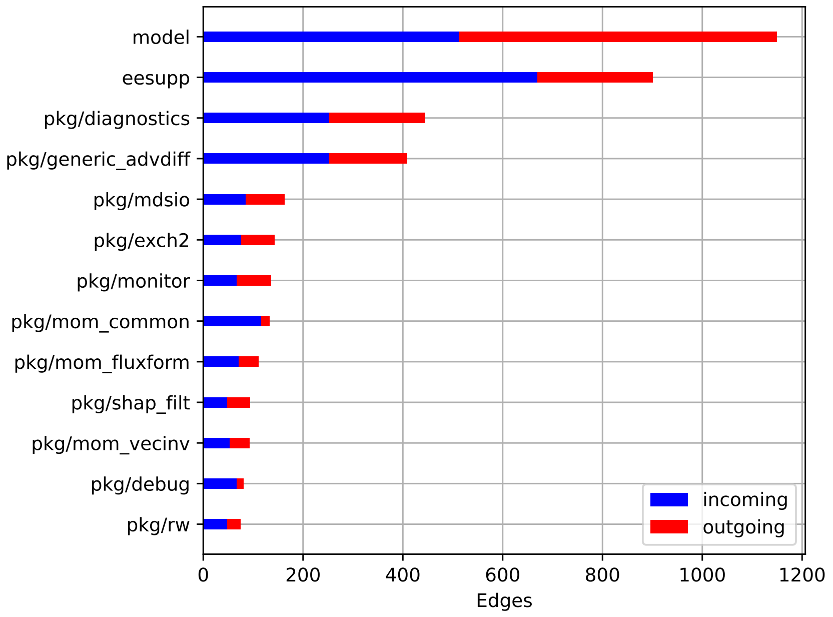

Figure 5.7. The number of incoming and outgoing edges related to dataflow per package of the MITgcm tutorial variant `tutorial_held_suarez_cs`.

5.3.4 Comparison

Even though the number of packages between the MITgcm variants `tutorial_barotropic_-gyre` and `tutorial_held_suarez_cs` only differs by three packages, the complexity of the latter is over 61% higher than of former. The biggest difference in the number of edges associated to packages between the two variants comes from the packages `model` and `pkgdiagnostics`. But by far the most significant difference lays in the use of `COMMON` blocks. The architectural model of `tutorial_held_suarez_cs` has 43.7% more *writes* to and 75.5% more *reads* from `COMMON` blocks. This increase comes to a large extend from the use of the diagnostics package, which reads many values from `COMMON` blocks and stores them in specially allocated arrays that are incremented for each value change. From comparing the edges associated to each package, we can also see that the dataflow to and from the execution environment setup package `eesupp` only increased slightly, with most of the increase coming from incoming edges. The dataflow to and from the `model` package of `tutorial_held_suarez_cs` increased nearly by the amount of edges added from the additional packages compared to `tutorial_barotropic_gyre`. The higher ratio of complexity to size of `tutorial_held_suarez_cs` with 2.1 compared to `tutorial_barotropic_gyre` with 1.9 indicates that the interconnectedness related to dataflow between components is higher in the former than in the latter.

Chapter 6

Discussion

During the evaluation process of the implemented dataflow analysis and recovered architecture, we gained valuable insights and results, which we discuss in this chapter. We divide the discussion into two parts. Firstly, in Section 6.1, we assess and interpret the results of the architecture evaluation. Secondly, Section 6.2 concerns the dataflow analysis and architecture reconstruction implementations and their limitations.

6.1 Architectural Results

In this section we discuss the results from the architecture evaluation in Section 5.3. The key finding are:

- ▻ The complexity correlates stronger with the number of files than the lines of code.
- ▻ The complexity scales with a factor of slightly above two compared to size.
- ▻ The complexity and complexity to size ratio of the dataflow-based architecture is about twice as high compared to a call-graph-based architecture.
- ▻ The use of `COMMON` blocks within individual packages has a strong influence on the overall model complexity.
- ▻ The dataflow-based architecture reproduces architectural design decisions of the MITgcm.

We begin by assessing the correlation between complexity and number of files. That complexity correlates less with the lines of code than the number of files analyzed is an expected result, because each analyzed file corresponds to a component in the model independent from the file size. The same applies to program parts inside files. Each program part corresponds to a operation inside a component, independent of the size, so many smaller subroutines add more complexity to the model than fewer bigger ones.

An explanation for the growth in complexity by a factor of two compared to the size is the introduction of the `COMMON-Component` to the reconstructed architecture. For every program part there are possibly two or more operations added, one for the program part itself and another one for every `COMMON` block that the program part uses (except it is already added) along with the associated edges. Further, we base the analysis of dataflow on the abstraction level of a call graph and potentially add additional edges to it. For example, if function `A` invokes a function `B` with an argument and saves the result, the corresponding call graph has one edge from `A` to `B`. However, the dataflow graph has two edges, one in each direction, while maintaining the same component count. This explains the doubling in complexity and complexity to size ratio compared to the call-graph-based architecture. Following from this, components or packages that make use of many `COMMON` blocks can have a large influence on the overall complexity because their influence essentially doubles. This is a drawback of the way we model `COMMON` blocks in architectural model, but other solutions would require more complex processing, modeling, and make the architecture more difficult to understand. Different solutions are, for example, that each `COMMON` block is assigned to separate component, all `COMMON` blocks that share relations to same components are grouped in a separate component, or that if a `COMMON` block is used by just one component, it is directly added to this component. On the other hand are `COMMON` blocks a key structure in interprocedural and inter-package dataflow of Fortran programs and thus have a large influence on the dataflow through the system, so our from our point of view the influence of the `COMMON-Component` to the architectural complexity is not exaggerated.

The dataflow-based recovered architecture produces insightful results for the analyzed tutorial variants of the MITgcm. From the recovered architecture we are able to reproduce some of the design decisions of the MITgcm software architecture. Firstly, we are able to follow the startup sequence of the model and also obtain additional information not specified in the MITgcm manual, for example which components are dependent on values set during the start-up sequence via *read* and *writes* to `COMMON` blocks. Secondly, we see the separation of concerns between the numerical model and the *Wrapper*. Between the most and least complex components there was only a slight increase in the number of edges related to the `eesupp` component, which resembles the *Wrapper*. Rather, the additional complexity introduced mostly propagated to the numerical model represented by the package `model`. Additionally, the comparison of the tutorial variants `tutorial_barotropic_-gyre` and `tutorial_held_suarez_cs` revealed the strong influence individual packages can have on the model complexity by the extensive use of `COMMON` blocks, as shown by the package `pkg/diagnostics`.

6.2 Limits of the Implementation

The implementation and evaluation process revealed some issues with the current implementation of the data flow analysis. The first set of issues is related to *fparser*. A problem that ocurred are parsing issues with some files that could not be solved. The parser requires syntactically correct Fortran code to function correctly. If there is any character, structure, etc. in the code, which the parser cannot identify, the parsing of the whole file fails. The issue could not be clearly identified due to improper error messages, nor could the problem be solved by, e.g., removing comments before parsing. Additionally, *fparser*

Listing 6.1. Example Fortran program consisting of a subroutine and a program.

```
SUBROUTINE SUB(A, B)
IMPLICIT NONE
INTEGER :: A
INTEGER :: B

B = A + 1
END SUBROUTINE

PROGRAM RUN
IMPLICIT NONE
INTEGER :: A
INTEGER :: B

A = 1
CALL SUB(A, B)
END PROGRAM
```

has issues differentiating between statement functions and arrays within the code on some particular occasions, which leads to arrays being declared as functions. This issue can be circumvented by tracking array declarations, but this adds avoidable complexity to the dataflow analysis. Also an issue is the performance of *fparser*. For large files the parsing and especially searching operations on the AST take an increasing amount of time. Even though the parallelization of the implementation reduces the severity for overall execution time drastically, depending on the available processor count, the performance should be considered for the possible evaluation of a different Fortran parser. Conceivable, although not Python-based, alternatives to *fparser* are *fxtran*[1] or the *Open Fortran Parser*[2] (OFP) that construct ASTs in form of XML files.

Limits of the dataflow analysis procedure are mostly related to the tight adaptation the structure matching currently has to MITgcm and UVic. Structures that are within the Fortran 77/90 specification, but not used within one of the two ESMs are currently not covered by the analysis. Missing are for example the program unit `BLOCK DATA`, explicit matching of call by value via `%VAL()`, or module includes and declarations. To track call by value and call by reference the architecture meta-model has to be adapted additionally, to model the difference between the two. Furthermore, the dataflow analysis does not track the use of pointers, nor does the meta-model. This comes in part from the abstraction level we chose, because pointers and some call by reference specific use cases require the tracking of individual variables through the system, which the current implementation cannot do. The example code presented in Listing 6.1 illustrates this limitation. The subroutine `SUB`

[1]https://github.com/pmarguinaud/fxtran
[2]https://github.com/OpenFortranProject/open-fortran-parser

increments its first argument `A` by one and assigns the result to its second argument `B`. The main program `RUN` now calls `SUB` with the initialized variable `A`, which has the value `1`. Because Fortran is by default call by reference, `SUB` initializes `B` with the value `2`. So after the execution of the subroutine both variables of `RUN` have a value assigned, despite only `A` is initialized within the program. The dataflow analysis will only identify `A`, but not `B`, because program parts, such as `PROGRAM` and `SUBROUTINE`, are analyzed in the scope of themselves and independently of others.

For the architecture reconstruction, we also identified points of improvement, besides evaluating different strategies for the modeling of `COMMON` blocks and the differentiation between call by value and call by reference. Foremost, the architectural model only models at most one edge for a *read* and another for a *write* between two components. For instance, if a component `C1` writes data five times to a component `C2` and `C2` writes data one time to `C1`, the model would just indicate both *write* relations with one edge each in the respective direction. To also indicate the number of dataflow related operations, the edges can be annotated with the respective cardinality of the edges determined by the dataflow analysis. Further, there is currently no way to see which intention the dataflow has. For example, it is unclear if an edge from `C1` to `C2` is a *write* from `C1` to `C2` or a *read* of `C2` from `C1`. A possible solution is to add an additional annotation to edges, e.g., `intent:write` or `intent:read` that specify the intention of the dataflow.

Chapter 7

Related Work

In this chapter, we present related work in the fields of dataflow analysis and software architecture recovery, including publications on which approaches presented in this thesis are partially based.

7.1 Dataflow Analysis

The foundations of the in Section 2.3 and Chapter 3 presented concepts of dataflow analysis are based on the work of Fosdick and Osterweil [1976], who present an approach to dataflow analysis in the context of software reliability. In contrast to dataflow analysis in the context of architecture recovery, the analysis is focused on identifying the liveness and availability properties of individual variables, rather than identifying the relations between program parts in terms of dataflow. In addition, the article introduces the necessary preliminaries of graph theory to perform a graph-based dataflow analysis as well as the use of regular expressions to classify dataflow patterns and detect dataflow anomalies that indicate potential errors in the program.

Söderberg et al. [2013] present an approach for the implementation of intraprocedural control- and dataflow analyses at the abstract syntax tree level. In contrast to our approach, their's models the dataflow on the AST itself, instead of constructing a separate graph. Also, their evaluation is based on dead assignment analyses of Java programs by extending the *JastAdd Extensible Java Compiler* (ExtendJ) and is thus motivated by software reliability.

Atkinson and Griswold [1998] describe a demand-driven technique to analyze the dataflow of program slices from large software in the presence of pointers in the C programming language. Their goal is to develop a tool that slices large programs, analyzes the dataflow of the included pointers, and thereby assists program understanding. The dataflow analysis of pointers is achieved through a points-to analysis during the construction of a CFG, which is the basis for the dataflow analysis of the program slice. Since pointers can also be used in Fortran, the techniques described are theoretically relevant for our work. However, pointers are rarely used in Fortran and not at all in the ESMs MITgcm or UVic. Further, the construction of a CFG is required to include pointers in the dataflow analysis. That is something we want to avoid because it is a computationally intensive task, which does not provide much benefit to the accuracy of the recovered architecture of the ESMs.

7.2 Architecture Recovery

Eixelsberger [1998] describes his experiences recovering a reference architecture of a system family in a case study with a train control system. The described approach consists of three main steps: First, construction of different architectural views based on tool-supported source code analysis, design documents, and domain knowledge. Second, identification of architectural elements within the architectural views. And finally, the construction of an architectural representation of the analyzed software. The case study has the goal to identify architectural degradation through a potential mismatch between the *as-is-architecture* and *as-should-be-architecture*. Many aspects of the approach are similar to the approach taken by the OceanDSL project, for instance, combining different architectural views obtained from different approaches to architecture analysis into an architectural model. Only the ultimate goal deviates from identifying architectural degradation by Eixelsberger [1998] to architecture comprehension and assessment of modularization in the case of the OceanDSL project.

Guamán et al. [2020] propose a reference process for architecture reconstruction and reverse engineering called *Software Improvement in the Reconstruction of Architectures* (SIRA), which is based on the *Software and Systems Process Engineering Meta-model* (SPEM) of the OMG. It consists of the four main steps preprocessing, extraction, analysis, and visualization. During the preprocessing phase, relevant code characteristics and potential noise through classes that are not part of the system are identified and taken into account in later steps. In the extraction phase, features and classes of the software are extracted via documentation, static, or dynamic analyses. The analysis phase involves the identification of design patterns and architectural styles based on the extracted features and further static and dynamic analysis of the software. The last step is the visualization of the architecture to assess the results with domain experts and to determine a suitable visual representation of the architecture. SIRA combines and formalizes many approaches to software architecture reconstruction into a single reference process. Both the work of Eixelsberger [1998] and that of OceanDSL fit into the scheme defined by the reference process. Although SIRA does not provide much benefit to previous and running projects, it offers guidance for future projects, simplifies planning, and thus helps to increase the adoption of architecture reconstruction by the industry.

Chapter 8

Conclusions and Future Work

In this chapter, we summarize our work on dataflow analysis and architecture reconstruction. We provide a short overview of key findings of the analysis of the dataflow-based reconstructed architectures. Finally, we give an outlook on possible future work.

8.1 Conclusions

During the course of this thesis, we developed a procedure to extract interprocedural dataflow from an abstract syntax tree. Further, we implemented a tool that applies this procedure to the Fortran source code of the earth system model MITgcm. To reconstruct the dataflow-based software architecture of the MITgcm, we had to enhance the Kieker architecture meta-model and adapt the tooling provided by the OceanDSL project. This ultimately enabled us to analyze the reconstructed architecture based on the complexity, size, and number of edges connecting individual components of the architectural model. In more detail, we analyzed twelve of the predefined tutorial variants of the MITgcm and compared the results, i.a., to metrics obtained from call-graph-based architectures of the MITgcm. The key findings are, that `COMMON` blocks play a significant role in interprocedural dataflow of the MITgcm variants and have considerable influence on the complexity of the reconstructed architectures, even if only a single or a few components use them extensively. Additionally, we were able to identify architectural design decisions of the MITgcm in the reconstructed architecture. The tool and architecture evaluation process also revealed points of improvement for the modeling of dataflow and dataflow analysis procedure, i.a., parsing errors and loss of information in the architectural model related to the number of edges and intention of dataflow.

8.2 Future Work

A major goal for the future is to reinforce and generalize the dataflow analysis procedure to cover more Fortran-specific constructs and subtleties. The current implementation is very specific to the MITgcm, UVic ESCM, and Fortran 77/90. For example, the import and declaration of modules or the use of pointers are currently not recognized by the tool. To apply the tool to other ESMs, such as the *Shallow water model*[1] (SWM) of the *GEOMAR Helmholtz Centre for Ocean Research Kiel*, unsupported Fortran features have to be identified and the analysis implementation enhanced to cover them.

Besides that, the use of a different parser for Fortran code has to be evaluated due to in some cases unresolvable parsing errors of *fparser*. These errors propagate into the model and result in missing components and edges.

Furthermore, the accuracy of the architecture meta-model has to be enhanced by modeling of the intention of dataflow, the cardinality of edges, and the differentiation between call by value and call by reference.

Finally, the dataflow-based architecture has to be combined with the control-flow-based architecture of the ESMs to get a more complete view of the actual software architecture of the analyzed models. Therefore, the *SAR* tool has to be enhanced to read both, control flow and dataflow information, or the *model operation* (MOP) tool of the OceanDSL tools can be used to merge the control-flow- and dataflow-based graphs.

[1] `https://git.geomar.de/swm/swm`

Bibliography

[Allen 2002] E. B. Allen. Measuring graph abstractions of software: An information-theory approach. In: *Proceedings eighth IEEE symposium on software metrics*. IEEE. 2002, pages 182–193. (Cited on page 23)

[Atkinson and Griswold 1998] D. C. Atkinson and W. G. Griswold. Effective Whole-Program Analysis in the Presence of Pointers. *ACM SIGSOFT Software Engineering Notes* 23.6 (1998), pages 46–55. (Cited on page 37)

[Backus and Heising 1964] J. W. Backus and W. P. Heising. Fortran. *IEEE Transactions on Electronic Computers* 4 (1964), pages 382–385. (Cited on page 5)

[Cooper et al. 2004] K. D. Cooper, T. J. Harvey, and K. Kennedy. *Iterative data-flow analysis, revisited*. Technical report. 2004. (Cited on page 6)

[Eixelsberger 1998] W. Eixelsberger. Recovery of a reference architecture: A case study. In: *Proceedings of the third international workshop on Software architecture*. 1998, pages 33–36. (Cited on page 38)

[EMF Website 2022] EMF Website. *Eclipse Modeling Framework (EMF)*. Accessed 2022-08-30. Eclipse Foundation, 2022. URL: `https://www.eclipse.org/modeling/emf/`. (Cited on page 8)

[Erdos and Sneed 1998] K. Erdos and H. M. Sneed. Partial comprehension of complex programs (enough to perform maintenance). In: *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*. IEEE. 1998, pages 98–105. (Cited on page 1)

[FORTRAN 2010]. *FORTRAN 77 Language Reference*. Accessed 2022-09-03. Oracle Corporation, 2010. URL: `https://docs.oracle.com/cd/E19957-01/805-4939/index.html`. (Cited on pages 5, 15)

[Fosdick and Osterweil 1976] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys (CSUR)* 8.3 (1976), pages 305–330. (Cited on pages 6, 12, 37)

[Ghasemi et al. 2015] M. Ghasemi, S. M. Sharafi, and A. Arman. Towards an analytical approach to measure modularity in software architecture design. *Journal of Software* 10.4 (Apr. 2015), page 465. (Cited on page 1)

[Guamán et al. 2020] D. Guamán, J. Pérez, J. Diaz, and C. E. Cuesta. Towards a reference process for software architecture reconstruction. *IET Software* 14.6 (2020), pages 592–606. (Cited on page 38)

[Hasselbring and van Hoorn 2020] W. Hasselbring and A. van Hoorn. Kieker: A monitoring framework for software engineering research. *Software Impacts* 5 (June 2020). DOI: 10.1016/j.simpa.2020.100019. (Cited on page 8)

[Janni et al. 1986] J. F. Janni, R. Berry, J. Burgio, G. Cable, and R. Conley Jr. *FORTRAN-77 Computer Program Structure and Internal Documentation Standards for Scientific Applications.* Technical report. AIR FORCE WEAPONS LAB KIRTLAND AFB NM, 1986. (Cited on page 5)

[Lin and Chung 2014] X.-Y. Lin and Y.-C. Chung. Master–worker model for MapReduce paradigm on the TILE64 many-core platform. *Future Generation Computer Systems* 36 (2014), pages 19–30. (Cited on page 18)

[MIT 2022] MIT. *MITgcm User Manual*. Accessed 2022-08-30. Massachusetts Institute of Technology, 2022. URL: https://doi.org/10.5281/zenodo.1409237. (Cited on pages 3–5, 24)

[Ohlsen and Illmann 2022] S. Ohlsen and Y. Illmann. *Theses Artifacts: Dataflow Analysis of Earth System Models*. Version 1.3. Sept. 2022. DOI: 10.5281/zenodo.7116165. (Cited on page 23)

[Peterson 2009] P. Peterson. F2PY: a tool for connecting Fortran and Python programs. *International Journal of Computational Science and Engineering* 4.4 (2009), pages 296–305. (Cited on page 7)

[Rasool and Asif 2007] G. Rasool and N. Asif. Software architecture recovery. *International Journal of Computer and Systems Engineering* 1.4 (2007), pages 939–944. (Cited on page 1)

[Sant'Anna et al. 2007] C. Sant'Anna, E. Figueiredo, A. Garcia, and C. J. P. Lucena. On the modularity of software architectures: a concern-driven measurement framework. In: *Software Architecture*. Edited by F. Oquendo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pages 207–224. (Cited on page 1)

[Söderberg et al. 2013] E. Söderberg, T. Ekman, G. Hedin, and E. Magnusson. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Science of Computer Programming* 78.10 (2013), pages 1809–1827. (Cited on pages 6, 37)

[Van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. Boston, Massachusetts, USA, April 22-25, 2012: ACM, Apr. 2012, pages 247–248. DOI: 10.1145/2188286.2188326. (Cited on page 8)

[Wulf et al. 2017] C. Wulf, W. Hasselbring, and J. Ohlemacher. Parallel and Generic Pipe-and-Filter Architectures with TeeTime. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE. 2017, pages 290–293. (Cited on page 9)

[Zhang and Goddard 2005] S. Zhang and S. Goddard. xSADL: an architecture description language to specify component-based systems. In: *International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II*. Volume 2. 2005, 443–448 Vol. 2. DOI: 10.1109/ITCC.2005.303. (Cited on page 1)