Data Flow Analysis of the University of Victoria Earth System Climate Model

Yannick Illmann

Bachelor Thesis
September 27, 2022

Software Engineering Group
Department of Computer Science
Kiel University

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

Selbstständigkeitserklärung

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

Kiel,

_____

Abstract

Most of the earth system model (ESM) simulations implemented in the latest 90s are mainly implemented in the programming language Fortran 90/95. ESMs in general are often aged software applications, containing problematic architecture styles, but they are still evolving over time. To fully understand such data-centric applications every call and data flow relationship needs to be considered. But language characteristics of Fortran make it hard to reconstruct relationships between components or operations.
The OceanDSL project, launched and hosted by the GEOMAR Kiel and Christian-Albrechts-University Kiel, wants to provide solutions addressing the architecture recovery problem, among other challenges. OceanDSL declares a goal, wanting to analyze ESMs and remodulate them to make updates, improvements and maintainability more possible. As a part of OceanDSL this thesis covers an analysis part for identifying interprocedural data flow of an ESM, namely "The University of Victoria Earth System Climate Model" (UVic ESCM), published by the University of Victoria, Canada. The main focus is to implement a tool recovering any flow of data from a former parsed abstract syntax tree (AST) and visualize it by creating different meta-models. Multiple versions of the UVic ESCM are compared and evaluated with metrics such as model complexity. Results show a rising complexity due to added components and data flow connections.

Contents

Chapter 1

Introduction

Climate research becomes more important every day. In times when people all over the world want to fight climate change, a lot of simulations and calculations need to be made to predict any kind of change in our climate system or evaluate how the human race can prevent any upcoming climate disaster. Since the beginning of scientific research, scientist collect data by performing experiments. But to understand a complex ecological system like the Earth, data from every possible place on earth needs to be considered. Some of the data is collected by scientist going on field trips and start a series of measurements. However, these trips costs an organization a lot of money and only represent data of one specific time frame. Therefore, an experiment is set up over multiple months, sometimes years. Hence, any prediction of a future development is limited to the possibilities of financial support and time a team has.
An alternative to field trips are simulations using programmed models of an ecological system. Different models specify on different ecological systems from all over the world, making it possible to simulate any kind of change and effects on virtual bases. In example, *UVic-ESCM* allows to generate scientific data regarding ocean and atmosphere coupling.
ESMs describe a complex and data-centric software. They are designed to be durable with the need to be regularly updated. Durability of application include the possibility that software developer may change over time. Therefore, every new contributor needs time to learn and understand an ESM implementation. Mostly ESMs are designed to address very specific and complex ecosystems or at least parts of them. However, every provided documentation is inadequate. As a result, every onboarding process is very time consuming.
In order to modernize earth system models, the project OceanDSL was created. Together with GEOMAR Kiel members of 'Software Engineering' group at the Christian Albrechts University Kiel work on visualization of existing architecture and defining domain-specific languages and tools to make it possible "to develop, evolve and use ocean biogeochemical models"[*OceanDSL Domain-Specific Languages for Ocean Modeling and Simulation*]. To generate architectures of the provided earth system models, a suite of tools is used to retrieve data in the first step and store and present them in the following steps. Up to now, a static analysis is implemented and used to collect perform a call graph analysis of ESMs. But to perform a successful reverse engineering process data flow needs to be considered too, to identify the degree of complexity. Based on this information, it is possible to remodulate an ESM to improve its performance, stability and maintainability.

1.1 Motivation

Most of the ESMs are written in Fortran code. To understand a models design any call relationship next to data flow connections needs to be considered. Due to the physical background of any simulation every data flow relationship is immensely important to keep a model valid. Fortran is a programming language, describing itself as high performance language usable for computationally intensive applications. Due to its strongly typed character, storage possibilities and natively parallel strategy, it is used in the context of science or engineering [*Fortran High-performance parallel programming language*]. However, regarding implementation and complexity, it is not possible to recreate any architecture of a given model or application by looking at the source code only.
Most of the code is written in so-called `SUBROUTINE` or `FUNCTION` calculating and storing data in `COMMON BLOCK`s or other operations. One or multiple operations are placed in a Fortran file, in the following chapters referred to as "file component(s)". Despite, a `COMMON BLOCK` is referenced in only an operation or main calculating part. It represents a shared memory approach throughout any calculating unit in the whole application. Fortran makes it possible to create local variables for every `COMMON BLOCK` in an operation itself, without concerning about naming conventions from other operations, accessing the block as well. As a result, the fastest way to perform an architecture recovery based on project structure and source code is a static analysis. The project OceanDSL already implemented multiple tools and scripts to achieve a call graph recovery result. Up to now only no specific data flow analysis is implemented in a static analysis tool. In the following a tool to recover data flow information based on Fortran source code is presented next to a possibility to visualize our results by creating different meta models.
This thesis is a joined bachelor thesis by Simon Ohlsen and me. We jointly developed the tool to analyze data flow in ESMs and present them in distinct bachelor thesis. The architecture recovery part is explained in detail in this document. However every implementation detail regarding data flow analysis is covered in the thesis "DataFlow Analysis of the Earth System Model MITgcm", by Simon Ohlsen.

1.2 Goals

The main goal of the thesis is split in three subgoals containing two implementation steps and an evaluation to wrap it up. The first and second goal is completed by myself and my lab partner Simon Ohlsen. Together, we designed and implemented on time-shared slots to achieve best results.

G1: Implementation of Data Flow Analysis based on an Abstract Syntax Tree

In the beginning, we present our approach to design and implement an analysis tool, collecting data flow information and store it in a self defined interface. We define strategies regarding performance and data flow, already given next to our own concept. The data, which is analyzed, is provided by the FParser tool, generating an AST. In order to collect data, we need to define different Python modules to retrieve certain information from the tree and assign the information to data flow definitions. By completing G1 a fully functional tool shall be created to store data flow of any earth system model written in Fortran.

G2: Enhance Architecture Meta Model and Visualize Architecture

The second step presents a way to handle the provided data of G1. It shall be possible to read from the interface and create a meta model representing all provided data flow information. Therefore, the Kieker Architecture Model is extended to design a basis to create a meta model visualization. All the code written for G2 is implemented in an already given Java tool from the tool collection of OceanDSL. By completing G2, a fully functional tool extension is created to reconstruct the data flow architecture of a given ESM written in Fortran.

G3: Apply Implementation to UVic Earth System Model

To test our tool, the implementation is applied to the UVic ESCM to use the results we generated and analyze architecture particularities. In addition, information is gathered to evaluate the reliability and correctness of our tool. Furthermore, it is explained how the tool can be updated to collect more precise data.

1.3 Document Structure

First Chapter 2 presents our used tools and theories. It explains specific terms which are be referenced in the following chapters. Second, Chapter 3 presents the first part of our tool regarding data flow analysis on an abstract syntax tree. However, in this document only a concept is given. The main presentation is covered in the thesis "Data Flow Analysis of the Earth System Model MITgcm", by Simon Ohlsen. Next Chapter 4 introduces the second part of the tool. Despite to Chapter 3 the concept and implementation is explained in more detail. The chapter covers the data handling of our defined interface towards a meta-model, using the TeeTime framework. Afterwards, the presented tool is applied to the UVic Earth System Model in Chapter 5. It contains all setup configurations next to a detailed evaluation of the model itself. In the end of the document, Chapter 6 reviews the implementation and

analysis is presented. It explains how different design decisions cause eventual side-effects and how reliable the analysis results are. Following the presentation of related work in Chapter 7 the document closes with a conclusion and future work references in Chapter 8.

Chapter 2

Foundations and Technologies

To accomplish our goals and achieve valuable results we used the following tools, techniques and definition. It is to mention that we didn't implement any paper theory only. However, we used literature such as [Fosdick and Osterweil 1976] to understand how data flow works and how it is possible to adapt the theories to our own requirements to generate a valuable visualization.

2.1 University of Victoria Earth System Climate Model

The *University of Victoria Earth System Climate Model* (UVic ESCM)[1] is an "intermediate complexity Earth System Climate Model" [Weaver et al. 2001, p.363]. It was developed to combine and better understand feedback loops and processes of the earth in longer timescales. The main simulation is focused on ocean and atmosphere interaction and resulting value changes in both ecosystems. However, it only represents a simple model of the atmosphere, the ocean is designed in full three-dimensional general circulation models. According to the developer's philosophy, the "horizontal ocean gyre transports and subduction processes are fundamental to the stability and variability of the thermohaline circulation" [Weaver et al. 2001, p.363]. Overall, UVic ESCM is a combination of multiple smaller models influencing atmosphere and ocean. For instance, a glaciological model was designed and implemented at the University of British Columbia. Also, the *Optimality-based Plankton Ecosystem Model* (OPEM[2]) is a subcomponent of the UVic ESCM. According to the readme file, it introduces the use of stoichiometry for phytoplankton, diazotrophs, and detritus to the ESCM and allows better ocean configuration and more scenarios and experiments to simulate.

2.2 Fortran

Fortran is widely used scientific programming languages. Its imperative, procedural, and high-performance design makes it ideally suited for numeric computation and simulations. Since its first version published back in 1957 [Backus and Heising 1964], several new

[1] http://terra.seos.uvic.ca/model/
[2] https://git.geomar.de/open-source/uvic-updates-opem

releases added increasing functionality up to object-oriented and concurrent programming capabilities. 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 UVic ESCM.

In Fortran 77/90 a 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 operations to exchange data without using arguments [FORTRAN 77].

2.3 Data Flow 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 data flow 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 $G_{\mathrm{F}}(N, E, n_0)$, 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 data flow through the program at statement level. One simple way to model data flow 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 2.1 shows the result of an exemplary data flow analysis using the *worklist algorithm* described in [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 like one of the exit nodes. The

out-set of a node is the union of the successors in-sets. Then 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 did not use this concrete approach in our implemented analysis because it is graph-based, but it resembles our understanding of data flow and lays the foundation for our own approach described in Chapter 3.

```
1  # B1:
2  a = 42
3  b = 2
4  c = 0
5  x = 1
6  if a = b then
7
8  # B2:
9      c = a + b
10
11 # B3:
12 else:
13     x = a - b
14
15 # B4:
16 return c * x
```

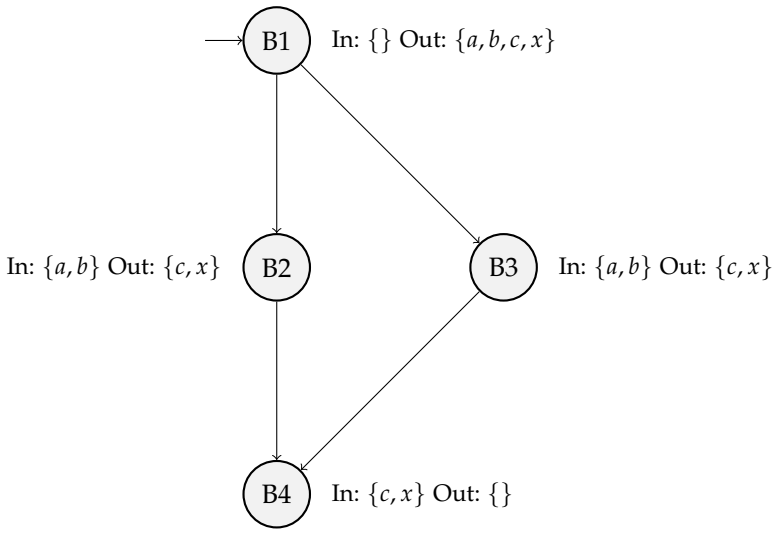


Figure 2.1. A simple algorithm on the left with its corresponding CFG on the right-hand side. Basic blocks are marked by the comments in the code. Annotations of nodes represent the respective in- and out-sets.

2.4 Analysis Tools

fparser is a Python package implementing a parser for the programming language Fortran from Fortran 66 to Fortran 2008. It includes fparser1 and fparser2. While fparser1 is deprecated, 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 (PyPI). We use fparser2 to obtain an abstract syntax tree (AST) from the Fortran code of the

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

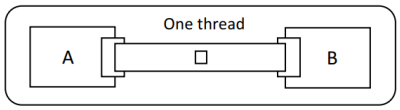
MITgcm and the built-in functions to navigate the AST.

OceanDSL Tools is a suite of software tools developed in the context of project OceanDSL. For our work the tools *SAR* (static architecture reconstruction), *MVIS* (model visualization) and ESM Coupling Analysis [3] are most important. The former reconstructs an architecture from calls and data access. The latter is the one we are going to adapt to cover a complete data flow analysis of an ESM.

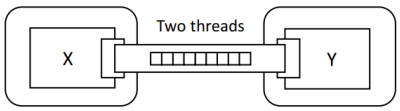
The *TeeTime Framework*[4] is a pipe and filter framework designed at the Software Engineering group from the Christian Albrechts University Kiel [Wulf et al. 2014; 2017]. Its purpose is to improve a commonly used architecture style and provide a research tool for multiple use cases. Due to the possibility of modification and extension of the framework, it is easy to adapt and implement. Furthermore, a goal of the framework's design is to allow more than one input streams, making it usable for more complex analysis and therefore interesting for more complex experiments.
In detail, the TeeTime Framework implements a multiprocessing style using a concept of *stages* and *pipes* [Wulf et al. 2014]. A stage by definition is "a processing unit within a pipeline" [Wulf et al. 2014, p. 5] handling incoming data on an input port and sending processed data with an output port. A pipe by definition "connects an output port of one stage with an input port of another stage" [Wulf et al. 2014, p. 5] allowing the elements to pass multiple stages. The main design idea is to build stages in parallel within one thread or separate threads, influencing the type of pipe connecting two stages. [Wulf et al. 2017] declares synchronized and unsynchronized pipes for two use cases. First, a design, where a thread is containing two stages, the connecting single element pipe is unsynchronized for intra-thread communication. Second, a design regarding one thread for each stage connected by a single-producer/single consumer pipe, see figure 2.2. It is defined as synchronized for inter-thread communication. To configure the pipeline setup, an separate analysis configuration class is implemented [Wulf et al. 2014].
One of the main characteristics of the pipe communication are typed ports, resulting in less time-consuming debugging. Additionally, TeeTime uses the passive/active stage theory of [Buschmann et al. 1996]. By declaring a stage as active, TeeTime automatically builds the thread concept early discussed containing actives stages followed by the right passive stage. Hence, the framework is able to manage to set up, maintain and terminate threads, without any further configuration by a developer [Wulf et al. 2017].

The *Eclipse Modeling Framework* is a code generation tool for structured data models. It implements UML Core packages and uses XMI data files to describe meta-models structured like the Meta Object Facility (MOF). It has similarities to the Essential MOF (EMOF) such as its goal "to allow simple meta-models to be defined by simple concepts" [*OMG*

[3] https://cau-git.rz.uni-kiel.de/ifi-ag-se/oceandsl/esm-coupling-analysis
[4] https://sourceforge.net/projects/teetime/



(a) Intra-thread communication with an unsynchronized, single-element pipe



(b) Inter-thread communication with a synchronized, bounded single-producer/single-consumer pipe

Figure 2.2. Thread Pipe Concept of [Wulf et al. 2017]

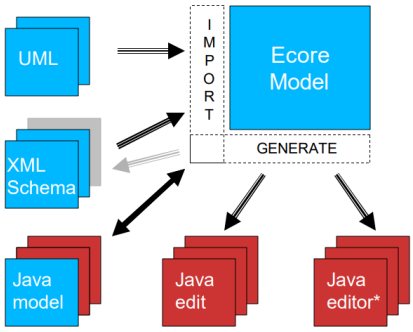


Figure 2.3. EMF Architecture [Merks et al. 2003]

Meta Object Facility (MOF) Core Specification] but extend it by an automated *EMF.Codegen*, a generation facility [*Eclipse Modeling Framework (EMF)*]. The framework uses an EMF Model, which is defined as "specification of an application's data" [Merks et al. 2003, p. 9]. In detail, it specifies relationships of objects containing attributes and operations, mostly written in UML but usable in XML/XMI schemas or Java interfaces. Based on an *Ecore architecture meta-model* the framework is able to generate reliable code, without needing the developer writing a single line of Java code, see Figure 2.3. It only requieres an installed version of Eclipse to run for the contained Java editor features. For further information on the generation techniques, see [Merks et al. 2003, p. 15-23].

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]. In the scope of the thesis, parts of Kieker definitions and tools are used, such as the *architecture meta-model* and *architecture visualization* by Kieler.

The *architecture meta-model* is an Ecore architecture model specified in Kieker repository[5]. It declares all in all seven submodels including different visualization possibilities. Four of the models are directly related by cross-references to three classes, see figure 2.4.

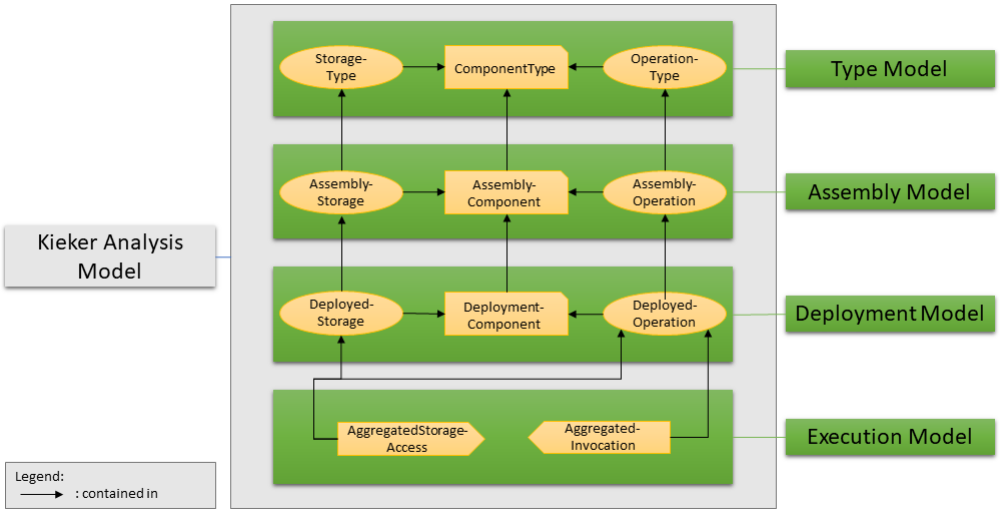


Figure 2.4. Kieker analysis Ecore architecture meta-model - concept graphic

The *architecture visualization* is accomplished by Kieler [6]. It uses imported XMI meta-models to create different diagrams depending on the model type. Kieler is used as an Eclipse Plugin maintained and managed by the Real-Time and Embedded Systems group. It provides an extra view window to show the diagrams created.

[5]GitHub: https://github.com/kieker-monitoring/kieker
[6]https://www.rtsys.informatik.uni-kiel.de/en/archive/kieler

Chapter 3

Data Flow Analysis on an Abstract Syntax Tree

This chapter presents a concept of a data flow analysis approach based on an AST structure. It represents a theoretical basis for implementing an ESM Data Flow Analysis tool, which generates multiple output files, containing valuable data. Further implementation information is presented in the Thesis "Data Flow Analysis of the Earth System Model MITgcm" written by Simon Ohlsen.

3.1 Concept

In contrast to many approaches to dataflow analysis, our 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 decided to base the analysis 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 procedure 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 $G_{\mathrm{C}}(N, E, n_0)$ 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 $G_{\mathrm{D}}(N_{\mathrm{D}}, E_{\mathrm{D}})$ with $N_{\mathrm{D}} = N_{\mathrm{C}}$ in the following way (see Listing 3.1): for every node $n_i \in N_{\mathrm{C}}$ of the call graph $G_{\mathrm{C}}(N_{\mathrm{C}}, E_{\mathrm{C}}, n_0^{\mathrm{C}})$, we take the set of all outgoing edges of a node $out(n_i) = \{(n_i, n_j) \in E_{\mathrm{C}} \mid n_j \in N_{\mathrm{C}}\}$. 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.

```
1  Input: G_C(N_C, E_C, n_0)
2  Output: G_D(N_D, E_D)
3
4  N_D = N_C
5  for each n_i ∈ N_C do:
6      for each (n_i, n_j) ∈ out(n_i) do:
7          if data is sent from n_i to n_j:
8              E_D = E_D ∪ {(n_i, n_j)}
9          fi
10         if data is retrieved from n_j by n_i:
11             E_D = E_D ∪ {(n_j, n_i)}
12         fi
13     done
14 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 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 adapted the described concepts to Fortran.

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

```
1  Input: G_T(N_T, E_T, r)
2  Output: G_D(N_D, E_D)
3
4  for each n_i ∈ N_T defining an operation do:
5      N_D = N_D ∪ {n_i}
6      for each successor n_j of n_i do:
7          if n_j involves execution of another operation n_x:
8              N_D = N_D ∪ {n_x}
9              if data is sent to n_x:
10                 E_D = E_D ∪ {(n_i, n_x)}
11             fi
12             if data is retrieved from n_x:
13                 E_D = E_D ∪ {(n_x, n_i)}
14             fi
15         fi
16     done
17 done
```

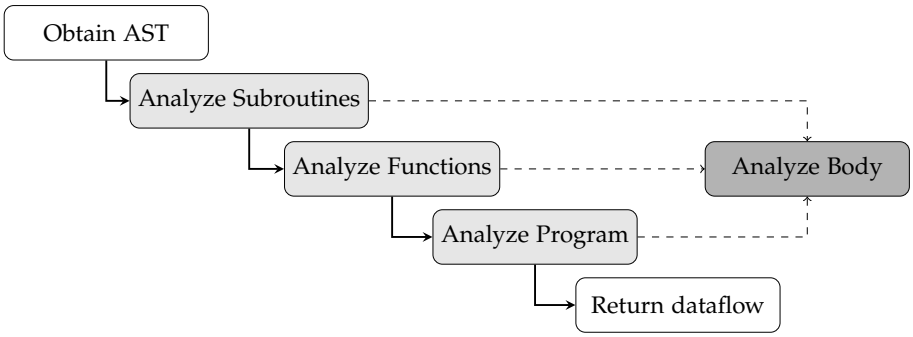


Figure 3.1. Course of the dataflow analysis of a single file. For each `SUBROUTINE`, `FUNCTION`, or `PROGRAM` defined within, the file the dataflow of all statements, e.g. assign statements, in scope of the operation are analyzed.

3.2 Implementation

Every implementation decision was made on a pair-programming bases. Therefore, all details regarding the algorithm are specified in the data flow analysis of MITgcm. To give a brief explanation, see figure 3.1. Each Fortran file is split into a different analysis parts. That means, identifying every subroutine as well as function declaration. For each of them,

a body analysis is performed containing a high amount of checks to identify data flow. Due to Fortran characteristics, every file may contain a main `PROGRAM` part. It defines a script part, not accessible by other operations, and therefore only usable from the file itself. Consequently, to catch any data flow initiating an applications simulation the last step of the tool is analyzing an existing main part. After a whole iteration, one file is analyzed and saved in a CSV file. Every following result is added to this file.

Chapter 4

Data Handling and Visualization using OceanDSL Tools

This section covers the second part of implementing a data flow analysis for OceanDSL. First, a detailed explanation of the Kieker architecture meta-model is presented. Second, the implementation is presented in two parts. The first part focusses on the pipeline architecture of TeeTime and the second part on the package structure and class implementation. This chapter ends with an explanation how the created models are visualized and what metrics are used to retrieve valuable evaluation data.

4.1 Concept

In order to achieve an understandable concept of an application, Kieker allows a developer to define structures within an architecture meta-model. This model is designed in different sub model levels, providing different classes and references. The latter allows other models to connect its classes and structure each model for example with a top to bottom approach, making it easy to update and maintain, see figure 4.1. To visualize data flow we adapt the already implemented design of Kieker's analysis model. Each following subsection is describing a specific model level and how it is related to other models.

Type Model

The type model of the Ecore architecture meta-model represents the top referenced level. Its classes store all semantic information. The most interesting classes regarding following implementations are "ComponentType", "OperationType" and "StorageType", alongside the mapping classes.

A ComponentType class stores attributes such as "name", "package", "signature" as well as references to operations, storages and other components. Therefore it is possible to relate OperationType classes with a ComponentType. Moreover, it allows to relate classes of ComponentType with other classes of ComponentType, allowing to group multiple related ComponentType objects. Consequently, any visualization is presented in a more related context and therefore easier to analyze. In order to model a Fortran written ESMs, we did not change any classes nor added any to the Kieker type model. We defined

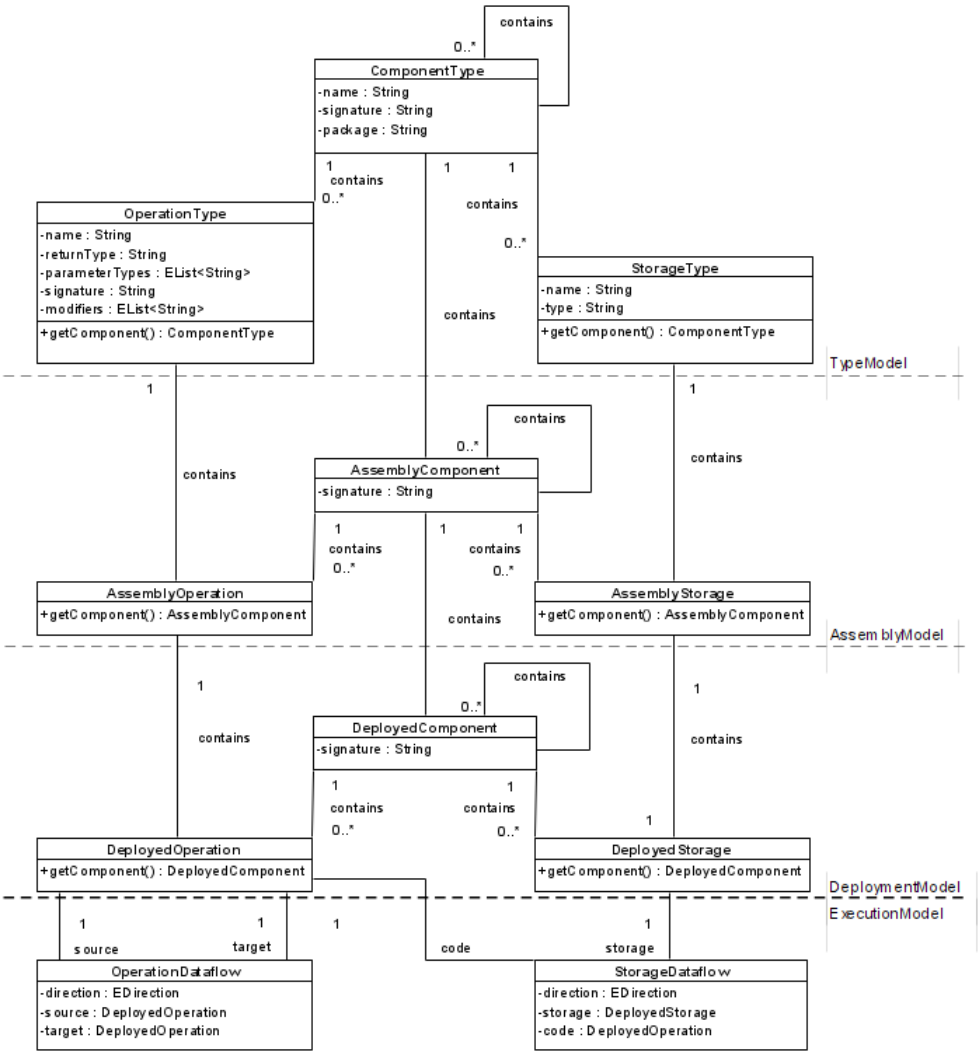


Figure 4.1. Kieker Analysis Model Basic Class in UML

Fortran parts as follows: A ComponentType represents a package and a Fortran file. Therefore, we store every file component in a package component. Next, we declare every `SUBROUTINE` or `FUNCTION` of Fortran as ComponentOperation. A `COMMON BLOCK` is stored as a ComponentStorage. Both are stored in a related file component.

Assembly Model

The assembly model of the Ecore architecture meta-model represents the second level, containing classes of the type model level. In detail, the assembly level stores similar structures of the type model, defined for an assembly model, as you can see comparing layers in figure 4.1. There are classes such as "AssemblyComponent", "AssemblyOperation" and "AssemblyStorage". Again, the component class references operations, storages and other components. Yet, it only stores minimal semantic data for each class because every semantic attribute can be accessed by the referenced ComponentType class, see figure 4.1. Due to the highly connected classes no changes to the Kieker assembly model were made.

Deployment Model

The deployment model represents the next level after the assembly level of the Kieker analysis model. It is usually divided in different DeploymentContexts. In this case, we only handle a single context. There are similar defined classes compared to the assembly model. Classes like "DeployedComponent", "DeployedOperation", and "DeployedStorage" are supported and again, only basic attributes and operations are defined, because of the cross-references to classes of the assembly model, see figure 4.1. We used the origin definition of the Kieker deployment model and mapped all Fortran packages, components, operations and storages as described above.

Execution Model

The execution model represents the last and most interesting level of the Kieker analysis model. The model defines all data flow interaction between operations and storages. Compared to other levels, we added and reworked classes of the original Kieker model. Next to the required mapping classes, we added a class called "OperationDataflow". It contains one attribute, "direction" of type "EDirection" and two references, "source" and "target" of type DeployedOperation. The EDirection is a Java enumeration type, declaring data flow interaction, such as *read*, *write* and bidirectional data flow. Therefore, an OperationDataflow class declares which operation accesses which operation and with what kind of intention. According to its type for attribute "source and "target, a OperationDataflow object is related to the deployment model, see figure 4.1.

Kieker's execution model already defines a storage data flow class, called "AggregatedStorageAccess". It contains one attribute, "direction" of type "EDirection" and two references, "code" of type "DeployedOperation" and "storage" of type "DeployedStorage". To make the execution more uniform, we refactored the class to "StorageDataflow", but still containing the same attribute and references.

We are now able generate Java code using the EMF to create a fully qualified meta-model of an Fortran ESM. Semantics defined on type level down to actual data flow

connections on execution level create a concept for storing and modeling most of Fortrans characteristics.

4.2 Implementation

The goal of the last section is to explain design choices according to TeeTime's characteristics and advantages. Also, it says how the theoretical model design above is related to practical implementation and what is achieved after a successful run.

Pipeline Design

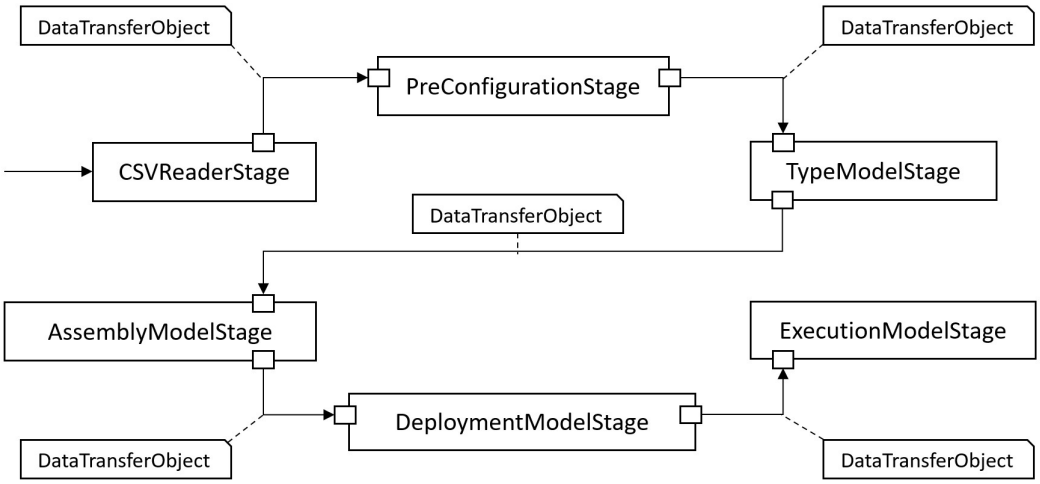


Figure 4.2. TeeTime Stage Concept

Being a pipe and filter architecture framework, TeeTime allows to create meta-models using stream processing. A first draft of recovering architecture based on data flow analysis data was already implemented. The pipeline was designed as follows. As a starting point, a TeeTimeConfiguration class configures a pipeline of six stages and creates, two lookup classes containing additional meta-data of the ESM structure, e.g. which operation is stored in which component and which component is related to which package. All stages are linked by "DataAccess" type pipes, and placed in a non parallel way. The idea is to read in a CSV file containing parts of data flow information in a single line and convert all valuable data into an object, which will be passed through the whole pipeline. Each line of the CSV file triggers another pipeline run and is handled differently depending on the current stage. After the first stage read and created the passing object,

the TypeModelStage retrieves all semantic data to store it in a type model. Afterwards, the AssemblyModelStage links the created type model classes and stores them in an assembly model. The DeploymentModelStage works the same way, storing all information in a deployment model. Only the ExecutionModelStage differs from the other and creates classes representing parts of data flow information. As a result, every stage is encapsulated with the name referenced model, making changes while developing become less difficult. The above-mentioned cross-referencing of models connects all stages in a way, that allows for instance debugging a type level problem on execution level.

Our approach of in reading and storing data into meta-models is very similar. It only varies in the detail of information that are passed on and in the concept of creating model classes. By looking at figure 4.2 it is possible to identify stages, which were mentioned before. Then again, we want to display data flow based architecture using new generated inputs. Therefore, we insert a new stage, called PreConfigurationStage, to the pipeline receiving from the reader stage and passing objects to the TypeModelStage. Its purpose is to configure the passing object, called DataTransferObject (DTO). Not only it filters eventually caught false classified data flow, it separates between a storage access and an operation access. Consequently, the upcoming stages can easier create classes for meta-models. The rest of the design is adapted from the existing design of *SAR* but reimplemented in its detail. But still, a TypeModelStage creates components containing storage or operation objects, an AssemblyModelStage links type model objects, a DeploymentModelStage links an assembly model objects and the execution model stores the final data flow connections using deployment classes. All information is read from a DTO. However, only the first two stages write to a DTO. When the reader stage is finished from reading the input file and the pipelines handles the last passing data, the tool is stopped by the outer TeeTime configuration. A shutdown process writes all created models to a given directory.

Extending SAR tool

The implementation of the concept above is placed next to the *SAR* default package in the respective GitHub repository. It is divided in a "model" and "stages" packages, along-side a configuration class for the TeeTime pipeline.

The Package "model" contains three classes storing information for pipeline stages. First, the DataTransferObject class, a plain old Java object, implements attributes describing possible data flow. Its only methods are getter and setter functions, to allow external read and write access to a DTO object. Attributes are identifiers for the source operation and the related component and package. Also target identifiers for operation or storage, component and package are stored. Next to the direction of data flow, represented by an enumeration type containing *read*, *write* and *both*, flags can be stored to identify a data flow to a storage or operation. Second, the ComponentLookup class implements a way to lookup existing components and its content, meaning operations such as `SUBROUTINES`s and `FUNCTION`s. It is highly connected to the ComponentStore class, the last one of the model package. The lookup class stores a map matching component identifier to a ComponentStoreObject, while

a store object holds all collected information for a specific component. A lookup object provides getter and setter to edit a store object and methods to a different characteristic of a data flow input. For instance, it is possible to check the origin of a specific operation or whether another operation or a storage is accessed.

Next, the package "stages" contains all pipeline classes. Beginning with the CSVBsc-DataflowReaderStage, a simple CSV reader is implemented, taking four-value input. If a line is successfully validated, the first stage initializes a DTO object and sends it to its output port. Lines containing more or less than four values are skipped to prevent any fault occurrence in the following pipeline steps. After finishing the input file, the stage triggers a "workCompleted" method by the TeeTimeFramework to initiated shutdown service. The second stage in the pipeline is a PreConfigurationStage. Its main task is to separate between valid, invalid and so-called unknown data flow using the lookup classes provided by the TeeTimeConfiguration class. A validity check has one requirement: is the current target identifier referenced in a listed component? Additionally the check is split into two questions:

1. is the target object a `COMMON BLOCK` and therefore a storage?

2. is the target object an operation?

If no component contains the target, the DTO is marked as *unknown*. However, is the target listed in a component and the received DTO object is marked as either operation or storage data flow and after setting the related target attributes it is passed on to the first model stage.

Moving on, the next three stages work very similar but on different model levels. Talking about TypeModelStage, AssemblyModelStage and DeploymentModelStage each stage creates a complete model of the level the name it is referenced to. The process is structured as shown in figure 4.5. First, a component object including an operation object is depending on recent stage actions, initialized to or retrieved from the current model, saving source attributes of a DTO. Additionally, a related package component is initialized or retrieved containing the just created component. Second, whether the current data flow includes another operation or storage, further process is divided. Is the DTO representing the former case, a method is called to create and save a component to or get it from the current model and add the target operation. Therefore, the DTO is cloned and edited to reuse already implemented code. Last, a package component is created or retrieved to classify the target component's origin. In case of the latter data flow a component is created to store storage objects. Despite the other components holding operations, the component holding storages has no related file or package. Due to the Fortran characteristics, all `COMMON BLOCK`s are referenced where used, making them impossible to bind to specific files. The so called "COMMON-Component" therefore stores all storages independent of their original operation usage. Again, the component is only created if no existing component was stored in the current model. By the end of the process and passing all three stages, the

type model, assembly model and deployment model contains all parts the DTO includes with the correct cross-referencing between model levels.

The final step of the pipeline is the ExecutionModelStage. Here, all DeployedComponents and -Storages are connected according to the passing DTOs. Likewise before, any class initialization depends on the passing data flow type. In case of storage data flow a StorageDataflow object holding a DeployedStorage and DeployedOperation, is added to the execution model. When an operation accesses a specific storage multiple times, the already initialized data flow object is edited by the value of direction. It is changed to direction *both* when the stored data flow object represents a *read* and the current DTO represents a *write* access. Additionally, it is changed in the switched case. An OperationDataflow object, holding two DeployedOperations, is added to the execution model when the current DTO is marked as operation data flow. Its only requirement is both components origins are known. Otherwise, the initialization is skipped. On the contrary to StorageDataflow, an OperationDataflow object is not reedited when an operation is accessed multiple times. Due to Fortran characteristics and operation definitions, based on our data flow concept in Chapter 3, it is not possible to access an operation with *read* but later on with a *write* instead. Consider the following example: Operation `OpX` calls Operation `OpY`. The first appearance is declared as *read* data flow. That means according to our data flow implementation `OpY` is a function having no function arguments. To specify a data flow connection from `OpX` to `OpY` with *write*, `OpX` discards a return value of `OpY` and provides at least on argument. This contradicts the scenario of *read* and consequently proves our implementation strategy as valid and keeps the created meta-model right. As the last pipeline stage, the ExecutionModelStage only listens to the input port, sending no objects to any output port.

Every modeling stage is designed in layers, see figure 4.5. The first layer contains an inherited execution method, which coordinates the main action of each stage. Except for the execution model stage, every second layer is the setup layer. It is called by the first layer to initialize components to store operations and storages. Next, an adding layer is defined to provide methods to fill in package and file components. It is accessed by the setup layer and the execution layer. Beneath the adding layer, a creation layer is implemented. It is called by the adding layer to create first appearances of components, operations and storages. In case of any helping functions, an additional supporting layer is provided. The chosen implementation design allows easier maintaining and updating code to different requirement changes.

To run the bachelor implementation of *SAR*, additional argument matcher are provided in the Settings.java file. To avoid runtime errors, three flags should be set. The first flag (*-bsc-j*) sets the location of a data flow CSV file containing the essential data for every pipeline run. Then, the second flag (*-bsc-c*) contains the location of a file content CSV. Along with the third flag (*-bsc-p*), a location of a CSV file holding package information, the DTO is fully edited to start any model creation. By using *-bsc-j* the main class of the *SAR* tool is called, initializing the presented TeeTime configuration for data flow architecture recovery.

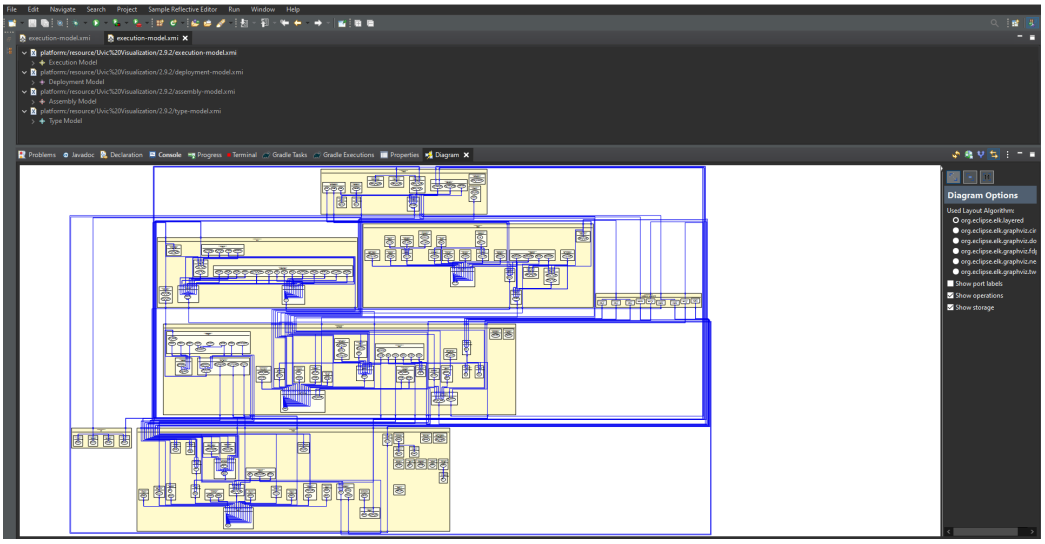


Figure 4.3. Eclipse view of Model Visualization

4.3 Visualization and Metrics

The last step of our analysis is the metric generation and visualization of created XMI meta-models. Therefore, we made use of two tools. To calculate multiple metrics we adapted the *MVIS* tool of the OceanDSL tool collection. The main goal of *MVIS* is to create a graphic file and compute metrics based on it. We decided to only implement the metric generation considering our changes in Kieker's meta-model and model complexity. By running *MVIS* on provided XMI files, we calculate incoming and outgoing edges of operations and modules. Also, we are able to use metrics defined by [Allen 2002]. Regarding model graph theory [Allen 2002] calculates a model-size as well as model-complexity. Both are used to compare changes to a complete ESM architecture in the following Chapter 5.

As mentioned, model complexity was a critical criteria in case of visualization. Comparing a generated graph of *MVIS* with a visualization of Kieler, the latter makes analyzing easier regarding identifying any relationships due to the interactivity of the Eclipse plugin. That is why we collaborate with our supervisor to implement the new changes into Kieler. The final view is generated in Eclipse by the Kieler Plugin, as shown in figure 4.3 and can be exported to PNG or SVG files.

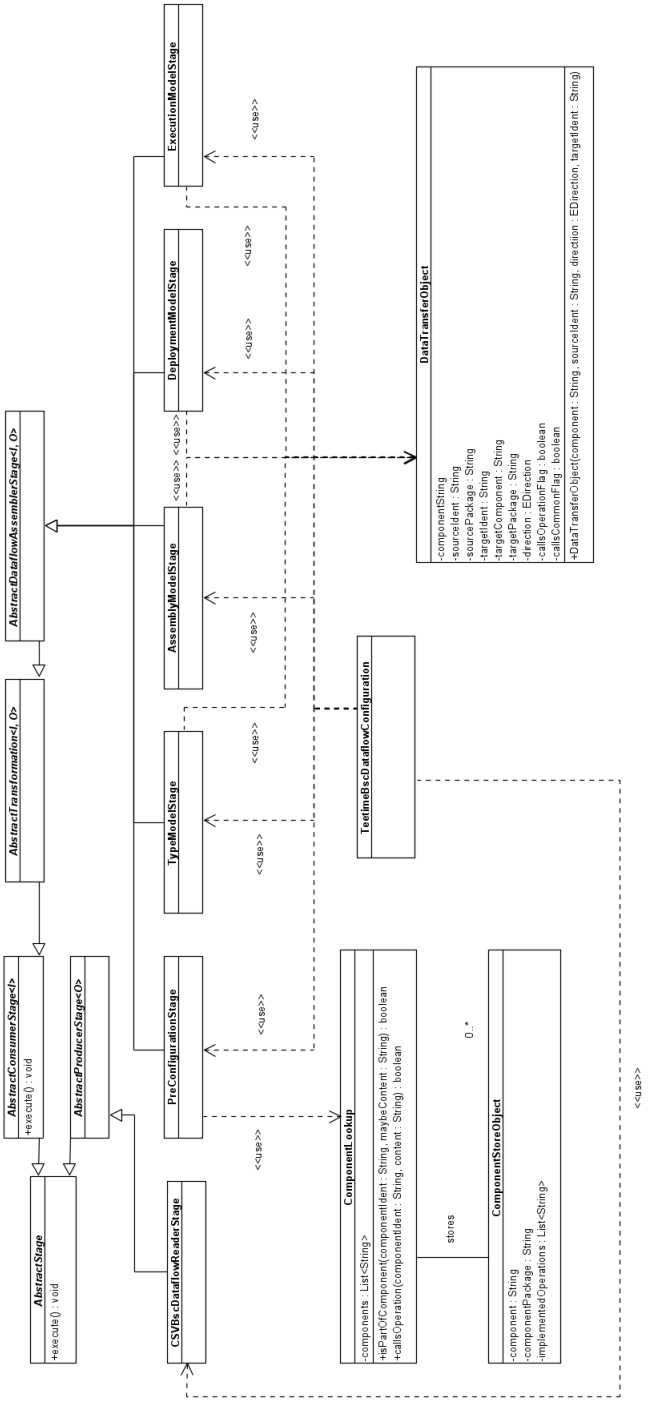


Figure 4.4. TeeTime Implementation presented as Class Diagram

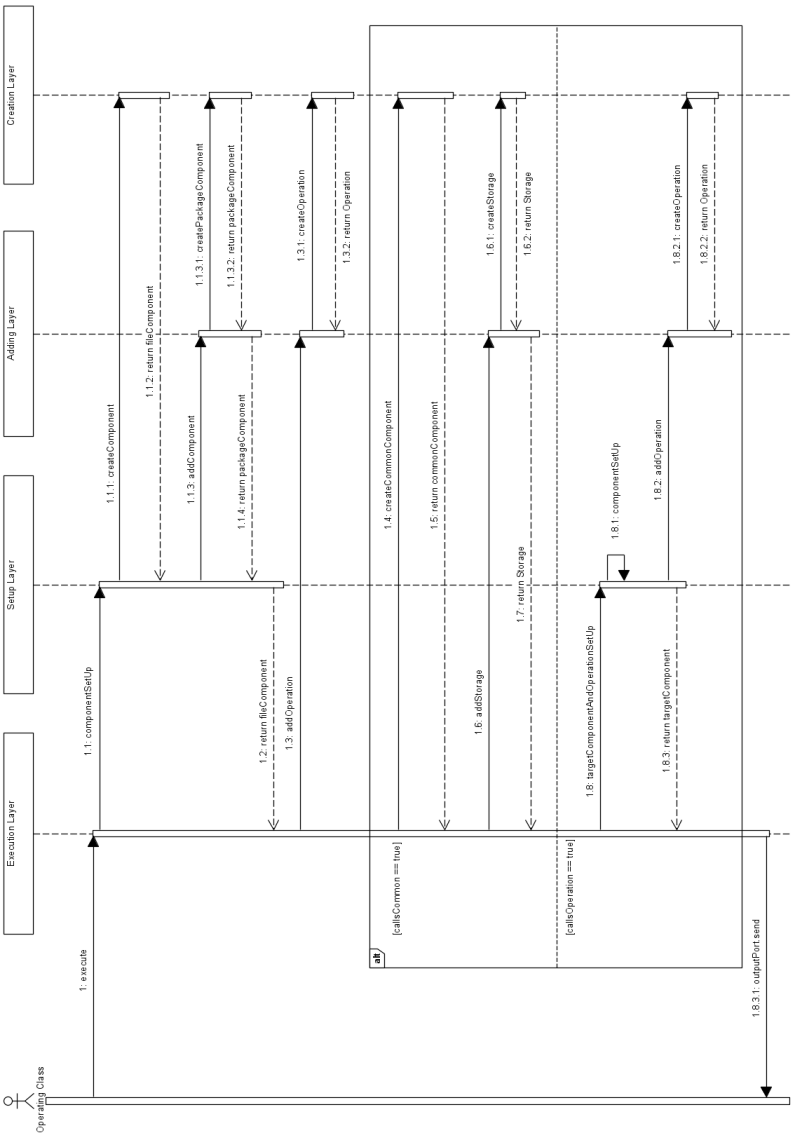


Figure 4.5. Sequence diagram modeling process of TypeModelStage, AssemblyModelStage and DeploymentModelStage. Method names can differ slightly to actual implementation due to level differences

Chapter 5

UVic Earth System Model Evaluation using ESM Data flow Tool

This chapter contains evaluation work on different versions of UVic ESCM, using visualization in Eclipse by Kieler Plugin. All versions are evaluated and compared to other versions to characteristic updates and improvements over time. First, the setup is presented following the analysis of the architecture reconstruction. Second, each version of the extending UVic part which was analyzed is presented with different visualizations. Every data is based on the execution model generated by the OceanDSL tool *SAR*.

5.1 Architecture Reconstruction

The experiment is structured in three phases: Preprocessing, CSV generation and finishing with meta-model creation. Every step is scripted by bash files in our ESM data flow repository and coordinated by a self-hosted jenkins server. The idea is to provide maximum amount of information in a single line of a CSV file, to recreate every component, every operation and every storage used. In the beginning, the UVic ESCM repository[1] is cloned to check out the current version of interest. Next, the preprocess script generates analyzable Fortran code and stores it into a separate directory to keep every file in its original state. The directory is passed to the main Python script, initiating data flow output generation. By the time all files are written, a provided binary of the *SAR* tool extension is called to create XMI files, containing the data flow visualization. Input parameters are the output directory alongside the content, map and data flow file. To finalize the experiment, a *MVIS* binary is called to calculate metrics of the created meta-model. For UVic ESCM, all these steps were executed using a self defined Python script, starting a whole analysis run of UVic ESCM version "2.8", "2.9", "2.9.1" and "2.9.2" and downloading a zip file containing all generated output files. As mentioned in Chapter 4 all XMI models are imported to the Eclipse IDE and presented by the Kieler plugin. The software used and results that were generated are published here [Ohlsen and Illmann 2022]

[1] https://git.se.informatik.uni-kiel.de/oceandsl/uvic-reconstruction

Table 5.1. Table presenting sum of incoming and outgoing edges of components in packages of UVic ESCM version 2.8.

Package	Edges (Outgoing)	Edges (Incoming)
common	46	66
embm	14	21
ice	5	0
mom	55	34
mtlm	32	28

Table 5.2. Table presenting sum of incoming and outgoing edges of components in packages of UVic ESCM versions 2.9/2.9.1.

Package	Edges (Outgoing)	Edges (Incoming)
common	63	83
embm	13	24
ice	5	0
mom	53	28
mtlm	32	30
sed	26	27

5.2 Architecture Evaluation

All versions of UVic ESCM are analyzed using the metrics provided by *MVIS*. Mainly, complexity and size are evaluated next to the most interesting data flow designs. Every value given is rounded up or down for clarity reasons. By comparing the version history, changes and the resulting impacts are stated. Components and operations are described by incoming and outgoing edges. Based on our metric calculation and visualization plugin, there is no matching relation to *write* and *read*. For example, a component `A` is connected with component `B` with an outgoing edge of `A` and an incoming edge on `B`. There are now two possibilities to read: On the one hand, `A` as caller provides data for `B` as callee. On the other hand, `B` as caller reads data from `A` as callee. Metrics for edges of package, components and operations are presented in the following for each version. A detailed comparison is discussed in Chapter 6.
In advance, every analysis run of each version, failed to analyze a single component in package "netcdf" due to parsing reasons. Because the failing file is the only file defined in "netcdf", the package is left out in the following architecture analysis.

UVic ESCM 2.8

Version 2.8 is the earliest version analyzed in this evaluation. Its architecture is created containing an amount of 88 component files and 133 operations. All of them are connected

Table 5.3. Table presenting sum of incoming and sum of outgoing edges of components in packages in UVic ESCM version 2.9.2.

Package	Edges (Outgoing)	Edges (Incoming)
common	62	81
embm	13	24
ice	5	0
mom	53	28
mtlm	32	30
sed	26	27

by 171 data flow relationships. The component file having the maximum number of incoming data flow is "tmngr.f", a component of package "common". The maximum number of incoming edges in terms of operations is the operation "rdstmp" of component "tmngr.f" with nine inner model connections. On the contrary, both maximum values of outgoing connections are related with package "mom". Component "mom.f" has an amount of 15 outgoing edges, all related to its defining operation "mom".
The visualization shows the following characteristics points. Operation "mom" only transfers data inside its package, being a data flow bottleneck for most of "mom.f"s' components, see figure 5.1. In comparison in figure 5.2, the "main" operation of UVic ESCM only connects with five other operations. Additionally, package "common" includes six components having no data flow connection, see figure 5.3.
Version 2.8 is split into five packages containing inner model data flow, see table 5.1. Its source code defines two other packages "ism" and "rot" as well, but both are not utilized in the used scenario. Including the most incoming and outgoing edges, "common" is the largest package in terms of data flow. Compared to "common", having more outgoing edges than incoming, "mom" is the next largest package, having more incoming than outgoing edges. Package "ice" is, beside empty packages, the smallest data flow package, containing only five incoming edges.

UVic ESCM 2.9 / 2.9.1

Analyzing version 2.9 and 2.9.1, both represent a very similar results, regarding visualization and metrics. Therefore, this section presents 2.9 and 2.9.1 as 2.9/2.9.1
Containing 94 components and 156 operations, 2.9/2.9.1 has 202 edges stored. Similar to version 2.8, "tmngr.f" has the maximum number of incoming edges with an increased value of 27. Again, "rdstmp" in component "mom.f" represents the operation with the highest incoming connections with a value of 14. However, the outgoing maximum varies. Component "UVic_ESCM.f" has 18 outgoing edges, where its defined operation "main" represents 16 of it, having the highest outgoing operation edge-count .
The following characteristics are specifically noteworthy with respect to visualization. In complex packages, one component is a data distributor. For example, in figure 5.1, the

Table 5.4. Table presenting size and complexity of different model versions of UVic ESCM.

Version	Complexity	Size	Ratio	Lines of Code
2.8	161.049	1454.177	0.111	33134
2.9	244.871	1677.896	0.146	40130
2.9.1	244.871	1677.316	0.146	40170
2.9.2	250.142	1654.499	0.151	40176

shown operation connects multiple inner package operations of package "mtlm". Compared to version 2.8 the "main" operation of UVic ESCM in 2.9/2.9.1 is three times higher connected with other models operations. Additionally, new components are added, having no data flow relationships with other model operation, see figure 5.3. By updating UVic ESCM to 2.9/2.9.1 a new package "sed" is added next to edges and components, see table 5.2. It has 26 incoming and 27 outgoing edges. "common" and "mom" represent the most complex package.

UVic ESCM 2.9.2

The latest version analyzed 2.9.2, containing the same amount of components and operation in contrast to 2.9. The main and only difference is the reduction of edges by 3, regarding storage connections. Comparing table 5.3 and 5.2 only package "common" differs by one incoming and two outgoing edges less. The operation metric proves the relation to UVics' storage component. Therefore, no major differences or distinct features are found.

Applying Allen Metric to UVic ESCM

Table 5.4 shows all results regarding the complexity and size of the models. It describes a rising of complexity and lines of code over time. The latter influences the size of the UVic ESCM, resulting in different complexity changes. For example, 2.9 and 2.9.1 40 lines of code were added, without influencing any complexity raise. However, by adding 6 lines of code 2.9.1 and 2.9.2 separates a complexity of about 6.

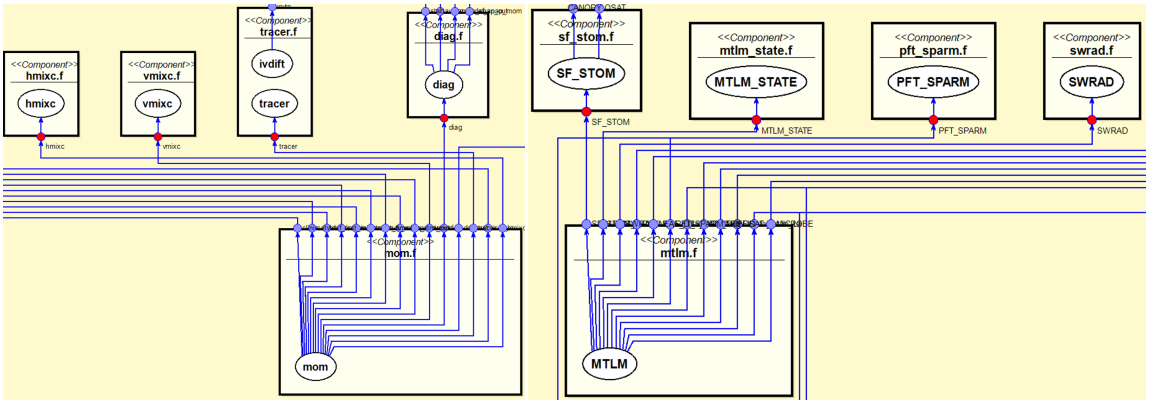


Figure 5.1. Excerpt of the visualization in Eclipse by Kieler plugin of operation "mom" (left) and operation "MTLM" (right) from UVic ESCM version 2.8 (left) and version 2.9 (right)

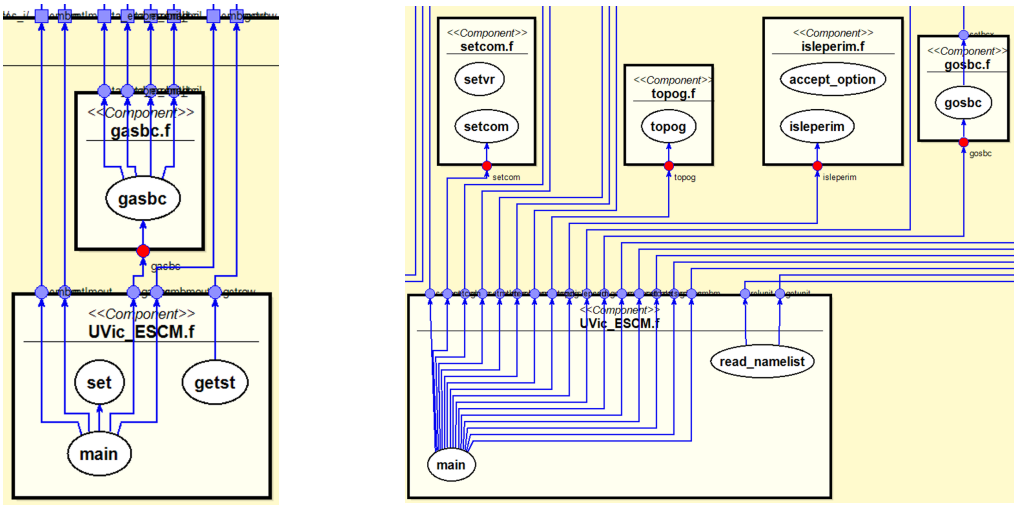


Figure 5.2. Excerpt of the visualization in Eclipse by Kieler plugin of operation "main", UVic ESCM version 2.8 (left), UVic ESCM version 2.9 (right)

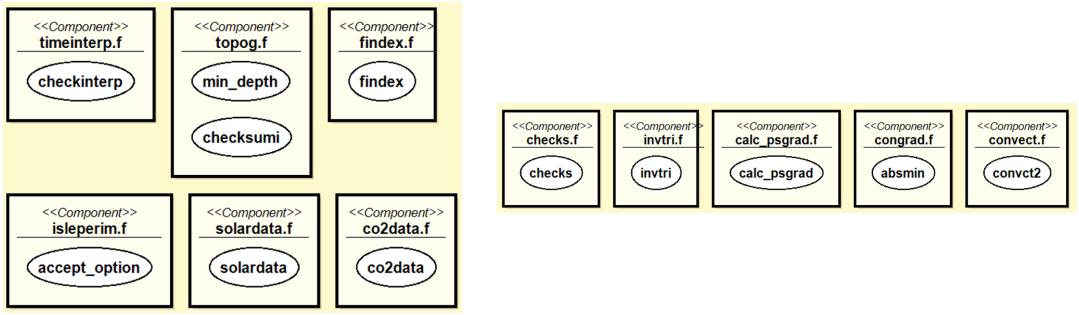


Figure 5.3. Excerpt of the visualization in Eclipse by Kieler plugin of operation having no data flow connections UVic ESCM Version 2.8 (left), UVic ESCM Version 2.9 (right)

Chapter 6

Discussion

This chapter discuss our approaches, decision results. We explain why a design choice was made and what impact it caused. In the beginning, a discussion of the results is given, to grade the tool implementation. Second, the analysis itself is discussed in context of design, performance and trustability.

6.1 Analysis Results

The results presented in Chapter 5 prove statistically data flow complexity changes in different dimensions through the applications evolution. Depending on the release of a new version, more or less changes can be evaluated. Looking at figure 6.1, the model size increases after 2 releases, as well as data flow complexity. By an amount of 7000 more lines of code, version 2.9 has a higher data flow complexity of 80 than 2.8, indicating package refactoring and implementing new features and model content. In contrast, the evolution of 2.9 indicates minor updates, regarding possible bug fixes, and storage access. The update from 2.9 to 2.9.1 reduces just the model size by 0.53, changing a total of 40 lines of code, but the computed complexity value is identical. However, the update from 2.9.1 to 2.9.2 increases the complexity by a value of six and reduces the model size to about 23, by only adding 6 lines of code, see table 5.4 and figure 6.2. Consequently, changes from 2.9 to 2.9.1 include no data flow connections based on complexity results, but changes of algorithm strategies used in an operation or components. Additionally, any relation between statistics of lines of code and data flow complexity can be excluded, because complexity rises by only adding a minimum amount of code lines. However, the added code lines in the first update have no effect on UVic ESCMs complexity.
Due to the visualization strategy used, unknown data flow is not presented in any visualization, but is still stored in all generated XMI files. Hence, model size changes are not shown in the visualization, but included in the model size calculation. For example, the reduction in size from 2.9 to 2.9.2 is related to unknown classified data flow. Therefore, the actual value varies from the computed value, because a moderate size ".unknown" component, stored in the model, increases the model size.
Seeing figure 6.2, the complexity ratio is significantly low, compared to a UVic ESCM call graph analysis. Earlier experiments, made by the Software Engineering group, resulted in an average call graph complexity ratio of 1. However, the computed data flow complexity

reaches only $\frac{1}{13}$ of the maximum. This indicates a highly connected application but a minimal amount of data flow between operations. The model generation is designed to provide an optimal view to its architecture, ignoring all redundant edge. Therefore, the data flow interaction between two operations can be stronger than assessed, but the ratio value is influenced to a minimum.
All in all, metrics of updates after version 2.9 have stable behavior, indicating small but significant fixes or feature improvements.

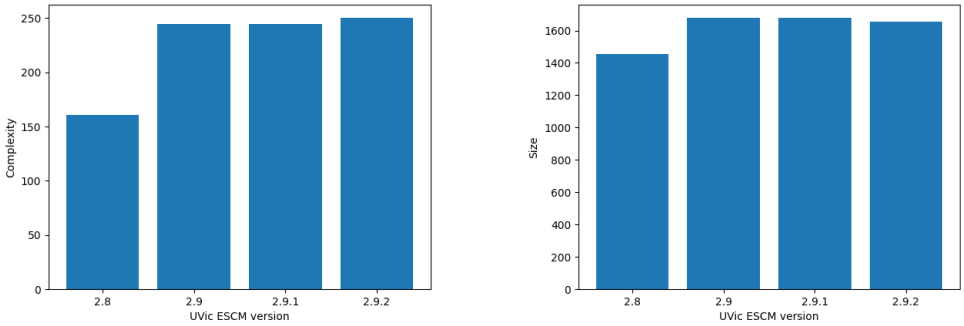


Figure 6.1. Model complexity (left) and model size (right) based on table 5.4 from UVic ESCM version 2.8 to 2.9.2

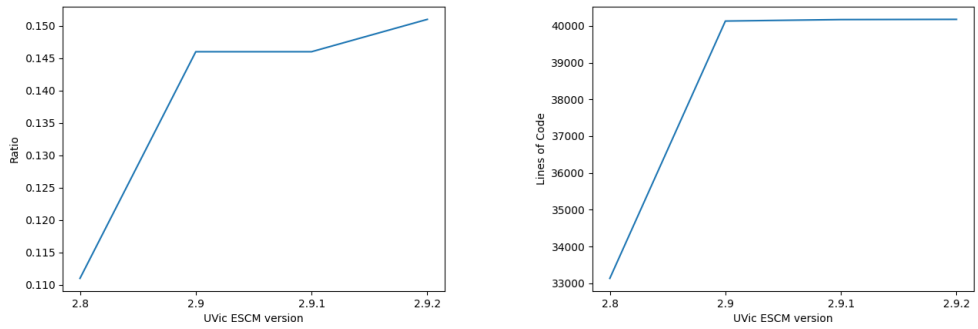


Figure 6.2. Ratio of Model complexity (left) and lines of code (right) based on table 5.4 from UVic ESCM version 2.8 to 2.9.2

Regarding the visualization, two things stand out. First, independently from a specific version, every package contains one or two data flow key components, meaning a distributor of data inside the package. Examples are "UVic_ESCM.f" in package "common" and

components named similar to the package and containing operation, see figure 5.1. Second, package "common" and "mom" contain components having no inner model connection, see e.g. operation "mom" and "MTLM" in figure 5.3. This is related to unknown data flow classification. As mentioned in Chapter 4 the execution model is filtered by data flow connected to other model components. Therefore, any outer relationship are discarded and not visualized.

6.2 Limits of the Implementation

The implementation and evaluation process revealed issues with the current implementation of the data flow analysis. The first set of issues is related to *fparser*. A problem we encountered are parsing issues with some file for which found no solution. 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. This issue could not be clearly identified, due to improper error massages, nor could we solve the problem by, e.g., removing comments before parsing. Additionally, *fparser* has issues to differentiate between statement functions and arrays within the code in some particular occasions, which leads to arrays being declared as functions. We can circumvent this issue by tracking array declaration, but this adds to the complexity to the data flow analysis. Possible, 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 data flow analysis procedure are mainly related to the tight adaptation of the matching structures 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 include and declaration. 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 data flow 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` 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 data flow 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.

[1] `https://github.com/pmarguinaud/fxtran`

[2] `https://github.com/OpenFortranProject/open-fortran-parser`

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

```
1  SUBROUTINE SUB(A, B)
2  IMPLICIT NONE
3  INTEGER :: A
4  INTEGER :: B
5
6  B = A + 1
7  END
8
9  PROGRAM RUN()
10 IMPLICIT NONE
11 INTEGER :: A
12 INTEGER :: B
13
14 A = 1
15 CALL SUB(A, B)
16 END
```

For the architecture reconstruction, we also identified points of improvement, besides evaluating different strategies of 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 data flow related operations, this edges can be annotated with the respective cardinality determined by the data flow analysis. Further, there is currently no way to see which intention the data flow has. For example, it is unclear if an edge from `C1` to `C2` is a *write* from `C1` in `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 data flow.

The implementation of *SAR* is designed to validate incoming data. Therefore, not only structures in the CSV file need to be respected, but also a second check is required to classify wether a data flow interaction is an inner model connection or possible false classified data flow. Consequently the visualization represent correct inner model data flow and provided trustability. However, the current implementation collects all false classified data flow and create an unknown component, which influences the calculated model size metric. Instead of an extra component, logging false classified data flow is advisable.

Lastly, the chosen concurrency pattern improved the analysis performance of the data flow analysis on an AST compared to earlier implementations. Using the master worker pattern to separate the workload on four threads, made parallelism possible. The former approach of the *Software Engineering* group used linear analysis implementation to extract data from Fortran source code, leading to higher operation time depending on the current model size. Our implementation performs a complete analysis, containing the whole

experiment explained in Chapter 5 for four UVic ESCM models, in 1:43 minutes. The ESM evaluated by my partner Simon Ohlsen contains a major number of variants of the ESM. A whole analysis run, including all variants is achieved in 3:06 hours, even though parallelism is enabled.

Chapter 7

Related Work

In this chapter some related work of the keywords this thesis is based on, are presented. It is split into two main parts: data flow analysis and pipe and filter architectures.

7.1 Dataflow Analysis

[Allen and Cocke 1976] published a paper, presenting methods to determine data flow relationships in a given program. It focuses on data flow in a graph theory approach. In detail, a control flow graph representation is analyzed using different methods to answer three questions: "What data definitions reach each node in the graph and can therefore affect uses in the block represented by the node, what uses of data items are upwards exposed from each node and its successors and hence can be affected by definitions of the items, what definitions are ”live” on each edge of the graph" [Allen and Cocke 1976, p.146]. The main concept is based on different characteristics such as associating information with edges not nodes, introducing intervals and (ir)reducible graphs. In the end, a full analysis procedure is described.
[Lee and Ryder 1992] worked on a self developed framework to "exploit parallelism inherent in the solution process of data flow"[Lee and Ryder 1992, p. 236]. Their purpose is to improve execution time of a machine addressing multiple data flow problems. To prove a successful implementation, empirical statistic are discussed and used to motivate further research. In detail, the authors declare two types of problems: intraprocedural containing reaching definitions and available expressions and interprocedural side effect problems such as maybe modified, maybe reference and must be killed. On basis of these problems three parallelisms are classified, independent-problem, separate-unit and algorithmic parallelism. Using intra- and interprocedural definitions, every type of parallelism is broken down to a task dependence graph except for algorithmic problems describing solutions "to refine tasks for solving [...] problems on large procedures"[Lee and Ryder 1992, p. 240] All three shall be detected by their unified framework. By constructing and analyzing a tailored problem dependence graph, exploits for independent-problem and separate-unit parallelism are exposed. Further algorithmic exploits are revealed based on control flow graph characteristics and the chosen "algorithm to perform the intraprocedural analysis"[Lee and Ryder 1992, p. 241]. To study the performance of their implementation, they used more than 80 Fortran procedures and performed a parallel hybrid algorithm on

different processors setups. Without mentioning any numbers, the authors are encouraged to work on a first prototype of a parallel data flow analyzer and continue their research.

7.2 Pipe and Filter - Tools and Architectures

Papers presenting TeeTime include references for pipe and filter architectures and other tools performing similar work. For example, [Wulf et al. 2014] states the frameworks architecture is based on a Tee and Join approach, giving the presented framework its name. [Maschotta et al. 2001] uses the Tee and Join pattern to address different requirements in the field of signal processing as well. Tee- and Join- Pipeline characteristics are multiple input/output channels on each filter, allowing switches and loops. Its only requirement is to define specify data types for each port to avoid runtime errors and therefore system crashes. [Maschotta et al. 2001] describes advantages such as "rapid prototyping of pipeline" [Maschotta et al. 2001, p. 3] and flexibility and reusability of pipeline elements. By mapping single filter processing algorithms as filters and implementing more functionality for their purpose, the authors present a UML architecture design to handle problems such as fast processing and the ability of real-time processing as signal software demands and extensibility of components and reusability as modern object-oriented software requirements [Maschotta et al. 2001, p. 3].
[Wulf et al. 2014] references FastFlow, a tool using pipe and filter processing. In 2011 presented by [Aldinucci et al. 2011], the FastFlow framework is published "as a stack of C++ template libraries" "targeting cache-coherent shared-memory multi-cores"[Aldinucci et al. 2011, p.1]. Its main principles are a layered design next to stream parallelism. By a layered design, extend low-level programming elements by high-level programming elements based on the same core run-time support. High-level programming include different skeletons of pipelines or so-called farms to allow a developer to have an easy and manageable implementation effort, while the second layer provides different producer-consumer relations implemented by concepts of the third layer. Likewise, TeeTime [Aldinucci et al. 2011] introduces a possibility to build sequential and parallel stages to improve a pipelines' workflow. Again, a goal of FastFlow is to support simple and maintainable implementations, so they only support stream parallelism in their own library. An example for a real time application using FastFlow is the biosequence alignment SWPS3-FF. It is an implementation brought to the framework to use a Smith-Waterman algorithm determining "biosequence similarities [...] by computing their optimal local alignments."[Aldinucci et al. 2011, p.9]. Future Work includes feature extension according to its main goal and extend the framework by more parallel patterns to use.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This thesis introduced a new tool to analyze data flow of Fortran written ESMs. Parsing Fortran code into an abstract syntax tree structure, provides a way to retrieve data flow information, by passing specific nodes. Furthermore an already existing tool of the OceanDSL project is extended to handle generated data and create a meta-model. Moreover, the earth system model provided by the University of Victoria (UVic ESCM) is evaluated as well as the implementation strategy. As a result, four architecture visualizations of UVic ESCM versions 2.8, 2.9, 2.9.1 and 2.9.2 have been generated. The main changes are made by the update from 2.8 to 2.9, while the evolution of 2.9 to 2.9.2 include only minor modifications. Calculated metrics indicate a rise of complexity and lines of code. Despite the increasing data flow complexity, the size decreases between the versions 2.9.1 and 2.9.2 indicating implementation improvements. These are additional matching of the data flow characteristics like call by value or call by reference and cardinality of each connection. Moreover, a restructuring by replacing the parsing dependency is an advisable step.

8.2 Future Work

First, the inaccuracy of the fparser tool influences the data handling process and provided result. It would be advisable to discuss alternatives, to compare our achieved results with other possible outcomes. Additionally an algorithm alternative of mismatching function and arrays can be developed. In context of detail matching, another idea is to include call by references and call by value operations along with an argument list, and cardinality count in the visualization. Consequently, a more detailed analysis based on a meta-model and metric evaluation can be achieved.
Second, matching more Fortran characteristics such as `BLOCK DATA` etc., and test the tool compatibility with other earth system models is crucial to create a more independent analysis tool. Lastly, this thesis was written in the context of OceanDSL. Therefore, the analysis tool containing Python and Java code can be merged into the projects' tool collection. Moreover, the already existing call graph analysis of ESMs can be combined with our data flow analysis concept, to generate a fully functional static analysis tool.

Bibliography

[Aldinucci et al. 2011] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: high-level and efficient streaming on multi-core (a fastflow short tutorial). *Programming multi-core and many-core computing systems, parallel and distributed computing* (2011). (Cited on page 38)

[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 22)

[Allen and Cocke 1976] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM* 19.3 (1976), page 137. (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)

[Buschmann et al. 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture, volume 1, a system of patterns*. Volume 1. John Wiley & Sons New York, 1996. (Cited on page 8)

[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)

[*Eclipse Modeling Framework (EMF)*]. *Eclipse modeling framework (emf)*. https://www.eclipse.org/modeling/emf/. Accessed: 2022-09-26. (Cited on page 9)

[FORTRAN 77]. *FORTRAN 77 Language Reference*. Oracle Corporation, 2010. URL: https://docs.oracle.com/cd/E19957-01/805-4939/index.html. (Cited on page 6)

[*Fortran High-performance parallel programming language*]. *Fortran high-performance parallel programming language*. https://fortran-lang.org/. Accessed: 2022-09-26. (Cited on page 2)

[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 5, 6, 11)

[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 9)

[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 6)

[Lee and Ryder 1992] Y.-f. Lee and B. G. Ryder. A comprehensive approach to parallel data flow analysis. In: *Proceedings of the 6th International Conference on Supercomputing*. 1992, pages 236–247. (Cited on page 37)

[Maschotta et al. 2001] R. Maschotta, S. Boymann, S. Lehmann, and D. Steuer. Software architecture for modular, extensible and reusable signal processing components. *Proceedings of Advances in Automation, Multimedia and Video Systems, and Modern Computer Science* (2001), pages 304–308. (Cited on page 38)

[Merks et al. 2003] E. Merks, R. Eliersick, T. Grose, F. Budinsky, and D. Steinberg. The eclipse modeling framework. *retrieved from, total* (2003), page 37. (Cited on page 9)

[*OceanDSL Domain-Specific Languages for Ocean Modeling and Simulation*]. *OceanDSL domain-specific languages for ocean modeling and simulation*. https://oceandsl.uni-kiel.de/work-packages-2/. Accessed: 2022-09-26. (Cited on page 1)

[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. URL: https://doi.org/10.5281/zenodo.7116165. (Cited on page 25)

[*OMG Meta Object Facility (MOF) Core Specification*]. *Omg meta object facility (mof) core specification*. https://www.omg.org/spec/MOF/2.4.1/PDF. Accessed: 2022-09-26. (Cited on page 8)

[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)

[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 page 6)

[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 9)

[Weaver et al. 2001] A. J. Weaver, M. Eby, and E. C. Wiebe. *The uvic earth system climate model: model description, climatology, and applications to past, present and future climates*. English. UVic. 2001. 68 pages. 12-4-2001. (Cited on page 5)

[Wulf et al. 2014] C. Wulf, N. C. Ehmke, and W. Hasselbring. Toward a generic and concurrency-aware pipes & filters framework (2014). (Cited on pages 8, 38)

[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 pages 8, 9)