Scientific Research Publishing

Journal of Software Engineering and Applications, 2025, 18(3), 113-138
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Software Architecture Evaluation of Earth System Models

Wilhelm Hasselbring, Reiner Jung, Henning Schnoor

Department of Computer Science, Kiel University, Kiel, Germany
Email: hasselbring@email.uni-kiel.de, reiner.jung@email.uni-kiel.de, henning.schnoor@email.uni-kiel.de

How to cite this paper: Hasselbring, W., Jung, R. and Schnoor, H. (2025) Software Architecture Evaluation of Earth System Models. *Journal of Software Engineering and Applications*, **18**, 113-138.
https://doi.org/10.4236/jsea.2025.183008

Received: March 4, 2025
Accepted: March 24, 2025
Published: March 27, 2025

Abstract

Earth System Models (ESMs) play a vital role in understanding and assessing climate change and other earth system-related issues. They are complex and long-living software systems, mainly programmed in Fortran, that undergo changes as science progresses. They have a myriad of variants that must be maintained to support reproducing experiment results. In a research context, often with contributions from scientists on non-tenured contracts and without a formal software engineering education, this can lead to architecture erosion, hampering further development and, therefore, scientific progress. Furthermore, it harms code comprehension, introducing risks for the quality of the earth system models. To address these challenges, our goal is to design and study methods for improving the maintainability of ESMs implemented in Fortran. In this paper, we assess two widely used earth system models—namely UVic and MITgcm—combining dynamic software profiling with static code analysis for reverse engineering. We introduce a new approach to module interface discovery in Fortran systems. We provide a detailed analysis of the ESMs' architectures and report quantitative properties.

Keywords

Earth System Models, Software Architecture, Dynamic Analysis, Static Analysis, Architecture Evaluation

1. Introduction

Earth System Models (ESMs) are complex software systems used to simulate the Earth's climate and understand its effects on, e.g., oceans, agriculture, and habitats. They comprise different sub-models addressing specific aspects of the earth system, such as ocean circulation. Their code is partly decades old. Such scientific models can start as small software systems, which evolve into large, complex mod-

els or are integrated into other models. Software engineering for computational science has specific characteristics [1] and involves dedicated roles such as model developers and research software engineers [2].

The ESMs are continually modified and enhanced to provide new insights for specific research questions. This leads to numerous variants and versions for an ESM, which must be maintained as other scientists intend to base their research on new features and reproduce results. Thus, ESMs are long-living software [3] and face typical risks, e.g., blurring module boundaries and architectural erosion, due to changes in functionality, hardware, and language features. The resulting architectural debts can limit further research and may harm the validity of scientific results. Furthermore, the lack of documentation and the loss of knowledge as scientists move to other positions limit the maintainability of ESMs. Therefore, the long-term development of ESMs faces unique challenges. Our goal is to support the model developers and research software engineers in program comprehension and to aid architectural decisions.

A note on the term 'model' in climate science: In software engineering and information systems research, modeling is also an essential activity. Conversely to climate science, where climate models are the software systems to simulate the Earth, in software engineering and in information systems research, models are built as abstractions of the software systems themselves. A software architecture description is an example of such a model in software engineering [4], not to be confused with a (software) model of the Earth.

To ensure the maintainability of ESMs, an understanding of the software architecture is necessary. Documentation is usually rare in this domain. In particular, architecture documentation is, if it exists at all, often outdated and incomplete. Thus, rediscovering the architecture of ESMs is a key step to ensure their longevity, as it allows for guiding architectural improvements and identifying areas that should be restructured. Architectural analyses provide an overview of the ESM's structure and dependencies. They allow identifying interfaces that support developers in extending a model or adding alternative sub-models to an ESM. In addition, they help new scientists understand the implementation of the model. Furthermore, based on structural optimizations, developers can improve the architecture over time to facilitate future developments.

Our goal is to support program comprehension and to evaluate how software engineering tools and methods can assist in analyzing and improving the architecture of ESMs.

As contributions of this paper, we provide:

- Static and dynamic analysis of two ESMs, namely the *University of Victoria Earth System Climate Model* (UVic) [5] and the *MIT General Circulation Model* (MITgcm) [6], to recover and analyze their architecture. We compute their coupling complexity and discuss their overall design to support future architecture improvements.
- We introduce a new approach to module interface discovery and apply it to

Fortran systems.

- We also report on the challenges when performing dynamic analyses of ESMs.

We chose UVic and MITgcm for our analysis, as they are used in many research projects, such as SOLAS [7], PalMod [8] [9], CMIP6 [10], and the IPCC report [11]. Both ESMs are implemented in Fortran and run on Unix workstations without specific hardware requirements, and we collaborate with domain experts working with both models as research software engineers. MITgcm is referred to by the domain experts collaborating with us as a good example of a modern ESM, while UVic is a representative of a more traditionally developed ESM whose development began in the 1960s.

We describe our reverse engineering method in Section 2, followed by presenting the results of applying this method to our two case study ESMs in Section 3 (for UVic) and Section 4 (for MITgcm). We evaluate the results in Section 5 and present related work in Section 6 before we conclude the paper in Section 7.

Replication and Data Packages We provide a replication package for our reverse engineering results [12]. This package contains detailed instructions on (a) how to replicate our evaluation, (b) the exact setup used for the dynamic analysis, (c) all scripts and the programs used to perform the static analysis, and (d) our analysis and logging tools. A Docker file allows running the analyses without manually installing the required software.

2. Our Reverse Engineering Method

We employ several methods and techniques to analyze and evaluate software architectures of Fortran-based ESMs. Our analysis process, depicted in Figure 1, consists of eight tasks that allow us to recover an architecture model from static and

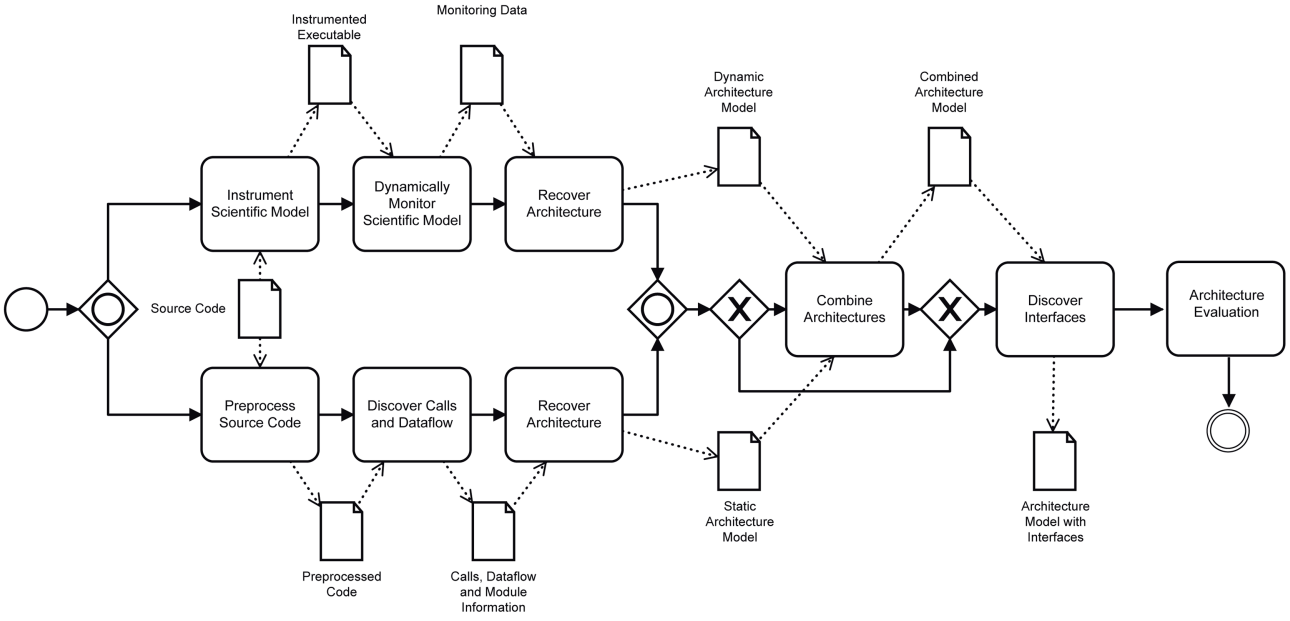

Figure 1. Our analysis process. First, we recover the architecture via reverse engineering with dynamic and static analysis. Then, we combine the results of dynamic and static analysis, interface discovery, and architecture evaluation.

dynamic information of a running scientific model. In addition to methods for collecting execution traces at runtime and for measuring metrics-based software quality (Section 2.1), we include dataflow analysis (Section 2.3) and interface recovery (Section 2.2) to enrich the recovered architectures.

2.1. Dynamic and Static Analysis

For reverse engineering, we use both dynamic and static analysis techniques to gain detailed insights into the software:

Dynamic Analysis: The upper-left pipeline part in **Figure 1** shows that we instrument the Fortran code with monitoring probes. We execute the ESMs utilizing configurations derived from climate science publications provided by our domain experts. The dynamic analysis observes the software at runtime and collects monitoring information about procedure calls. This approach detects which parts of software are actually used and how they behave. To instrument the ESM, we employ the monitoring framework Kieker [13] [14] in combination with the ability of the GNU Compiler Collection and Intel Fortran compilers to instrument functions utilizing the -finstrument-functions option.[1] Our dynamic architecture recovery tool dar[2], then recovers the architecture from the monitoring data.

Static Analysis: The lower-left pipeline part in **Figure 1** shows that to perform the *static analysis* on Fortran code, we first pre-process the source code. To discover calls, dataflows, and accesses to common blocks from Fortran source code, we developed our fxca[3] tool to produce lists of procedure calls, procedure name-to-file mappings, access to common blocks, and dataflows between procedures for the static architecture recovery tool sar[4], which generates the statically recovered architecture model. fxca is an extension of fxtran[5], which provides a concrete syntax tree of Fortran code.

While both dynamic and static analyses usually recover similar architectures, each one can identify certain aspects better than the other. The static analysis can identify common blocks in Fortran, while the dynamic analysis can collect information on procedures used from libraries, which the static analysis cannot provide with the same detail.

To take advantage of both analysis techniques, we then (optionally) merge the results of the static and dynamic analyses into a combined architecture model and discover module interfaces (right part of **Figure 1**).

2.2. Dataflow Analysis

In addition to control flow (*i.e.*, the study of which operations are called from which points in the code), we also study dataflow, *i.e.*, the question of which parts of the code communicate by accessing the same data. In Fortran, the default way

[1]https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html.
[2]https://github.com/cau-se/oceandsl-tools/tree/main/tools/dar.
[3]https://github.com/cau-se/oceandsl-tools/tree/main/tools/fxca.
[4]https://github.com/cau-se/oceandsl-tools/tree/main/tools/sar.
[5]https://github.com/pmarguinaud/fxtran.

of handling parameters in procedures is call-by-reference. This is used to pass data into subroutines and get results back. This has a significant impact on the *dataflow* analysis and the complexity of a system. In case a subroutine modifies call-by-reference parameters, a developer must keep side effects in mind when changing the program, while call-by-value operations have no side effects. A Fortran *function* is similar to a mathematical function, which takes parameter values as inputs and returns a single output value. A Fortran *subroutine* is a block of code that performs some operation on the input variables, and as a result of calling the subroutine, the input variables are modified (call-by-reference).[6] When a distinction between functions and subroutines is not required for our analysis, we will use the term *operation*.

Since the question of whether data flows from one procedure to another via any means of communication (e.g., writing/reading the same file or memory location using array indexing or pointer arithmetic) is, in general, undecidable, we focus on dataflow using global variables which in Fortran are expressed as common blocks [15] and procedure calls. Procedure calls can have call-by-value and call-be reference parameters. Call by value is interpreted as a dataflow from the caller to the callee. For call-by reference, we check whether the reference is used for read or write operations within the procedure. In case only read operations are used, the dataflow is from the caller to the callee. Write operations result in a dataflow from the callee to the caller. In case both operations are used, the architecture model contains a bidirectional dataflow edge, and the analysis graphs provide two edges for each direction. In case there are multiple dataflows between two procedures, they are merged following the same principle as with calls for two reasons:

1) Developers have to consider the dependency between both procedures regardless of the number of parameters.

2) Parameters can have basic or composed data types.

Compound data types can be replaced by a set of parameters with basic types. Common blocks are seen as one single "piece of data," and we do not distinguish between access to different variables in the same block. The reason for this decision is that we assume that the common block is usually seen as a single entity by the developers of the ESM, and therefore, two procedures that access the same common block are conceptually coupled. Our analysis scripts could also be used to analyze a more fine-grained view of the distinction between variables in the same block.

2.3. Interface Discovery

Fortran 77 does not provide syntactical structures that describe interfaces to modules, and even code conforming to Fortran 90 and above does not use this language feature in the two ESMs we analyzed. Thus, the recovered architecture, as initially obtained with Kieker, does not contain interfaces. To support the under-

[6]https://fortranwiki.org/fortran/show/procedure.

standing of the architecture and the code, our second step infers interfaces based on cross-module calls.

Interfaces help developers understand which (public) operations are used by other modules and which (private) operations represent module-internal functionality. Well-designed interfaces group operations together that address a specific concern, e.g., writing and modifying files.

Therefore, we discover interfaces based on calls crossing module boundaries, *i.e.*, calls that originate in one module and call operations in another module. Solely based on these inter-module calls, it is possible to generate candidates for interfaces in four different ways:

1) One large interface for each module pair, resulting in multiple interfaces for one module that may contain the same or similar sets of operations.

2) One interface for all exposed, *i.e.*, externally called operations, per module, resulting in multiple other modules requiring the same interface. These interfaces can become very large and mingle many concerns together.

3) One interface per exposed operation, resulting in many interfaces, making it hard to group associated operations together.

4) Grouping operations together that have the same set of callers. This may still lead to numerous interfaces, but if developers follow a pattern when using operations, this approach will reduce the number of interfaces in contrast to Option 3.

For this paper, we applied Option 4 and implemented four steps to compute interfaces for an architecture description.

For the formal description of our interface discovery process, we use O as the set of all operations in the ESM. The *modules* of the software are denoted M_1 to M_n. These modules constitute a partition of O, *i.e.*, $O = M_1 \cup \ldots \cup M_n$ and $M_i \cap M_j = \varnothing$ for $i \neq j$. The function *module* returns the corresponding module o for any given operation and is defined as:

$$module(o) = M_i \text{ with } o \in M_i$$

All calls between operations in the architecture model are represented by $E_{all} \subseteq O \times O$.

Step One: The first step creates a subset E_{sub} of all distinct calls E_{all} in the execution model where the caller o_r and callee o_p of a call are not in the same module:

$$E_{sub} = \left\{ (o_r, o_p) \mid (o_r, o_p) \in E_{all} \wedge module(o_r) \neq module(o_p) \right\}$$

Step Two: In this step, we identify all provided operations of one module M_j in relation to a requiring module M_i. This is realized by the following function taking the respective indices i and j of the modules. This function *provided_ops* returns the subset of M_j representing the provided operations. This resembles Option 1:

$$provided\_ops(i, j) = \left\{ o_j \in M_j \mid \exists o_i \in M_i \; \exists (o_i, o_j) \in E_{sub} \right\}$$

Step Three: As the previous step creates one provided interface for each requiring

module, in this step, we identify identical sets of provided operations and create a set of tuples P that contain the module index j and the respective set of operations:

$$P = \left\{ \left(j, provided\_ops\left(i, j\right)\right) \mid i \in 0 \cdots n \right\}$$

This step can be augmented by defining a similarity, allowing the merging of sets of operations.

Step Four: Identifies the use of provided interfaces and creates matching required interfaces to the required modules. For this purpose, we define the function *requiring_modules* that returns the set of indices of all requiring modules for a given provided interface I_p, where I_p represents the second element of a tuple of P:

$$requiring\_modules\left(I_p\right) = \left\{ i \mid \exists j, provided\_ops\left(i, j\right) = I_p \right\}$$

After this discovery, the architecture model is enriched by adding interface declarations to modules that provide these interfaces. Similarly, each module that requires an interface of another module obtains a required interface declaration (*requiring_modules*) that contains a reference to the corresponding provided interface (*provided_ops*). Thus, the previously discovered architecture model obtains interface declarations and usage information.

As an example, the results of our interface discovery applied to UVic are presented in fig:uvic-experiment.

2.4. Architecture Evaluation

Software engineering aims to modularize software in a way that cohesion is high while coupling among modules is low, and the overall complexity does not increase more than the size of a software project. These direct metrics can be used to measure the maintainability and comprehensibility of large software projects [16]. Quality metrics help us to evaluate the quality of architectural designs. They provide insights into the structural quality of architectures or parts of them.

Our architecture evaluation uses complexity, coupling, and cohesion metrics based on connections between Fortran procedures grouped by files or directories. Specifically, we used two sets of metrics based on counting and information theory. Both metrics work on a *coupling graph* of the software system, where nodes represent procedures and edges reflect the relationships between these nodes. A relationship can be a call or multiple calls, which both result in one edge, and dataflows, which result in one or two edges, depending on their directionality. Unidirectional dataflow results in an edge representing the flow direction, and bidirectional dataflow results in two edges. Sections 0.0.1 and 0.0.2 present our architecture evaluation via counting metrics and information theory metrics, respectively.

2.4.1. Counting Metrics

Counting metrics compute the number of incoming and outgoing edges for each

node. They work on the coupling graph introduced above. These metrics provide us with values for each procedure and cumulative values for the whole system.

Many different software quality metrics have been studied in the literature. For our study, we chose metrics that have been used in the context of software optimization previously:

- *Structural coupling* (*StrCoup*) is defined as the average in/out-degree of components,
- *Lack of structural cohesion* (*LStrCoh*) is defined as the average number of pairs of unrelated units in a component.

These metrics have been used in [17] for software automatization.

2.4.2. Information Theory Metrics

Allen proposed information theory-based metrics for size, complexity, and cohesion [18]. These metrics are designed to provide a language-independent way of measuring the size, complexity, coupling, and cohesion of software architectures. This metric is used to represent the mental load of a software developer and can, therefore, be used to indicate architecture degradation between versions or the improvement of the software architecture.

These information theory metrics work with unidirectional graphs. The size metric computes the information content of the graph representing the system. The complexity metric is a sum over subgraphs for each node with edges, *i.e.*, for each node, a subgraph is generated containing only those nodes of the system that are directly connected. Finally, the coupling metric operates on the subgraph only containing inter-module edges and computes the complexity of this subgraph.

The size usually grows with the number of nodes (procedures) and edges. The complexity depicts the interconnectedness of the graph. If the complexity increases faster than the size, it is an indicator of architectural degradation. Another indicator of architecture degradation is a faster growth of the coupling metric, which focuses on inter-module connections.

3. Reverse Engineering of UVic

The UVic ESM [5] includes code on which the development started in the 1960s. Our interface discovery (Section 2.3) identified all procedures called from other components and grouped them by use.

In the Energy-Moisture Balance Model (EMBM) component, we discovered a central facade [19] for calling procedures handling access to most parts of the model. The EMBM facilitates two sub-components, *setembm* and *embm*, which provide the facade for the model. They are used to set model variables and control the model during execution. Calls to the model IO are handled by two separate parts, which are interdependent but also duplicate features in other components of the ESM. They are accessible by the facade but are also accessed directly from the outside. Also, some facade calls directly invoke external in-

terfaces. These generate unnecessary coupling between the caller of the facade, the EMBM component, and the component called by the facade. Furthermore, certain facade features are bypassed, and procedures inside the component are directly called.

Figure 2 shows the resulting UVic architecture. The colored boxes represent portions of the model which reside in the same source directory. The white boxes indicate interfaces provided by a component, and the gray boxes indicate requiring or using an interface.

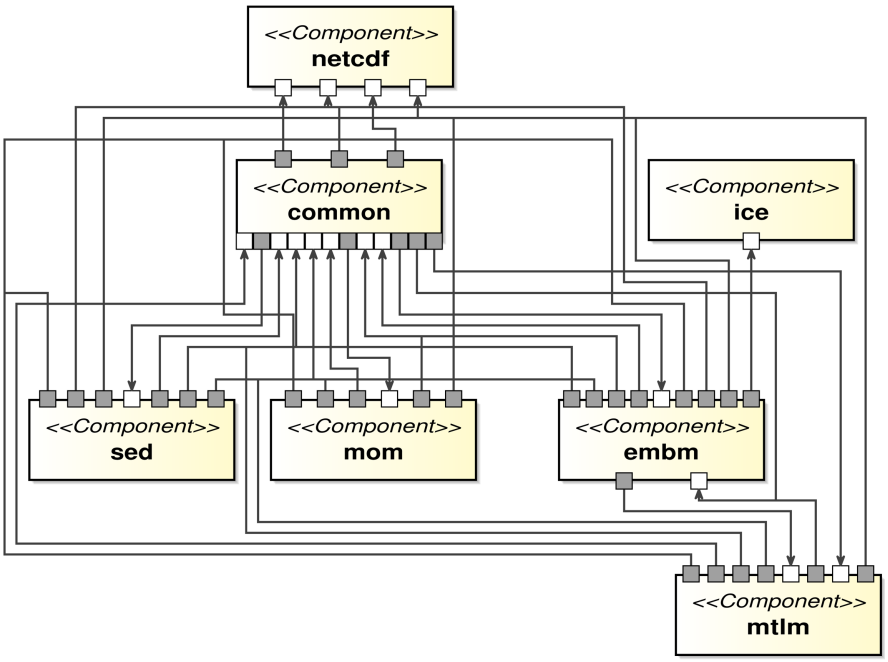

Figure 2. The module structure of the UVic setup. Interfaces are described via ports, with incoming (white) and outgoing (gray) dependencies. Generated with the Kieker architecture visualization Eclipse plugin [20].

All procedures are omitted in Figure 2 to show the overall structure and the inter-dependencies among modules. UVic consists of a central module "modular ocean model" (mom), coupled to an atmospheric energy-moisture balance model (embm), a dynamic-thermodynamic sea ice model (ice), a moses-triffid land model (mtlm), a sediment model (sed), and further shared modules. The code is grouped in multiple source directories that relate to sub-models, common functionality, and library support. It also provides a set of model configuration examples. Each module contains at least setup, IO, restart, and dedicated output procedures. An additional common module contains shared procedures. netcdf is a library used to serialize model output in ESMs, and this module contains procedures to serialize UVic's model state.

The analysis of UVic's reverse-engineered architecture shows an indication of architectural degradation. We illustrate this via some examples of cross-component calls and independent/shared procedures.

Cross-component calls Figure 3 shows that the procedure embmout located in the embm component calls unloadland located in the mtlm component. This indicates either low cohesion since the caller and callee are in different components, or tight coupling since the component embm depends on mtlm as a result of the call.

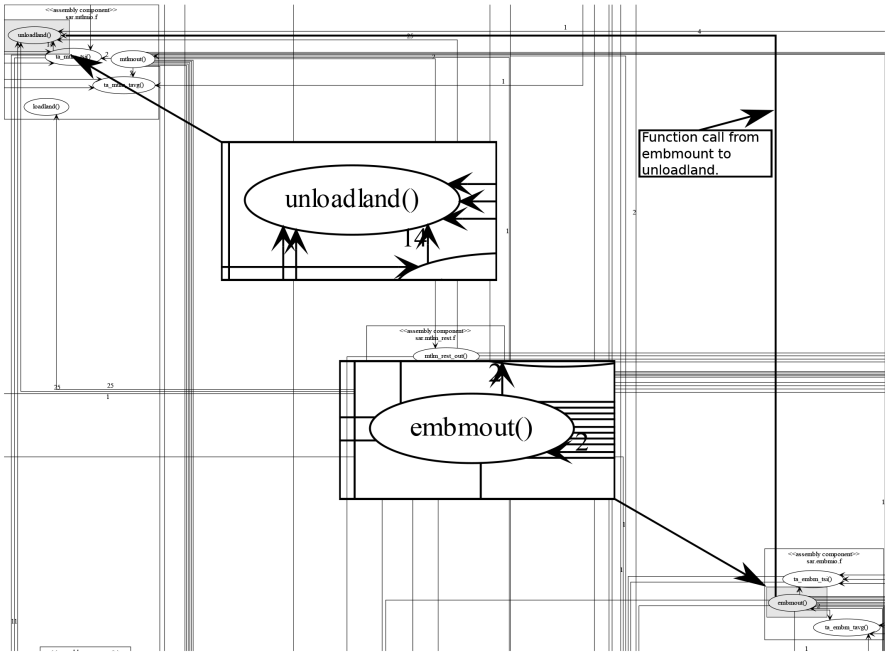

Figure 3. The reverse-engineered module structure of the UVic model Version 2.9.2 shows a call from the procedure embmout() of the embm module to the procedure unloadland() located in the mtlm module. The visualization utilized the Kieker trace analysis with the dot layout tool. Zoom-outs are added by hand for better readability.

Independent Procedures: The procedure glsbc for boundary conditions of the mtlm component bypasses and does not directly depend on the facade of the component. Other components contain similar procedures.

Shared Procedures: The getst procedure located in the common component depends on the getrow procedure located in the mom component. Other components contain procedures to extend the setup of components, e.g., procedures for grid usages are located in the embm component.

4. Reverse Engineering of MITgcm

MITgcm [6] can be used to simulate the atmosphere, the ocean, and the complete Earth's climate system. The model follows a modularization scheme and is shipped with a list of pre-configured model variants that also serve as regression tests for the model. Several tutorial configurations are included. We use the model variants provided by the MITgcm designers for our architectural analysis. Each variant has its own setup instructions, but follows a general scheme of assembling source files, resolving dependencies, compilation, and execution [12] [21].

MITgcm comes with a wide variety of configurations for different experiments. For the analysis, we collected general architectural properties on file-based, directory-based, and combined modularizations.

MITgcm uses a modular concept to organize its files, which may be combined to create specific model variants. The whole project uses base code located in the directories model and eesupp, as well as one directory per module in the pkg directory. In **Figure 4**, the reverse-engineered architecture of the MITgcm model is shown [21].

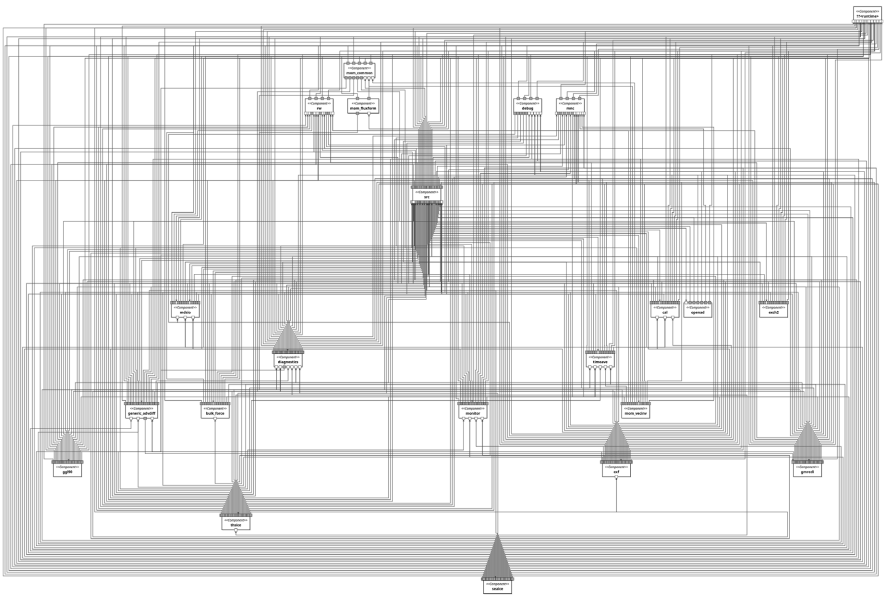

Figure 4. The reverse-engineered module structure of the MITgcm model [21]. The visualization uses the Kieker architecture visualization [20].

Based on this two-level modularization, we are able to identify different kinds of modules. There are modules with only one provided interface, like dic and seaice, or just two, like gchem, which are called either by the main source or by one other module. This pattern applies to most modules categorized by the documentation as specialized and general-purpose modules. There are a few exceptions, e.g., the *Open Boundary Conditions for Regional Modeling* obcs module, which has four groups of procedures that are reflected in the interfaces: (a) access by the main model controlling parameters, initialization, and information exchange, (b) special features that can be turned on (e.g., sponge), (c) code for Orlanski boundary and radiation conditions feature, and (d) diagnosis.

In **Figure 5**, the architecture of the model variant for the Southern Ocean box with Biochemistry so_box_biogeo is depicted as an example. We highlight two areas with a high coupling degree (light gray boxes) that we zoom into. On the left is the component model, and on the right is the components eesupp and optim. All three components are part of the foundational code base [6], providing core, supplemental, and offline optimization features, respectively. All other components have a low coupling degree.

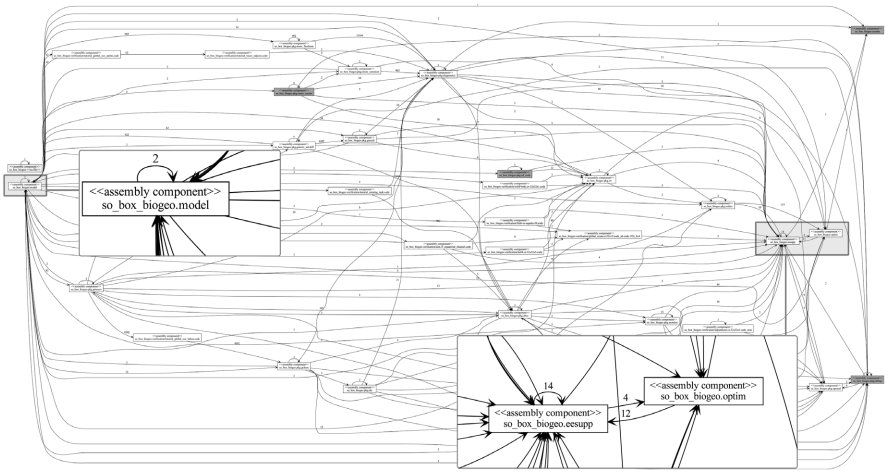

Figure 5. The module structure of the MITgcm model variant so_box_biogeo. White boxes indicate the module structure based on static and dynamic data. Dark gray boxes are derived solely from dynamic data. Light gray areas mark enhanced views. The visualization utilized the Kieker trace analysis with the dot layout tool. Zoom-outs are created by hand for better readability.

In contrast to the domain-specific parts of the model, there exists infrastructure and library functionality, e.g., the diagnostics module that provides logging for all modules and thus is required by many other modules. However, the requiring modules use different subsets of the diagnostics API, resulting in multiple single procedure interfaces. While most modules follow a common pattern for their composition, this is different for the base infrastructure. Here, functions are put together in one directory to address different concerns, e.g., printing messages.

5. Architecture Evaluation of UVic and MITgcm

In this section, we present the results obtained by applying the analysis techniques introduced earlier to the ESMs UVic and MITgcm. Firstly, we give an overview of the evaluation of different analysis methods in Section 5.1, followed by a detailed discussion of UVic and MITgcm in Section 5.2 and Section 5.3.

5.1. Overview

Since we have static and dynamic analyses available, and in the case of static analysis, we can choose between analyzing only call relationships, only dataflow relationships, or both, we have six different analysis methods, displayed in **Figure 6**. The containedness relationships between the analysis methods refer to the identified coupling relationships. As an example, the analysis method **combined/call** is obtained by simply merging the results of the methods **dynamic/call** and **static/call**. To keep this presentation succinct, in the sequel, we mainly focus on the two analysis methods, dynamic/call and combined/both.

To explain the different kinds of software analyses techniques and their relationships, we first apply the two software quality metrics, **StrCoup** and **LStrCoh**, to our ESMs.

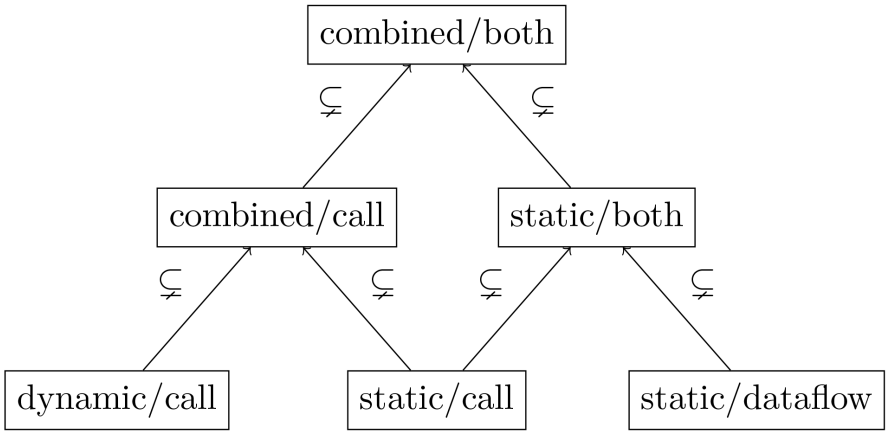

Figure 6. Hierarchy of analysis methods. At the bottom, we analyze dynamic and static call relationships, and static dataflows. Then, we combine the call and static methods. At the top, all methods are combined.

Figure 7 depicts our software quality metrics for the ESMs that we study in this section (namely Global Ocean cs32 × 15, Barotropic Gyre, and Global Oce Biogeo) and for UVic. For each of these four variants, we present the results of the six analysis methods from Figure 6. Hence, we have 24 analysis results in Figure 7. We use shapes to specify the analysis types and colors to specify the specific ESM variants. As an example, circles represent static/callanalyses, and the color red describes MITgcm cs 32 × 15. Therefore, the red circle specifies the values for the tatic/callanalysis of MITgcm cs 32 × 15, with a LStrCoh value of 4204 and a StrCoup value of 39.

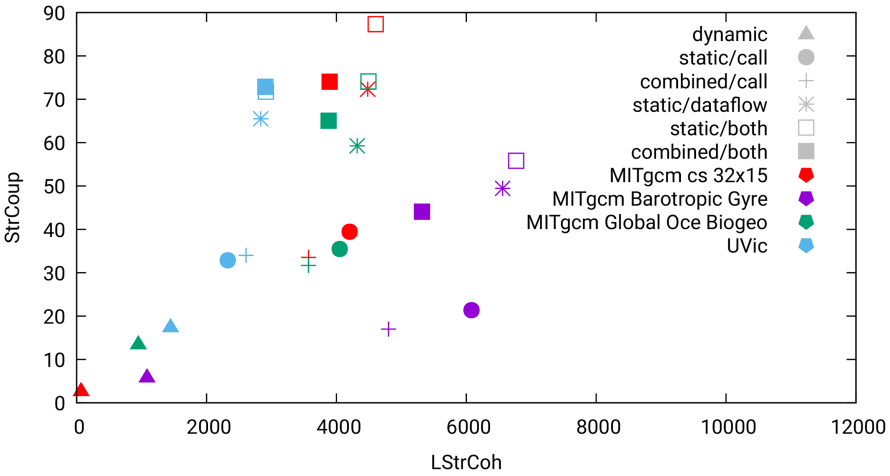

Figure 7. Coupling and cohesion metrics of three MITgcm variants and UVic were obtained by our six analysis methods.

Note that there are significant differences between the analysis results in Figure 6. In particular, the coupling metrics are significantly higher for the analyses including dataflow than for the ones only considering control-flow calls: Each value for analysis including dataflow is higher than each value for each analysis not in-

cluding dataflow. Further, the dynamic-only analyses metrics report much smaller values for both metrics than the analyses, which also include static analysis data.

In order to study the differences between the different analysis results, we next take a closer look at the analyses and consider the components, their size, and connections in the following subsection.

We emphasize again that the metrics given here always measure the existing structure of the software. Therefore, the differences come from applying the different analysis techniques (see Figure 6), not from any difference in software quality. This highlights that metrics like **StrCoup** or **LStrCoh** are performed on *abstractions*, like in our case, coupling graphs, and their results greatly depend on how these abstractions are obtained from the original software.

5.2. Evaluation of UVic

5.2.1. Counting Metrics

We now take a closer look at two analysis results of UVic, namely the dynamic/call analysis in Table 1 and the combined/call in Table 2. For each component of these analyses, we list its size (*i.e.*, the number of its units), as well as its number of independent pairs and its fan-out. The second-to-last line in each table contains the sum of the component sizes (*i.e.*, the number of units found by the corresponding analysis), as well as the average values of the columns indPairs and fanOut—*i.e.*, **LStrCoh** and **StrCoup**, respectively.

There are some notable differences between these analysis results. Obviously, the purely dynamic analysis (Table 1) has the lowest coverage with a size of 305 units, whereas the static analysis (table omitted here) identifies a size of 439 units. The combination of both architecture models of UVic (Table 2) measures 456 units, indicating that there are calls that are not detectable by the static analysis, like library calls, that are covered by the dynamic analysis. The static dataflow analysis has a higher size yet, as often data flows in both directions along calls and access to common blocks, *i.e.*, shared global variables, are also incorporated in the analysis graph. This most complex graph is produced when combining dynamic calls, static calls, and dataflows, resulting in 477.

In general, such differences between dynamic and static analyses are not surprising: While static analysis, by design, takes the entire software into account, dynamic analysis only records information about the parts of the program used in the studied run of the system. However, the dynamic analysis allows us to identify the portions of the ESM that were actually used. Difference graphs can be generated to indicate deviations that might lead to the removal of unused code and refine interfaces.

In UVic, the normalized **StrCoup** values range from 0.06 to 0.07 for the call-only analyses and from 0.14 to 0.15 for the analysis, including dataflow.

Similarly, the normalized values for **LStrCoh** are also very close to each other: With the exception of the dynamic/call analysis (which is an outlier as it is based on a relatively small set of call data, as discussed above), the resulting values for UVic, these are between 0.0121 and 0.0129. In contrast to **StrCoup**, adding data-

flow does not make a big difference here. The reason is that here, we count the number of independent pairs of units, so whether edges are bidirectional or unidirectional does not make a difference.

These normalizations also serve as an indicator that MITgcm's internal structure is, in fact, better than that of UVic, as the normalized **StrCoup** metrics are approximately half of the corresponding value for UVic for all of our analysis techniques.

Table 1. UVic analyzed with dynamic/call.

	COMPONENT NAME	SIZE	INDPAIRS	FAN-OUT
1	common	114	6237	30
2	embm	35	558	25
3	ice	11	45	2
4	mom	72	2477	32
5	mtlm	29	373	19
6	netcdf	18	134	0
7	sed	26	293	15
	Σ/LStrCoh/StrCoup	305	1445.29	17.57
	normalized		0.0155	0.06

Table 2. UVic analyzed with combined/call.

	COMPONENT NAME	SIZE	INDPAIRS	FAN-OUT
1	??<runtime>	32	496	0
2	common	154	11513	64
3	embm	39	701	49
4	ice	12	56	11
5	mom	115	6428	63
6	mtlm	33	487	40
7	netcdf	28	341	8
8	sed	43	841	37
	Σ/LStrCoh/StrCoup	456	2607.88	34.00
	normalized		0.0125	0.07

5.2.2. Information Theory Metrics

As shown in **Table 3**, we computed for UVic Versions 2.6 to 2.9.2 their size and complexity utilizing the Allen metrics [18].

Table 3. Allen complexity and size metric results for Versions 2.6 to 2.9.2 of UVic.

Version	Complexity	Size	Ratio	LOC
2.6	1948.92	2133.95	0.913	49,869
2.7	2372.68	2522.46	0.941	55,188
2.7.1	2372.68	2522.46	0.941	55,213
2.7.2	2372.68	2522.46	0.941	55,218
2.7.3	2376.63	2541.88	0.935	55,922
2.7.4	2830.96	2776.33	1.020	58,719
2.7.5	2800.76	2805.82	0.998	58,995
2.8	2899.16	2727.28	1.063	57,348
2.9	3678.48	3185.78	1.155	60,340
2.9.1	3678.48	3185.78	1.155	60,457
2.9.2	3678.48	3185.78	1.155	60,706

As Table 3 shows, the complexity increases faster, especially from Version 2.8 to Version 2.9, indicating increasing inter-dependencies and, therefore, architecture degradation. In our interviews, the model developers referred to the code as 'spaghetti code.' Between Versions 2.7.5 and 2.8, we can see that the complexity increases while the lines of code decrease, indicating growing interconnections.

5.3. Evaluation of MITgcm Global Ocean

Out of the three MITgcm variants mentioned in Figure 7, we focus on Global Ocean cs32 × 15 since this is the largest of the three. The results and takeaways are comparable for these three variants.

5.3.1. Counting Metrics

In the same way, as for UVic above, we now take a closer look at three analysis results of MITgcm Global ocean cs32 × 15, namely the dynamic/call, combinedPlainCallMap, and ombined/both analyses (see Tables 4-6).

As expected, the absolute values of these metrics differ significantly between the different analyses. One reason for this is that, as discussed before, the size of the models resulting from the analyses is quite different (this is apparent from the "size" column in Table 4, Table 5).

To allow for more meaningful comparisons of the metric values that account for the size differences, we also consider normalized values of **LStrCoh** and **StrCoup** in the last line of each table. These values are obtained by dividing the metric values by (for **StrCoup**), the number of units discovered by the analysis, and (for **LStrCoh**) the squared number of units. We treat these two differently since **StrCoup** counts connections to units and, therefore, is a linear measure, while **LStrCoh** counts pairs of units and is therefore, quadratic.

Table 4. MITgcm analyzed with dynamic/call.

	COMPONENT NAME	SIZE	INDPAIRS	FAN-OUT
1	no-file	2	0	0
2	unknown-component	1	0	1
3	mnc_create_dir	1	0	0
4	bulk_force	1	0	1
5	cal	1	0	1
6	diagnostics	3	2	7
7	eesupp	1	0	0
8	exch2	11	45	3
9	exf	1	0	1
10	ggl90	1	0	1
11	gmredi	1	0	3
12	mdsio	1	0	3
13	mnc	10	37	5
14	monitor	3	1	4
15	openad	1	0	1
16	rw	4	6	0
17	seaice	1	0	1
18	src	51	1201	21
19	thsice	1	0	1
	Σ/LStrCoh/StrCoup	96	68.00	2.84
	normalized		0.0074	0.03

Table 5. MITgcm analyzed with combined/call.

	COMPONENT NAME	SIZE	INDPAIRS	FAN-OUT
1	++no-file++	2	0	0
2	++...	1	0	1
3	??/home/...	1	0	0
4	??(runtime)	47	1081	0
5	bulk_force	11	50	23
6	cal	30	382	16
7	debug	13	73	14

Continued

8	diagnostics	66	2073	57
9	eesupp	1	0	0
10	exch2	111	5999	19
11	exf	42	826	58
12	generic_advdiff	98	4631	37
13	ggl90	16	114	34
14	gmredi	22	223	48
15	mdsio	28	329	19
16	mnc	93	4096	15
17	mom_common	27	348	12
18	mom_fluxform	20	171	20
19	mom_vecinv	12	55	33
20	monitor	27	315	26
21	openad	2	1	12
22	rw	53	1376	22
23	seaice	60	1695	100
24	src	372	68353	221
25	thsice	35	567	68
26	timeave	16	109	17
	Σ/LStrCoh/StrCoup	1206	3571.81	33.54
	normalized		0.0025	0.03

Table 6. MITgcm analyzed with combined/both.

	COMPONENT NAME	SIZE	INDPAIRS	FAN-OUT
1	no-file	2	0	0
2	unknown-component	1	0	1
3	mnc_create_dir.c	1	0	0
4	<runtime>	47	1054	434
5	bulk_force	11	50	30
6	cal	32	443	33
7	debug	13	73	48
8	diagnostics	67	2139	126

Continued

9	eesupp	1	0	0
10	exch2	111	5999	69
11	exf	42	826	65
12	generic_advdiff	98	4631	48
13	gg190	16	114	43
14	gmredi	22	223	59
15	mdsio	34	512	68
16	mnc	93	4096	49
17	mom_common	27	348	18
18	Mom_fluxform	20	171	21
19	mom_vecinv	12	55	34
20	monitor	27	315	35
21	openad	4	6	13
22	rw	54	1429	72
23	seaice	60	1695	109
24	src	393	76374	439
25	thsice	37	638	78
26	timeave	16	109	33
	Σ/LStrCoh/StrCoup	1241	3896.15	33.54
	normalized		0.0025	0.06

From our results, we can also observe the above-mentioned difference between analyses including dataflow and those using only call data (comparing the analyses combined/call from Table 5 and ombined/both from Table 6):

While the number of detected units is pretty close (the analysis including dataflow adds a further 35 units; these are most likely Fortran common blocks), the average fan-out of the components (*i.e.*, **StrCoup**) is more than doubled by adding dataflow analysis. This is because dataflow usually exists in both directions between two operations, if data flows at all (e.g., with parameters in one direction and return values in the other direction), while call relationships often only exist in one direction. Therefore, the number of directed edges is much larger for a dataflow analysis.

Note also that the normalized values of **StrCoup** are very close to each other: For the variant Global Ocean cs32 × 15 shown in the tables, the normalized values of **StrCoup** range from 0.03 to 0.07 (compared to the original ranges 2.84 to 87.32 for MITgcm). More interestingly, the thus-normalized **StrCoup** is 0.03 for all

MITgcm analyses, including only call relationships (Table 3 and Table 5, and the analysis results found in the supplementary material), and is 0.06 or 0.07 for the MITgcm analyses that include dataflow relationships. Hence, the normalizations indicate that the increased coverage of the analysis methods explains some of the differences observed in the metrics. This also shows that normalization captures the difference between analyses including dataflow and those omitting it better than the original metrics.

Similarly, the normalized values for **LStrCoh** are also very close to each other: With the exception of the dynamic/call analysis (which is an outlier as it is based on a relatively small set of call data, as discussed above), the resulting values for MITgcm are all between 0.0025 and 0.0031. These measurements are only a 1/4 of UVic measurements, which are between 0.0121 and 0.0129, indicating that the overall architecture of MITgcm is in better shape.

In contrast to **StrCoup**, adding dataflow does not make a big difference here. This is because here, we count the number of independent pairs of units, so whether edges are bidirectional or unidirectional does not make a difference.

5.3.2. Information Theory Metrics

For MITgcm, we computed complexity and size with the Allen metrics and estimated the coupling degree for different variants and versions. Due to the large number of variants, we display the results as a histogram in Figure 8. It shows that most variants range from 6000 to 7000, and more complex variants are less frequent.

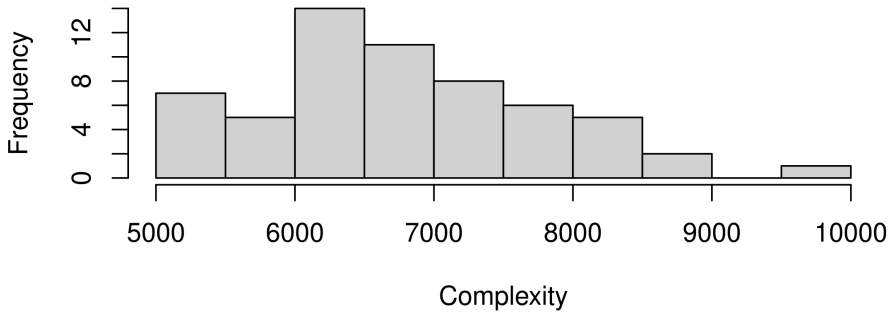

Figure 8. Architecture complexity for all MITgcm model variants.

We compared the size and complexity based on the Allen metrics, see Figure 9. The data suggests that complexity grows faster than size. The fitted curve describes the relationship of the complexity to the size as logarithmic.

There is a group of outliers of 3 ESM variants between 5000 and 6500 bits of complexity that have exceptionally high values. These variants use the fizhi module that addresses atmosphere physics. This indicates that this module introduces a lot of procedures, but fewer calls than in the remaining model.

As shown in Figure 9, the largest and most complex ESM variant is Global Ocean cs32 × 15. Thus, we picked this variant to compute the Allen metrics [18] for all versions of the ESM.

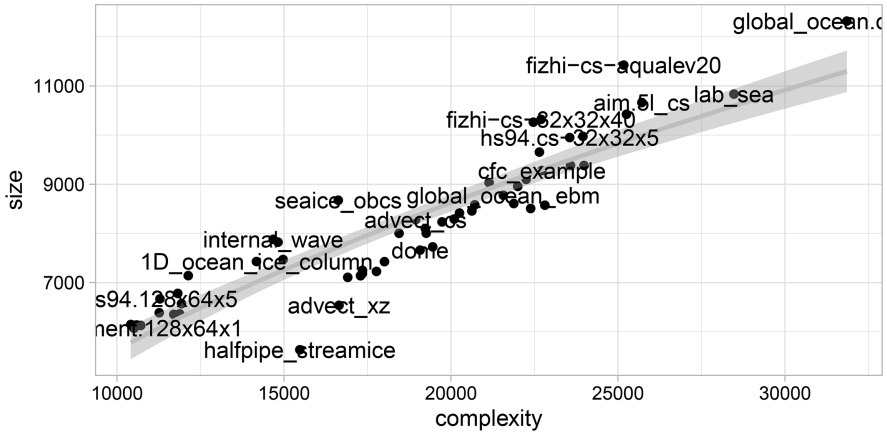

Figure 9. Architecture complexity to size relationship for all selected MITgcm variants.

In Table 7, we depict major versions of MITgcm, whereas in fig:mitgcm:allen, we show the measurements for all versions from checkpoint54 to checkpoint69a.

We omit the lines of code measurements (LOC), as due to the build system of MITgcm, we could not safely detect whuch parts of the code are actually used for a model variant, and we could not extract this information from the Fortran parser used to collect calls and dataflows.

Table 7. Allen complexity and size metric results for Versions checkpoint54 to checkpoint69a (key versions only).

Version	Complexity	Size	Ratio
checkpoint54	32085.59	12445.79	2.578
checkpoint55	32085.59	12445.79	2.578
checkpoint57	32085.59	12445.79	2.578
checkpoint58	32085.59	12445.79	2.578
checkpoint59a	33065.04	13361.10	2.475
checkpoint60	33357.39	13537.92	2.464
checkpoint61	33357.39	13537.92	2.464
checkpoint62	33357.39	13537.92	2.464
checkpoint63	31161.24	13465.16	2.314
checkpoint64	32759.68	13476.95	2.431
checkpoint65	32602.79	13406.23	2.432
checkpoint66a	32492.55	13382.66	2.428
checkpoint67	33092.39	13443.58	2.462
checkpoint68a	34743.43	13644.12	2.546
checkpoint69a	34751.12	13608.71	2.556

As the table suggests complexity and size increase over time with some exceptions indicating major architectural changes. As MITgcm has the aim to be a modular ESM that can be used for a wide variety of experiments, such efforts are necessary to be able to be used in that manor. The ratio between complexity and size show that the ESM improves over time, but in the recent version, the ratio increased again.

In Figure 10, we can see a sudden jump in size around checkpoint59a, which also increased the complexity, while at the same time, efforts were made to reduce the coupling of modules in MITgcm, effectively improving the architecture. Another effort to reduce coupling was made between checkpoint62f and checkpoint62g, resulting in less complex major release checkpoint63. After that, the complexity started to increase again without an impact on the overall size and with no relevant impact on the coupling. This shows that functionality inside modules was added or improved. Only in more recent versions did new functionality arrive and cause a large increase in coupling and complexity.

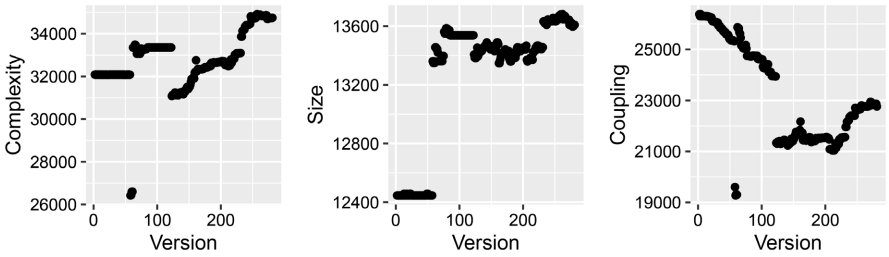

Figure 10. Allen complexity, size, and coupling metric results of Global Ocean cs32 × 15 for all Versions from checkpoint54 to checkpoint69a.

6. Related Work

Alexander and Easterbrook [22] analyzed climate model software architectures by considering the model source code. The focus was on analyzing the modularization based on the relative sizes of software components, as well as data flow between components and design decisions. They visualized the software architecture, including the dependencies between the components. The code analysis was performed on preprocessed code with external software. They used a predefined scenario to make the analyses of different models comparable. The developers were contacted for information on design decisions. Different from this work, we conduct a more fine-grained, tool-based architecture analysis employing a combination of static and dynamic analysis.

Simm *et al.* [23] interviewed model developers and environmental scientists about their approaches to engineering computer-based environmental models. At the architectural level, they found that models are often monolithic and utilize specific coupling interfaces to link to other models. This observation is confirmed by our tool-based architecture analysis.

Basic architectural diagrams are often created manually for individual models [24]-[26]. These descriptions only show a general software architecture based on

the functionality of the models and their components. The actual climate model's architecture can differ from these architectural diagrams, due to variant configurations. Compared to these works, we combine static and dynamic analysis to reverse engineer fine-grained architecture information of earth system models.

Riva and Rodríguez [27] also use a combination of static and dynamic analysis for architecture reconstruction, but they used a simulation technique, while the monitoring tools we use can be run on a production system and, therefore, take runtime information into account.

7. Conclusions and Future Work

We analyzed the two ESMs, UVic and MITgcm, to recover their architecture and understand the key properties of their software structure. In both cases, we could not find large portions of unused code. This was expected to be the case for MITgcm, as it emphasizes its modular approach. Based on previous interviews with climate scientists [28], we expected that this would be an issue with UVic. However, there were only minor differences between the static and dynamic analysis regarding the ESM structure. Some differences can be attributed to the use of data handling libraries, which were not detected by the static analysis. Parts only visible in the static analysis were, e.g., debug and logging code, which can be switched on and off at the start time of the model. In the UVic architecture, we found that the major modules heavily rely on each other, which is observable in the directory and file-based modularization scheme.

Since our data shows that the results of dynamic and static analyses are similar to each other, for some analyses, it might be enough to perform only the (cheaper) static analysis, which is an interesting difference to, for example, interactive web-based systems. Note, however, that there are many types of analyses that simply cannot be done in a static way, such as profiling analyses with the goal of runtime optimizations.

Additionally, our findings confirm our expectations that the architecture of an ESM differs significantly from that of, e.g., web-based systems, where the differences between dynamic and static analyses are much more significant [29]. A possible reason for this is that the design of an ESM can anticipate the way the code will execute much better than the design of an event-based system, where much of the actual behavior depends on events that cannot be predicted at design time.

With the UVic log, we have also shown that our logging and analysis tools are capable of creating and processing huge logs, which is essential in going forward to analyze even larger community ESMs.

ESMs process big datasets. The coupling of sub-models is, therefore, also based on data. In our current analysis, we considered dataflow using Fortran common blocks, but did not analyze, e.g., data flow resulting from writing and reading the same files. Based on a more complete picture of control and data flow analyses, we will revisit model coupling issues, which we identified in the ESMs.

We provided a new approach to discover interfaces based on usage patterns

between modules. To potentially increase the quality of these interfaces, one could merge and split these interfaces based on shared use of data types, textual similarity, e.g., identical prefixes and suffixes, substring similarity or distance functions, such as Levenshtein distance [30], which we will address in future work.

Funding

This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) (Grant No. HA 2038/8-1-425916241).

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Johanson, A. and Hasselbring, W. (2018) Software Engineering for Computational Science: Past, Present, Future. *Computing in Science & Engineering*, **20**, 90-109. https://doi.org/10.1109/mcse.2018.021651343

[2] Jung, R., Gundlach, S. and Hasselbring, W. (2022) Software Development Processes in Ocean System Modeling. *International Journal of Modeling, Simulation, and Scientific Computing*, **13**, Article ID: 2230002. https://doi.org/10.1142/s1793962322300023

[3] Reussner, R., Goedicke, M., Hasselbring, W., Vogel-Heuser, B., Keim, J. and Märtin, L. (2019) Managed Software Evolution. Springer. https://link.springer.com/book/10.1007/978-3-030-13499-0

[4] Hasselbring, W. (2018) Software Architecture: Past, Present, Future. In: Gruhn, V. and Striemer, R., Eds., *The Essence of Software Engineering*, Springer International Publishing, 169-184. https://doi.org/10.1007/978-3-319-73897-0_10

[5] Weaver, A.J., Eby, M., Wiebe, E.C., Bitz, C.M., Duffy, P.B., Ewen, T.L., *et al.* (2001) The Uvic Earth System Climate Model: Model Description, Climatology, and Applications to Past, Present and Future Climates. *Atmosphere-Ocean*, **39**, 361-428. https://doi.org/10.1080/07055900.2001.9649686

[6] Artale, V., Calmanti, S., Carillo, A., Dell'Aquila, A., Herrmann, M., Pisacane, G., *et al.* (2009) An Atmosphere-Ocean Regional Climate Model for the Mediterranean Area: Assessment of a Present Climate Simulation. *Climate Dynamics*, **35**, 721-740. https://doi.org/10.1007/s00382-009-0691-8

[7] Brévière, E.H.G., Bakker, D.C.E., Bange, H.W., Bates, T.S., Bell, T.G., Boyd, P.W., *et al.* (2015) Surface Ocean-Lower Atmosphere Study: Scientific Synthesis and Contribution to Earth System Science. *Anthropocene*, **12**, 54-68. https://doi.org/10.1016/j.ancene.2015.11.001

[8] Pahlow, M., Chien, C., Arteaga, L.A. and Oschlies, A. (2020) Optimality-Based Non-Redfield Plankton-Ecosystem Model (OPEM V1.1) in Uvic-ESCM 2.9—Part 1: Implementation and Model Behaviour. *Geoscientific Model Development*, **13**, 4663-4690. https://doi.org/10.5194/gmd-13-4663-2020

[9] Chien, C., Pahlow, M., Schartau, M. and Oschlies, A. (2020) Optimality-Based Non-Redfield Plankton-Ecosystem Model (OPEM V1.1) in UVic-ESCM 2.9—Part 2: Sensitivity Analysis and Model Calibration. *Geoscientific Model Development*, **13**, 4691-4712. https://doi.org/10.5194/gmd-13-4691-2020

[10] Mengis, N., Keller, D.P., MacDougall, A.H., Eby, M., Wright, N., Meissner, K.J., *et al.*

(2020) Evaluation of the University of Victoria Earth System Climate Model Version 2.10 (UVic ESCM 2.10). *Geoscientific Model Development*, **13**, 4183-4204. https://doi.org/10.5194/gmd-13-4183-2020

[11] Stocker, T.F., *et al.* (2014) Climate Change 2013—The Physical Science Basis: Working Group I Contribution to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change. Cambridge University Press.

[12] Jung, R., Schnoor, H. and Hasselbring, W. (2024) Replication Package for: Software Architecture Evaluation of Earth System Models. https://zenodo.org/records/11371816

[13] van Hoorn, A., Waller, J. and Hasselbring, W. (2012) Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. *Proceedings of the* 3*rd ACM/SPEC International Conference on Performance Engineering*, Boston, 22-25 April 2012, 247-248. https://doi.org/10.1145/2188286.2188326

[14] Hasselbring, W. and van Hoorn, A. (2020) Kieker: A Monitoring Framework for Software Engineering Research. *Software Impacts*, **5**, Article ID: 100019. https://doi.org/10.1016/j.simpa.2020.100019

[15] Overbey, J.L., Negara, S. and Johnson, R.E. (2009) Refactoring and the Evolution of Fortran. 2009 *ICSE Workshop on Software Engineering for Computational Science and Engineering*, Vancouver, 23 May 2009, 28-34. https://doi.org/10.1109/secse.2009.5069159

[16] Bogner, J., Wagner, S. and Zimmermann, A. (2017) Automatically Measuring the Maintainability of Service- and Microservice-Based Systems. *Proceedings of the* 27*th International Workshop on Software Measurement and* 12*th International Conference on Software Process and Product Measurement*, Gothenburg, 25-27 October 2017, 107-115. https://doi.org/10.1145/3143434.3143443

[17] Candela, I., Bavota, G., Russo, B. and Oliveto, R. (2016) Using Cohesion and Coupling for Software Remodularization: Is It Enough? *ACM Transactions on Software Engineering and Methodology*, **25**, 1-28. https://doi.org/10.1145/2928268

[18] Allen, E.B. (2002) Measuring Graph Abstractions of Software: An Information-Theory Approach. *Proceedings* 8*th IEEE Symposium on Software Metrics*, Ottawa, 4-7 June 2002, 182-193. https://doi.org/10.1109/metric.2002.1011337

[19] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1996) Design Patterns—Elements of Reusable Object-Oriented Software. Addison-Wesley.

[20] Kieker Project (2021) Kieker Development Tools.

[21] Adcroft, A., *et al.* (2022) MITgcm's User Manual.

[22] Alexander, K. and Easterbrook, S.M. (2015) The Software Architecture of Climate Models: A Graphical Comparison of CMIP5 and EMICAR5 Configurations. *Geoscientific Model Development*, **8**, 1221-1232. https://doi.org/10.5194/gmd-8-1221-2015

[23] Simm, W.A., Samreen, F., Bassett, R., *et al.* (2018) SE in ES: Opportunities for Software Engineering and Cloud Computing in Environmental Science. *Proceedings of the* 40*th International Conference on Software Engineering*: *Software Engineering in Society*, Gothenburg, 27 May-3 June 2018, 61-70. https://dl.acm.org/doi/10.1145/3183428.3183430

[24] Collins, W.J., Bellouin, N., Doutriaux-Boucher, M., Gedney, N., Halloran, P., Hinton, T., *et al.* (2011) Development and Evaluation of an Earth-System Model-HadGEM2. *Geoscientific Model Development*, **4**, 1051-1075. https://doi.org/10.5194/gmd-4-1051-2011

[25] Giorgetta, M.A., Jungclaus, J., Reick, C.H., Legutke, S., Bader, J., Böttinger, M., *et al.*

(2013) Climate and Carbon Cycle Changes from 1850 to 2100 in MPI-ESM Simulations for the Coupled Model Intercomparison Project Phase 5. *Journal of Advances in Modeling Earth Systems*, **5**, 572-597. https://doi.org/10.1002/jame.20038

[26] Hurrell, J.W., Holland, M.M., Gent, P.R., Ghan, S., Kay, J.E., Kushner, P.J., *et al*. (2013) The Community Earth System Model: A Framework for Collaborative Research. *Bulletin of the American Meteorological Society*, **94**, 1339-1360. https://doi.org/10.1175/bams-d-12-00121.1

[27] Riva, C. and Rodríguez, J.V. (2002) Combining Static and Dynamic Views for Architecture Reconstruction. 6*th European Conference on Software Maintenance and Reengineering* (*CSMR* 2002), Budapest, 11-13 March 2002, 47. https://ieeexplore.ieee.org/document/995789

[28] Jung, R., Gundlach, S. and Hasselbring, W. (2022) Thematic Domain Analysis for Ocean Modeling. *Environmental Modelling & Software*, **150**, Article ID: 105323. https://doi.org/10.1016/j.envsoft.2022.105323

[29] Schnoor, H. and Hasselbring, W. (2020) Comparing Static and Dynamic Weighted Software Coupling Metrics. *Computers*, **9**, Article No. 24. https://doi.org/10.3390/computers9020024

[30] Levenshtein, V.I. (1966) Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, **10**, 707-710.