Architecture Recovery from Fortran Code with Kieker

Reiner Jung
Kiel University, Germany
reiner.jung@email.uni-kiel.de

Henning Schnoor
Kiel University, Germany
henning.schnoor@email.uni-kiel.de

Sven Gundlach
Kiel University, Germany
sven.gundlach@email.uni-kiel.de

Wilhelm Hasselbring
Kiel University, Germany
hasselbring@email.uni-kiel.de

Abstract

Scientific models are software systems, which are key to understand and assess a range of challenges, such as climate change mitigation. However, such models are usually developed over decades. To support program comprehension for software maintenance and restructuring, we designed an architecture recovery process for Fortran-based scientific models utilizing Kieker 4 C to collect call traces at runtime. Furthermore, we derive structural information from the recovered architecture. In this paper, we present our analysis process and some results from analyzing three scientific models. Additionally, we describe how to use the information obtained by our analysis to identify possible optimizations of the scientific models.

1 Introduction

Scientific models play a vital role in understanding and assessing current challenges, including economic and ecological impacts of resource extraction, and climate change mitigation. These models are mainly implemented in Fortran and have been developed over a long period of time, e.g., the UVic Earth System Climate Model has been developed since the 1970's. The code typically uses pre-processing directives such as #ifdef and #include to handle variants. In addition, different incompatible Fortran dialects are used together in the source code. Therefore, static source code analysis is limited and may not cover the correct parts of a variant when recovering the architecture. While we still use and improve static code analysis for architecture recovery, we need a second approach to ensure that we get a complete picture. Therefore, we can use Kieker 4 C, which also supports other programming languages compiled into native machine code. In addition, scientists can also gain insights into the model's performance in this way.

In this paper, we explain the process to apply Kieker [1, 2], recover the architecture and infer structural information of the recovered model, as well as show preliminary results from scientific models.

Section 2 introduces the recovery process. Section 3 presents preliminary results on the recovered architectures. Finally, Section 4 provides a short summary and an outlook on model recovery and optimization.

2 Recovery Process

The generic recovery process consists of seven steps, whereby especially the recovery can be repeated to improve the results. In contrast to enterprise software, scientific model use their own build system, which often also covers configuration and feature selection. Furthermore, the software is usually non-interactive and produces one huge trace, e.g., in one experiment we create 1.5 TB of compressed monitoring data.

Step 1: Understand the Model's Build Process Before we can perform any analysis, we need to understand the build process of a scientific model and how to instrument it with Kieker.

Step 2: Configuration and Parameter Setup It is of great importance to develop a model setup that ensures that all required parts of the model are executed, but also does not take an excessive amount of time to execute.

Step 3: Instrument with the Monitoring Probes There are different instrumentation approaches available with Kieker 4 C, but in our analyses with scientific models, we rely on the ability of the GNU Compiler Collection (GCC) and the Intel Fortran compiler (ifort, version 19.0.4 and 2021.1.2) to weave in instrumentation probes (command line option `-finstrument-functions`). Both compiler suites are capable to instrument all functions, procedures and subroutines (we refer to these as *operations*) in Fortran, C and other compatible languages. It is possible to select only a subset of these operations. Besides activating instrumentation by the compiler, we have also included the Kieker monitoring library in the build path. This library provides an implementation of the two probes with the following signatures:

```
void __cyg_profile_func_enter
    (void *this_fn, void *call_site);
void __cyg_profile_func_exit
    (void *this_fn, void *call_site);
```

The Kieker 4 C-library implements both probe functions and produces with Kieker the minimal set of trace events, i.e., `BeforeOperationEvent`, `AfterOperationEvent`, and `TraceMetadata`.

In Java, we can obtain method and class names at runtime. This is not possible in compiled Fortran code. Instead, the compiler can append debug symbols to the program, which are then used to resolve name during analysis. Our analysis tools automatically call this program to extract the necessary information to resolve the names in the Kieker events.

The names in Fortran are case-insensitive, but the symbols in object code are case-sensitive. Thus, compilers convert names to lowercase and prefix them with `_`. During recovery, we remove these `_`, otherwise it deviates from the source code.

Details on how to introduce compiler flags and the library into the respective models can be found in our replication package [3].

Step 4: Model Execution When the scientific model is set up, we execute it to collect monitoring data. Depending on the model and setup, this can take minutes or hours. For the actual collection of the monitoring data, we use the Kieker collector[1] to receive all monitoring data, compress it and store it. The collector can be started on a different machine and produce Kieker logs, including splitting up logs to avoid file size issues.

In case the collector is too slow to process all events, as it instantiates new objects for every event and facilitate event modifications, we can use NetCAT as a server which is available for various platforms. Together with `split`, it is possible to create a setup that allows to store huge monitoring logs. These dumps can then only be read by the TCP reader stage of Kieker when replaying the log.

To execute and monitor the scientific model, we first start the collector or NetCAT and then start the instrumented scientific model.

Step 5: Monitoring Data Analytics After the model run, we analyze the collected monitoring data. Depending on how the monitoring data was collected (see above), we use the file reader or TCP reader stage with our analysis tools (cf. [3]). The standard Kieker `trace-analysis` cannot be used, as traces can become huge and will not fit in memory. Fortunately, the architecture reconstruction only relies on operation calls and can be created from the log data with a minimal memory footprint. Our tools utilize the Kieker Architecture Model, but other architecture models can be used too.

The analysis produces a basic architecture model, based on observations and debug symbols. To improve the results, it is possible to generate a map file that lists all the functions found in the monitoring data, the file in which they are defined, and add a column to identify an additional grouping. The directory structure of the source code is an example for such a grouping. Our tooling provides options to generate such mapping files automatically, which can then be tweaked to satisfy the engineer.

Besides a dynamic analysis with Kieker, we also perform static code analysis and merge the recovered static architecture. All elements from these architectures are tagged to indicate their origin. This allows to identify whether an operation or component exists in the static or dynamic recovered architecture. It is also possible to join multiple dynamic analyses to identify shared components. This is helpful when analyzing variants and versions.

Step 6: Recover Interfaces While newer Fortran dialects support interfaces comparable to interfaces of modules and units in Pascal and Modula-2, respectively, older versions do not have any interface information. Therefore, we aim to recover interfaces based on the calls between two components. There are different strategies available, for example, all calls from one component to one other component are grouped into one interface. This will produce very large interfaces and is not helpful for program comprehension. Therefore, we collect for each provided operation all its callee and caller components. Then, operations with an identical set of caller components are put into one provided interface of the callee component. This will create too many interfaces, as not every component will use all operations provided by another component. However, it provides a good starting place for semi-automated refinement.

Step 7: Inspect the Recovered Architecture There are different tools available to visualize and measure the recovered architecture. First, the Kieker development tools include two views that allow to view the architecture in Eclipse utilizing KLighD[2]. One view only addresses the composition of the assembly model without links based on calls, the other one includes call information. Both visualizations allow to inspect the recovered model interactively.

Second, the `mvis` command line tool allows to visualize, inspect and measure recovered architectures. It can color the model based on the data source of a recovery, which is helpful when mixing different recovered architectures from dynamic and static recovery. For example, to identify components and operations present in both architectures, shared elements can be colored differently. In addition, `mvis` is able to compute different metrics regarding the architecture.

3 Analyzed Scientific Models

So far, we analyzed three Earth System Models.

[1] The collector is the successor of the Kieker Data Bridge `https://kieker-monitoring.readthedocs.io/en/latest/kieker-tools/Collector.html`

[2] KLighD `https://github.com/kieler/KLighD`

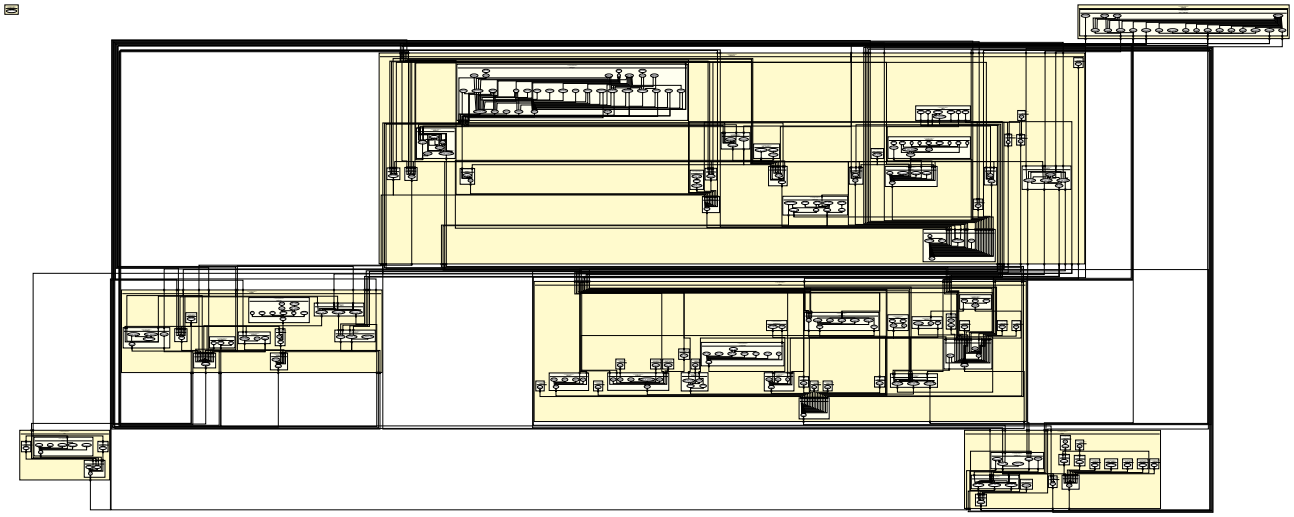

Figure 1: UVic model (v2.9.2) architecture with two levels of components

UVic is a fairly old model, with some of its ocean model introduced in the early 1970's.[3] Its code has been adapted to new computer architectures multiple times through its lifetime. Other parts of UVic are based on work from the late 1990's and 1970's, but were integrated into the UVic ESM in 2001. Its code base is mainly Fortran 77 and Fortan 90 code and uses one central configuration and build settings file.

As an example result, we briefly report on the architecture recovery of UVic. Figure 1 depicts the model with two levels of components reflecting the file and directory structure of the code, and therefore, the structure the developers use. Due to the page limit, we cannot give a full overview in the present paper. We refer to the replication package for details [3].

MITgcm is a newer model developed at MIT since 1998.[4] It has extensive documentation, a simple feature management in its build system, and a coarse grained conceptual architecture. It is mostly implemented in Fortran 77 and Fortran 90, apart from some external C packages. Its configuration is scattered across many different files and its build system is also specifically designed for this model.

Shallow-Water Model is a newer model. Its development started in 2010 at GEOMAR and the model is completely implemented with Fortran 90.[5] It uses Fortran modules and interfaces, which we map to components in the recovered architecture.

4 Conclusions

Dynamic recovery provides information on the used part of software across languages and includes timing information. Depending on the libraries, we can trace into library functions and get a wider picture than a static code analysis can provide. However, static information provided by debug symbols supports the recovery process, as well as, path and map based grouping strategies which rely on static information.

File and directory based grouping of operations is a starting point for the recovery. On inspection, it might be the case that directories include too many files and too few operations for a useful architecture. Thus, the strategy can be modified based on user input informed by architecture visualizations.

In future, we will use the data collected as described in this paper to optimize the mentioned scientific models with respect to standard coupling and cohesion metrics. Preliminary results indicate that UVic gains significantly more from such optimization than MITgcm. This confirms our expectations, since MITgcm's software architecture is more modern than UVic's. In addition, we plan interviews with ESM developers to discuss our methods and findings.

Acknowledgment Funded by KMS Kiel Marine Science - Centre for Interdisciplinary Marine Science at Kiel University and the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation), grant no. HA 2038/8-1 – 425916241.

References

[1] A. van Hoorn, J. Waller, and W. Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis". In: *Proceedings ICPE 2012*. Apr. 2012, pp. 247–248. DOI: 10.1145/2188286.2188326.

[2] W. Hasselbring and A. van Hoorn. "Kieker: A monitoring framework for software engineering research". In: *Software Impacts* 5 (Aug. 2020). DOI: 10.1016/j.simpa.2020.100019.

[3] R. Jung et al. *Replication package*. Aug. 2022. DOI: 10.5281/zenodo.7020117.

[3] UVic http://terra.seos.uvic.ca/model/

[4] MITgcm https://github.com/MITgcm/MITgcm

[5] SWM https://git.geomar.de/swm/swm