Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

FAKULTÄT
FÜR MATHEMATIK, INFORMATIK
UND NATURWISSENSCHAFTEN

Unittests für die Klimamodellentwicklung

Dissertation

zur Erlangung des akademischen Grades

Dr. rer. nat.

an der Fakultät für Mathematik, Informatik und Naturwissenschaften

der Universität Hamburg

eingereicht am

Fachbereich Informatik

durch

Christian Hovy

Hamburg, Januar 2020

Gutachter

Prof. Dr. Thomas Ludwig, Universität Hamburg

Prof. Dr. Axel Schmolitzky, HAW Hamburg

Tag der Disputation

18. Dezember 2020

URN urn:nbn:de:gbv:18-ediss-90774

Zusammenfassung

Moderne Klimamodelle, d.h. Computerprogramme zur numerischen Simulation klimatologischer Prozesse, werden von interdisziplinären Teams entwickelt und enthalten zahlreiche miteinander gekoppelte Teilmodelle, die unterschiedliche Komponenten und Prozesse des Klimasystems abbilden. Die steigende Komplexität der Modelle sowie die Notwendigkeit, sich an immer leistungsfähigere Rechnerarchitekturen anzupassen, ist mit hohen softwaretechnischen Herausforderungen verbunden. Dies betrifft insbesondere auch das Testen dieser Modelle. Diese Dissertation im Fach Informatik befasst sich daher aus softwaretechnischer Sicht mit dem Testen von Klimamodellen und der Frage, wie sich die Erstellung von Unittests, d.h. von isolierten Tests kleinerer Codeeinheiten auf Prozedur- oder Modulebene, in diesem Kontext unterstützen lässt.

Im Rahmen einer internationalen Studie wird die Softwaretestpraxis in der Klimamodellierung an vier Forschungsinstituten untersucht. Ein zentrales Ergebnis der Studie ist, dass Unittests dabei selten zum Einsatz kommen. Stattdessen werden sowohl für die wissenschaftliche Evaluation der Modelle als auch beim Regressionstesten im Wesentlichen sog. Ende-zu-Ende-Tests eingesetzt. Dabei wird jedes Mal das Hauptprogramm der Modellsoftware mit einer ausgewählten Konfiguration ausgeführt und alle Phasen einer Simulation durchlaufen.

Aufbauend auf den Ergebnissen der Studie wird argumentiert, dass Unittests eine sinnvolle Ergänzung dieser Testpraxis darstellen könnten, um die Produktivität der Entwicklung zu erhöhen. Dies setzt jedoch voraus, dass der Aufwand für die Erstellung der Tests in einem vertretbaren Verhältnis zum Nutzen steht. Um den Aufwand für die Erstellung von Unittests für Klimamodelle mit Hilfe eines anwendungsorientierten Verfahrens zu reduzieren, wird der sog. *Capture-&-Replay*-Ansatz vorgeschlagen. Dabei werden Testdaten für Unittests einzelner Prozeduren aus existierenden Ende-zu-Ende-Tests gewonnen, indem bei deren Ausführung Ein- und ggf. Ausgangsdaten der zu testenden Prozedur aufgezeichnet werden. Die Voraussetzungen und Gestaltungsoptionen für eine anwendungsorientierte Umsetzung dieses Vorgehens im Kontext Klimamodellierung werden diskutiert.

Zudem wird ein Softwarewerkzeug, der *FortranTestGenerator*, vorgestellt, welches auf Grundlage einer statischen Codeanalyse Code zum Aufzeichnen der Testdaten generiert sowie ein Testprogramm, in dem die aufgezeichneten Daten geladen werden, die zu testende Prozedur isoliert ausgeführt und dessen Ergebnisse validiert. Sämtlicher generierter Code basiert dabei auf modifizierbaren Templates und lässt sich so an die Anforderungen der jeweiligen Umgebung und einzelner BenutzerInnen anpassen. Im Rahmen von Experimenten mit vier unterschiedlichen Klimamodellen und einer Benutzerstudie mit drei Klimamodellentwicklern wird die Funktionsfähigkeit, Tauglichkeit und Benutzbarkeit des Werkzeugs demonstriert.

Abstract

Climate models are computer programs for the numerical simulation of climatological processes. They are developed by interdisciplinary teams and consist of numerous submodels representing different components and processes of the climate system. The increasing complexity of these models and the need for adaptation to more and more powerful computer architectures come along with big software engineering challenges. This also effects the testing of such models. Therefore, this doctoral thesis in Informatics looks at the testing of climate models from a software engineering perspective. Specifically, it addresses the question how the creation of unit tests (i.e. isolated tests of smaller code regions on module or procedure level) can be supported in that context.

A qualitative study on the software testing practices at four international climate modeling centers is conducted. One of the main results is that unit tests are used quite rarely in this area. Instead, scientific evaluation as well as regression testing is done mainly with the help of so-called end-to-end tests. For those tests, the main program of the model software is executed, and all phases of a simulation are went through.

Based on the results of this study it is argued that unit tests could be a beneficial addition to the current testing practice. Such tests can help to increase the productivity of the model development provided that the effort for their creation stands in a reasonable relation to their benefits. As an application-oriented mean to reduce the effort for the creation of unit tests for climate models the so-called *Capture & Replay* method is proposed. With this approach, test data for unit tests of individual procedures are derived from existing end-to-end tests by capturing in- and output data of the procedure to be tested during the end-to-end test execution. Requirements and design options of an application-oriented implementation of this approach in the context of climate modeling are discussed.

In addition, a software tool called *FortranTestGenerator* is presented. Based on a static source code analysis, the FortranTestGenerator generates a) the code for the capturing of the test data and b) a test program which loads the captured data, runs the procedure to be tested in isolation, and validates its results. All parts of the generated code are derived from customizable templates and can hence be adapted to requirements of the current environment and individual users. With experiments on four different climate models and a user study with three climate model developers the functioning, suitability, and usability of the tool is demonstrated.

Danksagung

Zahlreiche Menschen haben mich auf meinem Weg zur Fertigstellung dieser Arbeit begleitet und auf die unterschiedlichsten Arten und Weisen zum Gelingen beigetragen. Bei all jenen möchte ich mich bedanken.

Ich danke meinen Betreuern Prof. Thomas Ludwig und Prof. Heinz Züllighoven für ihr Vertrauen, ihre Geduld und ihre Unterstützung. Ich danke den Kolleginnen und Kollegen der Arbeitsbereiche SWA, WR und dem DKRZ für die schöne Zeit, die ich mit ihnen hatte und die vielen anregenden fachlichen und nicht so fachlichen Diskussionen. Speziell danke ich Petra Nerge für die Vermittlung meiner ersten Kontakte zu den KlimaforscherInnen in der Nachbarschaft, Mohammad Reza Heidari („Who is in my office?“) für die tolle gemeinsame Zeit mit all unseren ausschweifenden Diskussionen und Smartboard-Kritzeleien und nicht zuletzt Julian Kunkel für seine Hilfe, die gemeinsamen Projekte und insbesondere auch dem High-Speed-Korrekturlesen zentraler Kapitel dieser Arbeit.

Zudem bedanke ich mich bei allen ICON-Entwicklerinnen und -Entwicklern von MPI-M, DWD und allen anderen Beteiligten, dass sie mich in Ihre Mitte aufgenommen und immer bereitwillig Auskunft über ihre Arbeit gegeben haben. Hervorzuheben ist dabei vor allem Luis Kornblueh, der mich und meine Forschungsarbeit immer unterstützt hat.

Einen großen Beitrag zu dieser Arbeit leisteten auch die Early Adopter des FortranTestGenerators von CSCS, MeteoSchweiz und ETH. Vielen Dank für Euer Vertrauen und euer Feedback. Ein besonderer Dank gilt hier vor allem Will Sawyer, der entscheidende Weichen für Entwicklung und Einsatz des FortranTestGenerators gestellt hat. Vielen Dank auch an die Verantwortlichen der Serialbox2-Bibliothek für die Aufnahme des FTG-Interface in ihren Code.

Bedanken möchte ich mich auch bei Youngsung Kim für den regen Austausch zu KGEN und dem FortranTestGenerator. Diese Diskussionen waren sehr hilfreich, um die Gemeinsamkeiten und Unterschiede der beiden Werkzeuge herauszuarbeiten.

Darüber hinaus gilt ein großer Dank allen Entwicklerinnen und Entwicklern vom GFDL, von JAMSTEC und vom R-CCS, die an meinen Interviews teilgenommen haben. Besonders bedanken möchte ich mich zudem bei alle jenen, die an der Organisation der Interviews beteiligt waren. Vom GFDL ist hier Seth Underwood zu nennen, der meinen gesamten Aufenthalt organisiert hat, sowie V. Balaji, der mein Anliegen bereitwillig unterstützt hat. Dank auch an Tom Clune vom NASA Goddard Space Flight Center für die Vermittlung des Kontakts. Von JAMSTEC seien genannt Hiroaki Tatebe, Kenichi Itakura, Makoto Tsukakoshi, Michio Kawamiya und Tomohiro Hajima. Ein besonderer Dank gilt zudem Hisashi Yashiro vom R-CCS, nicht nur

für die Organisation meines Aufenthalts, sondern auch für all seine Hilfe mit dem NICAM-Modell.

Und nicht zuletzt danke ich natürlich meiner Familie. Ich danke meinen Eltern für alles, was sie für mich getan haben und was letztendlich mit dazu geführt hat, dass ich diese Arbeit schreiben konnte. Der allergrößte Dank geht aber an meine Frau Katrin, dass sie dieses Projekt über Jahre mitgemacht hat, auf vieles verzichten musste, mich immer unterstützt und die Hoffnung auf ein erfolgreiches Ende nie aufgegeben hat.

Inhalt

Kapitel 1

Einleitung

Klimamodelle sind Computerprogramme zur numerischen Simulation klimatologischer Prozesse. Neben den ihnen verwandten numerischen Wettervorhersagemodellen gehören sie zu den ältesten Anwendungen der Computertechnik. Seit ihrem Entstehen in den 1950er Jahren hat sich die Klimamodellierung stetig weiterentwickelt. Anfänglich alleinstehende *Zirkulationsmodelle* von Atmosphäre und Ozean wurden zu gekoppelten Modellen verbunden. Heutige Modelle beinhalten zudem Simulationen der Landoberfläche und ihrer Vegetation, des Meereises sowie biogeochemischer Prozesse wie etwa des CO_2-Kreislaufs. Solche Modelle werden auch *Erdsystemmodelle* genannt (vgl. Lynch, 2008). Weltweit gibt es geschätzt etwa zwanzig bis dreißig Forschungsgruppen, die an derartigen Modellen arbeiten[1]. Hauptzweck dieser Modelle ist die Erforschung von klimatologischen Zusammenhängen, aber auch das Erstellen von Prognosen, etwa um die Entwicklung und die Folgen der globalen Erwärmung vorherzusagen – innerhalb der Grenzen einer inhärenten Unsicherheit (siehe auch IPCC, 2013). Aufgrund des hohen Bedarfs an Rechnerressourcen werden Klimamodelle auf massiv-parallelen Hochleistungsrechnern ausgeführt. Sie sind damit ein typischer Anwendungsbereich des *High-Performance Computings (HPC)*.

Erdsystemmodelle werden von interdisziplinären Teams mit bis über einhundert Beteiligten entwickelt, teilweise in Zusammenarbeit mehrerer Forschungsinstitute, und erreichen Größen von mehreren hunderttausend bis zu mehr als einer Millionen Codezeilen (vgl. Méndez, Tinetti und Overbey, 2014). Der größte Teil der EntwicklerInnen von Klimamodellen verfügt über keine formale Ausbildung in Softwareentwicklung oder Informatik. Um trotz der Größe und Komplexität der Modelle und der Menge an Beteiligten die Entwicklung beherrschbar zu halten, wurden Softwareentwicklungsmethoden und -werkzeuge adaptiert. Einige Methoden und Werkzeuge wurden aus der industriellen Softwareentwicklungspraxis übernommen, wie beispielsweise der Einsatz von Versionskontrolle oder Issue Tracking. In manchen Bereichen haben sich

[1] Eine genaue Zahl ist schwer zu bestimmen, da es sehr viele Modelle unterschiedlicher Komplexität gibt, an denen zum Teil mehrere Forschungseinrichtungen beteiligt sind. Einzelne Forschungsgruppen pflegen wiederum mehrere Modellgenerationen.

aber auch ganz eigene Praktiken entwickelt. Charakterisierend ist unter anderem die konsequente Verwendung der Programmiersprache *Fortran*.

Insbesondere das Testen von Klimamodellen unterscheidet sich von den in der Softwaretechnikliteratur beschriebenen Vorgehensweisen. Dies hat sowohl fachliche, soziokulturelle als auch technische Gründe. Berechnungen von Klimamodellen stellen Approximationen historischer oder fiktiver Szenarien dar, die keine absolute Korrektheit im Sinne einer genauen Lösung erreichen können. Ziel ist es dagegen, mit Hilfe von schrittweisen Verbesserungen Fehlergrößen und Unsicherheiten zu verkleinern. Bei der Bewertung eines Klimamodells steht somit nicht die Korrektheit des jeweiligen Modells im binären Sinne (korrekt/fehlerhaft) im Mittelpunkt, sondern die Güte der Ergebnisse. Um diese Bewertung vornehmen zu können, gibt es verschiedene wissenschaftlich fundierte Methoden, welche jedoch ein hohes Maß an Fachwissen und Detailkenntnisse über das jeweilige Modell verlangen. Änderungen an einem Klimamodell fügen keine neuen Funktionen im Sinne neuer Interaktionsmöglichkeiten für die BenutzerInnen hinzu, welche isoliert getestet werden können, sondern verbessern oder verschlechtern die Ergebnisse bestimmter Modellkonfigurationen bzw. fügen neue Konfigurationsoptionen hinzu. Tests solcher Änderungen bestehen in der Regel aus der Ausführung des Modells in ausgewählten Konfigurationen und der manuellen Bewertung der Ergebnisgüte (vgl. z.B. Easterbrook und Johns, 2009; Guillemot, 2010; Clune und Rood, 2011).

Aus softwaretechnischer Sicht würde man solche Tests als *Systemtests* oder *Ende-zu-Ende-Tests* bezeichnen, da hierbei ein vollständig integriertes Softwaresystem ausgeführt wird, auch wenn, je nach Konfiguration, nicht alle Systemteile bei jedem einzelnen Test beteiligt sind. Das Gegenstück zu dieser Form von High-Level-Tests sind *Unittests*. Darunter versteht man isolierte Tests kleinerer Codeabschnitte, z.B. einzelner Prozeduren (vgl. Fowler, 2014). Diese haben den Vorteil, dass sie sich zum einen schneller kompilieren und ausführen lassen als Systemtests, dadurch öfter ausgeführt werden können und so den EntwicklerInnen ein schnelleres und häufigeres Feedback über ihre Arbeit geben können. Zum anderen erleichtern sie durch ihre Fokussierung auf kleine Bereiche des Codes die Fehlerlokalisierung.

1.1 Fragestellungen und Ziel dieser Arbeit

Diese Arbeit leistet einen Beitrag zu *Softwaretestverfahren* mit Fokus auf Anwendungen in der Klimamodellierung. Als Softwaretest wird dabei ein Test eines Computerprogramms im softwaretechnischen Sinne verstanden, d.h. das Ausführen des Programms mit dem Ziel Fehler aufzudecken (vgl. Myers, Sandler und Badgett, 2011). Im Kontext Klimamodellierung sind hiervon insbesondere Tests zur wissenschaftlichen Evaluation eines Modells abzugrenzen. Dabei wird ein mutmaßlich fehlerfreies

Modell ausgeführt, um anschließend die Güte der Ergebnisse zu bewerten. Nicht jede Testaktivität lässt sich eindeutig der einen oder anderen Kategorie zuordnen; eine Ausführung eines Modells kann sowohl dem Aufdecken von Fehlern dienen und in dem Fall, dass keine Fehler erkannt wurden, können ihre Ergebnisse wissenschaftlich ausgewertet werden. Dennoch stehen für beide Testformen spezifische Methoden bereit: softwaretechnische Methoden für die eine, klimawissenschaftliche Methoden für die andere. Der Fokus dieser Arbeit liegt auf softwaretechnischen Testmethoden.

Hierzu wird der Gegenstandsbereich zunächst anhand folgender Forschungsfragen untersucht:

1. Was charakterisiert die Praxis des Softwaretestens in der Klimamodellierung?
2. Welche Defizite lassen sich dabei aus softwaretechnischer Sicht identifizieren?

Ein wesentliches Ergebnis dieser Untersuchung ist die Feststellung, dass EntwicklerInnen von Klimamodellen nur sehr selten Unittests einsetzen, wodurch die Produktivität des Softwareentwicklungsprozess leidet. Daraus ergeben sich die Anschlussfragen:

3. Was hält EntwicklerInnen von Klimamodellen vom Einsatz von Unittests ab?
4. Wie können EntwicklerInnen von Klimamodellen beim Erstellen von Unittests unterstützt werden?

Die Beantwortung der Frage 4 stellt den Hauptbeitrag dieser Arbeit dar. Hierzu wird eine mögliche Lösung in Form eines Softwarewerkzeugs, des *FortranTestGenerators*, zur unterstützten Erstellung von Unittests für einzelne Prozeduren von Klimamodellen vorgeschlagen. Diese basiert auf dem sog. *Capture-&-Replay*-Ansatz, dabei werden Eingabedaten aufgezeichnet, mit denen eine zu testende Prozedur innerhalb der Originalanwendung aufgerufen wird, beispielsweise im Rahmen von Ende-zu-Ende-Tests. Die aufgezeichneten Daten werden dann für die isolierte Ausführung der Prozedur innerhalb von Unittests verwendet.

Die Formulierung der Forschungsfrage macht bereits deutlich, dass hierbei keine rein technische oder theoretisch-informatische Sichtweise eingenommen wird, sondern eine *anwendungsorientierte*. Heinz Züllighoven, Dirk Bäumer u. a. (1998, S. 261) definieren Anwendungsorientierung wie folgt:

Definition 1.1: Anwendungsorientierung

„Anwendungsorientierung bei der Softwareentwicklung zielt auf folgende Merkmale der Software ab:

- *Die Funktionalität des Softwaresystems orientiert sich an den Aufgaben im Anwendungsbereich.*
- *Die Handhabung des Softwaresystems ist benutzergerecht.*
- *Die im System festgelegten Abläufe und Schritte lassen sich je nach Anwendungssituation problemlos den tatsächlichen Erfordernissen anpassen.“*

Ziel ist es, im Anwendungsbereich Klimamodellierung tätige EntwicklerInnen bei der Aufgabe Unittesterstellung zu unterstützen. Bei der Konstruktion der Lösung sind somit der Anwendungsbereich, die AnwenderInnen und die Aufgabe zu berücksichtigen. Hieraus ergeben sich sowohl konzeptionelle und technische Randbedingungen als auch Anforderungen, die die Handhabung der Softwarelösung betreffen.

1.2 Hintergrund

Diese Arbeit entstand im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Fachbereich Informatik der Universität Hamburg. Ich begann diese Tätigkeit als Mitglied des Arbeitsbereichs *Softwaretechnik und -architektur (SWA)* von Prof. Heinz Züllighoven. Da softwaretechnische Methoden in der Klimamodellierung im Mittelpunkt meiner Arbeit stehen sollten, begannen wir eine Kooperation mit dem Arbeitsbereich *Wissenschaftliches Rechnen (WR)* von Prof. Thomas Ludwig. Diese Gruppe ist am *Deutschen Klimarechenzentrum (DKRZ)* ansässig, dessen Direktor Prof. Ludwig ist. Das DKRZ betreibt Hochleistungsrechner und Datenspeicher und stellt diese ForscherInnen aus Deutschland für Klimasimulationen zur Verfügung. Nach der Pensionierung von Prof. Züllighoven und der damit verbundenen Auflösung von SWA wechselte ich vollständig zu WR und damit räumlich zum DKRZ.

Durch die Zusammenarbeit mit Kolleginnen und Kollegen von WR entstanden auch Kontakte zum *Max-Planck-Institut für Meteorologie (MPI-M)*, wo in Kooperation mit dem *Deutschen Wetterdienst (DWD)* das Klimamodell ICON (vgl. Zängl u. a., 2015) entwickelt wird. Hieraus ergab sich die Möglichkeit die Vorgehensweisen bei der Entwicklung von ICON aus softwaretechnischer Sicht zu untersuchen. Im Laufe dieser Untersuchungen grenzte ich meinen Forschungsschwerpunkt auf Methoden und Vorgehensweisen beim Testen des Modells ein. Neben dem MPI-M und dem

DWD sind noch weitere Forschungseinrichtungen an der Entwicklung von ICON beteiligt. Erwähnt sei hier das *Swiss National Supercomputing Centre (Centro Svizzero di Calcolo Scientifico, CSCS)*. Eine Gruppe des CSCS arbeitet an der Portierung von ICON auf GPUs. Aus der Diskussion mit den Beteiligten darüber, wie sich für diese Portierung Test generieren ließen, um zu überprüfen, ob die Ausführung auf GPUs dieselben (bzw. ausreichend ähnliche) Ergebnisse liefert wie die Ausführung auf herkömmlichen CPUs, entstand letztendlich die Idee für das Hauptthema dieser Arbeit, der unterstützten Erstellung von Unittests für Klimamodelle.

Meine am ICON-Projekt begonnene Studie über Testpraktiken in der Klimamodellierung konnte ich durch Besuche am *Geophysical Fluid Dynamics Laboratory (GFDL)* in den USA sowie bei der *Japan Agency for Marine Earth Science and Technology (JAMSTEC)* und am *RIKEN Center for Computational Science (R-CCS)* in Japan ergänzen.

In diese Arbeit flossen auch Erkenntnisse aus der Betreuung von Studienarbeiten im Rahmen meiner Tätigkeit an der Universität Hamburg ein. Genannt seien hier vor allem die Projektarbeit von Marcel Heing-Becker (2017) sowie die Masterarbeit von Tareq Kellyeh (2018).

1.3 Methodik

Diese Arbeit orientiert sich an der *Design-Science-Methode* in der Beschreibung von Roel J. Wieringa (2014). Wieringa definiert Design Science wie folgt:

Definition 1.2: Design Science

„Design science is the design and investigation of artifacts in context. The artifacts we study are designed to interact with a problem context in order to improve something in that context.“

Im Mittelpunkt eines Design-Science-Projekts stehen somit ein oder mehrere *Artefakte* und die Tätigkeiten *Konstruktion (design)* und *Erforschung (investigation)*. Ein Artefakt ist etwas, was von Menschen für einen praktischen Zweck geschaffen wird. Als Beispiele aus der Softwaretechnik nennt Wieringa (2014, S. 29) Algorithmen, Methoden, Notationen, Techniken oder konzeptuelle Rahmenwerke. Die Artefakte, die im Mittelpunkt dieser Arbeit stehen, sind der Capture-&-Replay-Ansatz und dessen Umsetzung in Form des FortranTestGenerators.

Ein Design-Science-Projekt befasst sich mit *Konstruktionsproblemen (design problems)* und *Erkenntnisfragen (knowledge questions)* (vgl. Wieringa, 2014, S. 4f). Das

Konstruktionsproblem dieser Arbeit leitet sich aus der Forschungsfrage 4 ab und lässt sich im Sinne Wieringas auch wie folgt formulieren:

> Konstruiere für EntwicklerInnen von Klimamodellen eine Unterstützung für das Erstellen von Unittests, um die Produktivität des Entwicklungsprozesses zu erhöhen.

Forschungsfrage 1–3 sind Erkenntnisfragen im Sinne Wieringas. Sie dienen der Erforschung des Problemkontexts und führen zur Formulierung des Konstruktionsproblems. Die Beantwortung dieser Fragen stützt sich dabei auf folgende, vornehmlich qualitativer Forschungsmethoden:

- Literaturrecherche
- Beobachtungen in der Fallstudie ICON mit Hilfe semistrukturierter Interviews, der Teilnahme an Entwicklertreffen und Workshops sowie einer Umfrage unter EntwicklerInnen
- Semistrukturierte Interviews mit Entwicklerinnen und Entwicklern weiterer Klimamodelle
- Begutachtungen von Quellcode und Dokumentationen

Durch die Konstruktion des Artefakts ergeben sich neue Erkenntnisfragen, insbesondere zur Adäquatheit der Lösung. Die Lösung des Konstruktionsproblems ist daher ein iterativer Prozess, bei der sich Konstruktion und Erforschung bzw. Erprobung des Artefakts im Problemkontext abwechseln. Der Problemkontext dieser Arbeit beinhaltet sowohl technische als auch anwendungsorientierte Aspekte. Die technischen Aspekte des Problemkontexts werden im Wesentlichen durch die Zielanwendungen Klimamodelle und deren Charakteristika definiert, aber auch durch das weitere technologische Umfeld, wie etwa die Rechnerplattformen, auf denen die Modelle in der Regel entwickelt und ausgeführt werden. Die anwendungsorientierten Aspekte ergeben sich aus den potenziellen BenutzerInnen des entwickelten Softwarewerkzeugs und deren Aufgaben.

Obwohl die anwendungsorientierten Aspekte bei der Konstruktion der Lösung eine wesentliche Rolle spielen, widmet sich ein Großteil der Erprobung des Capture-&-Replay-Ansatzes bzw. des FortranTestGenerator technischen Aspekten. Insbesondere wurden diese während der Entwicklung kontinuierlich mit Hilfe von Beispielen aus dem ICON-Modell erprobt. Darüber hinaus wurden Experimente zur Funktionsfähigkeit mit anderen Modellen durchgeführt. Die Nützlichkeit der erzeugten Unittests wird ebenfalls anhand von Fallbeispielen demonstriert.

Der Fokus auf technische Aspekte bei der Erprobung der Artefakte hat zwei Gründe: Zum einen ist die technische Funktionsfähigkeit die Voraussetzung, um anwendungsorientierte Aspekte untersuchen zu können, zum anderen lässt sich die Angemessenheit der Lösung im anwendungsorientierten Sinne nur subjektiv bewerten. Daher kann diese nur statistisch erforscht werden; was den Einsatz der konstruierten Artefakte durch eine statistisch relevante Zahl von AnwenderInnen notwendig macht. Dieser war im Rahmen dieser Arbeit nicht möglich. Jedoch wurde der FortranTest-Generator zur Erzeugung von Unittests in einem kleinen realen Projekt mit drei Anwendern eingesetzt und diese anschließend mit Hilfe eines Fragebogens zu ihren Erfahrungen mit dem Werkzeug befragt. Darüber hinaus flossen bereits während des Projekts Rückmeldungen der Anwender in die Weiterentwicklung des FortranTest-Generators mit ein.

1.4 Einordnung

Wie jede wissenschaftliche Arbeit entstehen auch Design-Science-Projekte nicht „im luftleeren Raum“. Sie nutzen existierendes Wissen aus Wissenschaft und Technik und erzeugen neues Wissen durch die Konstruktion von Artefakten und die Beantwortung von Erkenntnisfragen. Wieringa (2014) spricht in diesem Zusammenhang von *Wissenskontext (knowledge context)*. Im Folgenden werden die wichtigsten Themenbereiche des Wissenskontexts dieser Arbeit vorgestellt und die Arbeit darin eingeordnet.

1.4.1 Klimamodellierung

Die vom Menschen verursachte globale Erwärmung gehört zu den größten Herausforderungen der Gegenwart. Das Verstehen klimatologischer Prozesse und Wechselwirkungen ist eine wichtige Voraussetzung für den Umgang mit dieser. Hierbei spielen Modelle eine entscheidende Rolle – sowohl bei der Erforschung des Klimasystems als auch bei der Vorhersage zukünftiger Entwicklungen (siehe auch IPCC, 2013). Der Begriff Klimamodell wird in dieser Arbeit wie folgt verwendet:

Definition 1.3: Klimamodell

Computerprogramm zur numerischen Simulation klimatologischer Prozesse.

Ein solches Computermodell basiert auf einer Hierarchie von Modellen der physikalischen Realität. Abbildung 1.1 zeigt ein vereinfachtes Schema dieser Modellhierarchie sowie die Prozesse, die von einer Modellebene zur anderen führen. Diese Arbeit

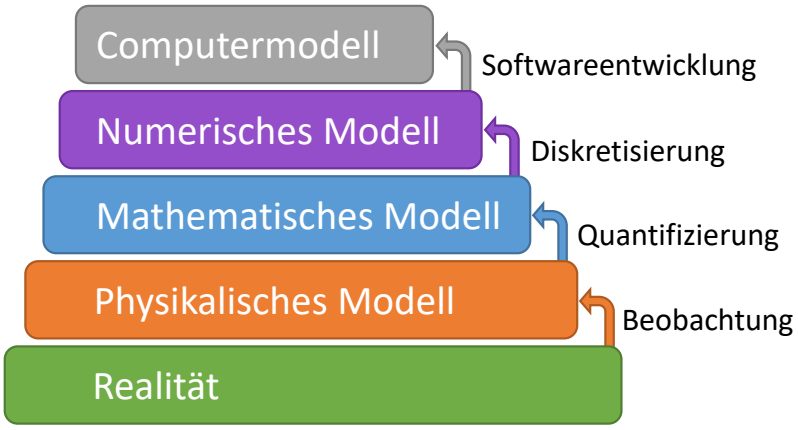


Abbildung 1.1: Modellhierarchie

beschäftigt sich mit der obersten Ebene dieser Pyramide und dem Softwareentwicklungsprozess, der zur Erstellung dieser Modelle führt. Der Fokus geowissenschaftlicher Literatur zur Klimamodellierung liegt dagegen meist auf der numerischen und der mathematischen Ebene. Die Beschreibung einzelner Klimamodelle, bei denen es sich eigentlich um Computermodelle handelt, beschränkt sich weitestgehend auf die zugrundeliegenden numerischen Verfahren (siehe z.B. Satoh, Matsuno u. a., 2008; Donner u. a., 2011; Watanabe u. a., 2011; Zängl u. a., 2015). Dabei findet häufig eine Gleichsetzung dieser beiden Ebenen statt, was zu einer Ausblendung des Softwareentwicklungsprozesses führt, obwohl dieser für die Erstellung des Computermodells von herausragender Bedeutung ist und den mutmaßlich größten Teil des Arbeitsaufwands im gesamten Modellierungsprozess in Anspruch nimmt. So lässt sich beispielsweise feststellen, dass im letzten Bericht des *Intergovernmental Panel on Climate Change (IPCC)* bzw. in dem 1.552 Seiten starken Teil des Berichts, welcher sich mit der wissenschaftlichen Basis der Erkenntnisse befasst (IPCC, 2013), im Fließtext nicht ein einziges Mal das Wort „Software“ vorkommt[2].

Es gibt jedoch auch Ausnahmen, wie etwa die Beschreibung des Softwaredesigns des Community Climate System Model von Drake, Jones und Carr (2005), welche ebenfalls einen Abschnitt über den Entwicklungsprozess enthält. In jüngerer Zeit kamen weitere Arbeiten hinzu, wie etwa die Beschreibung des NICAM-Modells von Satoh, Tomita u. a. (2014), welche auch einen Abschnitt zu softwaretechnischen Aspekten enthält. Zu erwähnen sei auch die fünfbändige Buch-Reihe „Earth System Modelling“ (Bonaventura, Redler und Budich, 2012; Ford u. a., 2012; Valcke, Redler und Budich, 2012; Balaji, Redler und Budich, 2013; Puri, Redler und Budich, 2013), welche sich mit verschiedenen technische Aspekten der Klimamodellierung beschäftigt.

[2]Im Literaturverzeichnis taucht das Wort „Software“ allerdings dreimal auf und in der Liste der Autoren einmal.

Wichtige empirische Arbeiten zur Klimamodellierung aus softwaretechnischer Sicht stammen von Steve Easterbrook. Dieser hat zusammen mit anderen Autoren beispielsweise den Softwareentwicklungsprozess am britischen MetOffice untersucht (Easterbrook und Johns, 2009), die Fehlerdichte verschiedener Modelle analysiert (Pipitone und Easterbrook, 2012) oder Softwarearchitekturen beschrieben (Alexander und Easterbrook, 2015). Zudem betreibt er einen Blog, welcher verschiedene Beiträge zum Thema enthält (Easterbrook, 2009).

Ein Abschnitt in Easterbrook und Johns, 2009 befasst sich auch mit dem Aspekt des Testens von Klimamodellen. Dieser steht auch bei Clune, Finkel und Rilee (2015) im Mittelpunkt. Deren Beschreibungen entstammen jedoch eher persönlichen Erfahrungen und weniger einer systematischen, empirischen Untersuchung.

Aufgrund der gesellschaftlichen Bedeutung der Klimamodellierung und dem Stellenwert, die die Softwareentwicklung bei der Erstellung der Modelle einnimmt, lohnt es sich, an diese Arbeiten anzuknüpfen. Hinzu kommt, dass weltweit eine Vielzahl von Forschungseinrichtungen in der Klimamodellierung tätig sind, welche sich mit ähnlichen Problemen beschäftigen. Infolge der langen Tradition dieses Zweigs der computergestützten Wissenschaften und des regen Austausches zwischen einzelnen Instituten, sowohl auf wissenschaftlicher und personeller als auch auf Softwareebene, existieren in unterschiedlichen Forschungsgruppen ähnliche Softwarestrukturen und Vorgehensweisen (vgl. Edwards, 2010, S. 167ff). Hieraus ergibt sich eine gewisse Generalisierbarkeit der Erkenntnisse, die in den im Rahmen dieser Arbeit untersuchten Organisationen gewonnen wurden.

Einige der Erkenntnisse dieser Arbeit sowie die vorgeschlagene Lösung zur unterstützten Erstellung von Unittests lassen sich mutmaßlich auch auf andere Bereiche der computergestützten Wissenschaften, des High-Performance Computings sowie der Fortran-Anwendungsentwicklung im Allgemeinen übertragen. Aufgrund des anwendungsorientierten Ansatzes konzentriert sich diese Arbeit jedoch bewusst auf einen konkreten Anwendungsbereich, die Klimamodellierung. Allerdings spielen die Eigenschaften von Klimamodellen, die sich aus der Zugehörigkeit zu den genannten Anwendungsklassen (wissenschaftliche Software, HPC, Fortran) ergeben, eine wichtige Rolle für diese Arbeit. So bestimmen die Eigenschaften der Sprache Fortran offenkundig die Konstruktion des FortranTestGenerators. Auch die Tatsache, dass es sich bei den EntwicklerInnen um WissenschaftlerInnen ihrer jeweiligen Disziplin und nicht um ausgebildete InformatikerInnen oder SoftwareentwicklerInnen handelt, beeinflusst maßgeblich den Entwurf der Lösung; ebenso der Umstand, dass Klimamodelle parallele Anwendungen für die Ausführung auf modernen Hochleistungsrechnern sind. Somit leistet diese Arbeit auch einen Beitrag für die softwaretechnische Forschung in diesen Bereichen.

1.4.2 Softwaretests

Diese Arbeit beschäftigt sich mit Softwaretests im Sinne der Definition von Myers, Sandler und Badgett (2011, S. 6):

Definition 1.4: Softwaretest

„Testing is the process of executing a program with the intent of finding errors.“

Beim Testen geht es somit darum, Fehler zu finden und nicht darum, deren Abwesenheit bzw. die Korrektheit eines Softwaresystems nachzuweisen. Edsger W. Dijkstra (1972) formulierte es so:

> *„Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.“*

Durch systematisches Testen und das Beheben der dadurch entdeckten Fehler erhöht sich jedoch das Vertrauen in die korrekte Funktionsweise eines Systems. Neben dem Testen neuer Funktionen spielt in der Softwareentwicklung aus das Regressionstesten eine wichtige Rolle. Dazu werden Tests bereits existierender Funktionen erneut ausgeführt, sobald eine Änderung am Softwaresystem vorgenommen wird. Ziel ist es, unerwünschte Nebenwirkungen und Fehler, die durch die Änderungen in das System eingebracht wurden, aufzudecken.

Aufgrund der hohen Bedeutung des Testens in der Softwareentwicklung, ist dieses auch ein wichtiger Gegenstand softwaretechnischer Forschung mit zahlreichen Beiträgen. Viele Arbeiten beschäftigen sich dabei mit der Automatisierung des Testens sowie der automatischen Erzeugung von Tests bzw. von Testdaten, insbesondere auch für Regressionstests. Häufig stehen dabei spezielle Anwendungsklassen im Mittelpunkt, wie z.B. Webservices, grafische Benutzeroberflächen oder Cloud-Anwendungen (vgl. Garousi und Mäntylä, 2016). Auch zum Testen wissenschaftlicher Anwendungen existieren einige Arbeiten (vgl. Kanewala und Bieman, 2014). Diese Arbeit reiht sich in diesen Forschungszweig ein, indem sie sich mit Möglichkeiten der Automatisierung bei der Unittesterstellung für Klimamodelle auseinandersetzt.

Das Testen wissenschaftlicher Software wie etwa von Klimamodellen gilt allgemein als schwierig. Als eines der Hauptprobleme wird das häufige Fehlen von Testorakeln genannt (vgl. Kanewala und Bieman, 2014).

Definition 1.5: Testorakel

Mechanismus oder eine Informationsquelle, der bzw. die dazu verwendet wird, zu entscheiden, ob das Verhalten eines zu testenden Programms in einem Test erwartungskonform ist oder nicht.

Testorakel geben die Erwartungen an ein Programm vor, gegen die dieses getestet wird. Dies können Spezifikationen sein oder analytische Lösungen von mathematischen Problemen, die ein Programm numerisch löst, oder sich aus dem Wissen der EntwicklerIn oder der TesterIn ergeben. In wissenschaftlichen Anwendungen, die dazu dienen neue Erkenntnisse zu gewinnen, stehen derartige Testorakel häufig nicht zur Verfügung. Dies betrifft auch Berechnungen von Klimamodellen.

Jedoch dienen nicht alle Änderungen an Klimamodellen der Erzeugung neuer Erkenntnisse. Sehr häufig unterliegen sie zum Beispiel nichtfunktionalen Änderungen, wie etwa Refactorings, Optimierungen, Portierungen auf andere Rechnerarchitekturen oder Veränderungen der Softwareinfrastruktur. Solche Änderungen sollen die Modellergebnisse nicht verfälschen. Auch bei Einführung neuer Konfigurationsoptionen sollen in der Regel die Ergebnisse bestehender Modellkonfigurationen unverändert bleiben. Als Testorakel können hier somit Vorversionen des Modells dienen. Im Rahmen von Regressionstests lassen sich solche Änderungen daher auch automatisiert testen, indem die Modellergebnisse mit denen einer Vorversion verglichen werden.

Hierbei wäre auch der Einsatz kleinteiliger, codenaher Unittests möglich. Vorläufig gelte hierfür die folgende Definition:

Definition 1.6: Unittest

Automatisierter Test eines kleinen Codeabschnitts einer größeren Anwendung.

Unter einem „kleinen Codeabschnitt“ kann z.B. ein einzelnes Modul oder eine einzelne Prozedur verstanden werden, wobei sich diese Arbeit auf Tests einzelner Prozeduren konzentriert. Unittests haben den Vorteil, dass sie sich zum einen schneller kompilieren und ausführen lassen und zum anderen das Lokalisieren von Fehlerursachen vereinfachen. Durch die Fokussierung auf kleine Teilbereiche des Modellcodes lassen sich ihre Funktionsweise und ihre Ergebnisse zudem mit weniger Fachwissen und Detailkenntnissen über das Gesamtmodell verstehen. Durch neue technische Herausforderung, wie die immer größere Zahl paralleler Rechnerknoten in modernen Hochleistungsrechner oder vermehrt heterogene Rechnerarchitekturen und die damit verbundene stärkere Zusammenarbeit von KlimaforscherInnen mit fachfremden IT-ExpertInnen verstärkt sich der Bedarf nach derartigen Tests. In der Praxis stehen sie

jedoch häufig nicht zu Verfügung bzw. werden bei Bedarf auch nicht erstellt. Diese Arbeit soll einen Beitrag dazu leisten die Erstellung von Unittests zu fördern.

1.4.3 Capture & Replay

Ein großer Teil des Aufwands beim Softwaretesten besteht in der Erzeugung von Testfällen und den dazugehörigen Testdaten. Die Automatisierung der Testfallerzeugung ist daher eines der zentralen Themen der Softwaretestforschung. Verschiedene Verfahren und Werkzeuge sind dabei entstanden (vgl. Anand u. a., 2013). Testfälle können beispielsweise automatisiert erzeugt werden, indem auf Grundlage einer Analyse des zu testenden Codes gezielt Testdaten erzeugt werden, die zu einer hohen Codeabdeckung führen (sog. *White-Box-Verfahren*). Ein anderer Ansatz ist es, Testdaten auf Basis von formalen Beschreibungen des zu testenden Programms und seiner Spezifikationen zu erzeugen (*modellbasiertes Testen*).

Beim Capture & Replay werden Testdaten für Unittests kleiner Codeabschnitte, wie etwa einzelner Prozeduren, durch Ausführung der Prozedur im Kontext der bestehenden Anwendung (Originalanwendung) erzeugt. Die Eingabedaten, mit denen die Prozedur innerhalb der Originalanwendung ausgeführt wird, werden aufgezeichnet und anschließend als Testdaten für die isolierte Ausführung der Prozedur im Rahmen des Unittests verwendet. Als Grundlage können hier sowohl Produktiveinsätze der Originalanwendung als auch existierende System- oder Ende-zu-Ende-Tests dienen. Da sich die bestehende Testpraxis in der Klimamodellierung sehr stark auf Ende-zu-Ende-Tests stützt, bietet sich dieses Vorgehen hier besonders an.

Der Begriff Capture & Replay entstammt ursprünglich dem Testen graphischer Benutzeroberflächen. Dabei geht es meist nicht um das Aufzeichnen von Eingabedaten einzelner Prozeduren, sondern um das Aufzeichnen einer Serie von Benutzeraktionen, welche anschließend automatisiert wiederholt werden kann (vgl. Beizer, 1990, S. 452f). Der Begriff wird jedoch auch für codenahe Ansätze der Datenaufzeichnung und -wiederverwendung verwendet (siehe z.B. Silverstein, 2003; Orso und Kennedy, 2005; Castro u. a., 2015).

Mehrere Arbeiten verwenden bereits diesen Ansatz, um Testdaten zu erzeugen, zum Teil unter unterschiedlichen Bezeichnungen für dieselbe Methode. So leiten beispielsweise Elbaum u. a. (2009) auf diese Weise Unittests aus Systemtests für Java-Systeme ab. Sie bezeichnen dieses Vorgehen wiederum als „Carving and Replaying“. In dieser Arbeit wird dieser Ansatz auf große Fortran-Systeme übertragen und untersucht wie eine anwendungsorientierte Gestaltung im Kontext Klimamodellierung aussehen kann.

Eine ähnliche Arbeit entstand gleichzeitig zu dieser am *National Center for Atmospheric Research (NCAR)* in den USA. Kim u. a. (2016) haben dort das Softwarewerkzeug *KGEN* entwickelt, welches ebenfalls den Capture-&-Replay-Ansatz verwendet und auf Klimamodelle in Fortran abzielt. Umsetzung und Handhabung unterscheiden sich jedoch vom FortranTestGenerator.

1.4.4 Anwendungsorientierung und Werkzeugbegriff

Auf der einen Seite befasst sich die Klimaforschungsliteratur mit den wissenschaftlichen Fragestellungen, zu deren Zwecke Klimamodelle entwickelt werden und beschreibt die Produkte eines Softwareentwicklungsprozesses, der dabei weitgehend verborgen bleibt. Auf der anderen Seite fokussiert sich die Softwaretestliteratur auf die Mittel, die bei der Softwareentwicklung zum Testen eingesetzt werden können und abstrahiert dabei von den Besonderheiten und spezifischen Anforderungen einzelner Anwendungsklassen. Der anwendungsorientierte Blick dieser Arbeit soll eine Brücke zwischen diesen beiden Sichtweisen bilden, indem sie bei der Untersuchung von Softwaretestmethoden den Anwendungskontext Klimamodellierung berücksichtigt und für diesen Anwendungsbereich eine Methode vorschlägt und ein entsprechendes Softwarewerkzeug konstruiert.

Dabei wird der Begriff der Anwendungsorientierung im Sinne der Definition 1.1 nach Züllighoven, Bäumer u. a. (1998) verwendet. Das Ziel anwendungsorientierter Softwareentwicklung (application-oriented software development) ist es, nützliche und nutzbare Software für professionelle AnwenderInnen zu erstellen, welche sich an den Aufgaben der AnwenderInnen und ihrem täglichen Umgang mit diesen orientiert.

> *„IT usage and software development have to be seen as means to an end which means providing professional users with useful and usable software so that the[y] can offer adequate services to their customers. This way of looking at IT and software development is what we call application-orientation.*
> *Application-orientation means that analysis, design and construction of software is firmly based on the tasks and the way of dealing with them in everyday work situations. Understanding the tasks at hand and the concepts behind them is the main challenge for software developers. Application software, therefore, should reflect and represent the core concepts and the familiar objects and means of work of the application area.“*
> (Züllighoven, Gryczan u. a., 1999)

Die Autoren zielen mit ihren Ausführungen zu anwendungsorientierter Software, welche im sog. Werkzeug-und-Material-Ansatz münden, vornehmlich auf interaktive, graphische Arbeitsplatzanwendungen für komplexe Expertentätigkeit. Diese grenzen

sie explizit von der großrechnerbasierten Datenverarbeitung und von kommandozeilenbasierten Anwendungen ab, zu welchen auch der in dieser Arbeit entwickelte FortranTestGenerator gehört. Die in Definition 1.1 genannten Anforderungen sind jedoch allgemein formuliert und lassen sich auch auf derartige Anwendungen übertragen.

Der in dieser Arbeit entwickelte FortranTestGenerator wurde vorangehend und wird auch im Folgenden als „Softwarewerkzeug“ bezeichnet. Die Verwendung dieses Begriffs stützt sich dabei auf die Definition von *Software Tool* nach Knut Hildebrand (1990):

Definition 1.7: Software Tool

„Software Tools sind Werkzeuge aus Software für die Entwicklung von Software.“

Werkzeug definiert Hildebrand (1990) wiederum folgendermaßen:

Definition 1.8: Werkzeug

„Ein Werkzeug ist ein ganz oder teilweise automatisiertes Verfahren (bzw. Methode).“

Es erscheint zunächst etwas verwirrend, dass in der vorliegenden Arbeit der Begriff „Softwarewerkzeug“ synonym für sein englisches Pendant „Software Tool“ verwendet wird, welcher wiederum über denselben deutschen Begriff definiert wird. Die Bevorzugung des deutschen gegenüber dem englischen Wort erfolgt dabei lediglich aus stilistischen Gründen. Belässt man es bei dieser Übersetzung und setzt die eine Definition in die andere ein, löst sich der Widerspruch auf:

Definition 1.9: Softwarewerkzeug

„Softwarewerkzeuge sind ganz oder teilweise automatisierte Verfahren (bzw. Methoden) aus Software für die Entwicklung von Software.“

Nach meiner subjektiven Einschätzung entspricht diese Definition der in der Softwaretechnik etablierten Verwendung der Begriffe (Software-)werkzeug bzw. Tool (siehe z.B. Xie, 2012; Zoller und Schmolitzky, 2012; Méndez, Tinetti und Overbey, 2014; Kim u. a., 2016). Da diese darüber hinaus auch in anderen Bereichen verwendet werden, ließe sich die Definition auch allgemeiner formulieren, indem man auf die Spezifizierung „für die Entwicklung von Software“ verzichtet. Da sich die vorliegende Arbeit aber gerade mit der Entwicklung von Software beschäftigt, ergibt sich hieraus keine Beschränkung.

Hervorzuheben in der Definition nach Hildebrand ist die Automatisierung, welche auch in der vorliegenden Arbeit eine wichtige Rolle spielt. Der Begriff des Werkzeugs nimmt auch in dem Werk von Züllighoven (1998), aus dem die Definition 1.1 übernommen ist[3], eine zentrale Position ein. Hier liegt der Fokus jedoch weniger auf der Automatisierung als mehr auf der Handhabung:

> *„Werkzeuge sind Gegenstände, mit denen Menschen im Rahmen einer Aufgabe Materialien verändern oder sondieren können.*
> *Werkzeuge eignen sich meist für verschiedene fachliche Zwecke und für die Arbeit an unterschiedlichen Materialien. Sie müssen geeignet gehandhabt werden.*
> *Werkzeuge vergegenständlichen wiederkehrende Arbeitshandlungen.“*
> (Züllighoven und Gryczan, 1998, S. 177)

Davon abgegrenzt wird der Begriff des *Automaten*:

> *„Automaten sind im Rahmen einer zur erledigenden Aufgabe ein Arbeitsmittel, um Material zu bearbeiten. Sie erledigen lästige Routinetätigkeiten als eine definierte Folge von Arbeitsschritten mit festem Ergebnis ohne weitere äußere Eingriffe.*
> *Automaten laufen unauffällig im Hintergrund, wenn sie einmal vom Benutzer oder von der Arbeitsumgebung gestartet sind.*
> *Sie können auf ihren Zustand überprüft und im vorgegebenen Rahmen eingestellt werden.“* (Züllighoven und Gryczan, 1998, S. 187)

Die Begriffe werden als Metaphern für den Entwurf interaktiver, grafischer Anwendungssysteme verwendet und müssen in diesem Kontext verstanden werden. Jedoch basieren diese Metaphern auf der Dissertation von Reinhard Budde und Heinz Züllighoven (1990), in der der Werkzeugbegriff erkenntnistheoretisch diskutiert wird. Im Mittelpunkt stehen wie bei Hildebrand dabei Werkzeuge für die Softwareentwicklung. Auch wenn die Ausführungen von Budde und Züllighoven letztendlich in einer Konstruktionsanleitung für ebenfalls interaktive, graphische Programmierumgebungen münden, wird der Begriff Softwarewerkzeug zunächst allgemeiner definiert und schließt auch typische Kommandozeilenprogramme mit ein:

> *„Software-Werkzeuge dienen entsprechend zur Bearbeitung der verschiedenen Formen des Programmiermaterials. Compiler, Editoren, browser und pretty-printer sind Software-Werkzeuge.*

[3] Bei Züllighoven, 1998 handelt es sich um ein Buch mit mehreren Autoren, dessen Hauptautor und Herausgeber Heinz Züllighoven ist. Lassen sich hier referenzierte Inhalte einzelnen Kapitel zuordnen, werden diese separat ausgewiesen, um allen Autoren der jeweiligen Kapitel gerecht zu werden. So etwa die Definition 1.1, als dessen Quelle das Kapitel Züllighoven, Bäumer u. a., 1998 angegeben wurde.

> [...]
> *Unter den Programmier-Hilfsmitteln sind Software-Werkzeuge derjenige Teil, der für das Besorgen und unser Verständnis davon besonders wichtig ist. Mit Software-Werkzeugen wirken wir auf das Programmier-Material ein, wir organisieren es und informieren uns über seinen Zustand.“* (Budde und Züllighoven, 1990, S. 161)

Automaten werden dagegen folgendermaßen definiert:

> *„Softwaresysteme, die sich im Umgang als undurchschaubare Maschinen darstellen, nennen wir Automaten.“* (Budde und Züllighoven, 1990, S. 147)

Auch wenn Automatisierung ein wichtiger Baustein der vorliegenden Arbeit ist, um den Prozess der Unittesterstellung zu unterstützen und den Arbeitsaufwand für die EntwicklerInnen zu reduzieren, soll es dem anwendungsorientierten Anspruch folgend nicht Ziel sein, eine „undurchschaubare Maschine“ zu konstruieren. Dennoch wird zum Schluss zu diskutieren sein, welche Eigenschaften des FortranTestGenerators in Buddes und Züllighovens Sinne eher automaten- und welche eher werkzeugartig sind. Trotzdem soll nicht darauf verzichtet werden, ihn zunächst als Softwarewerkzeug im etablierten Sinn des Wortes zu bezeichnen.

1.5 Aufbau der Arbeit

Die Arbeit ist folgendermaßen aufgebaut; in eckigen Klammern ist jeweils angeben, ob die Inhalte des jeweiligen Kapitels aus eigenen wissenschaftlichen Beiträgen bestehen oder überwiegend der Literatur entnommen sind:

- **Kapitel 2** führt in den Gegenstandsbereich Klimamodellierung ein. Dabei werden fachliche, sozio-kulturelle wie auch technologische Aspekte diskutiert, die für diese Arbeit relevant sind. [Literatur]
- In **Kapitel 3** werden Grundbegriffe des Softwaretestens definiert. Insbesondere wird hier auch der Begriff Unittest diskutiert. [Literatur]
- **Kapitel 4** beschreibt die Praxis des Testens in der Klimamodellierung und dient der Beantwortung der Forschungsfrage 1. Im Mittelpunkt steht eine empirische Untersuchung der Testpraxis mehrerer Entwicklungsteams. Ergänzt wird diese durch Erkenntnisse aus der Literatur. [überwiegend eigene Beiträge]
- Basierend auf den Ergebnissen der Studie werden in **Kapitel 5** die Forschungsfragen 2 und 3 bearbeitet und die Konstruktionsaufgabe dieser Arbeit motiviert. Der Fokus liegt dabei auf Unittests, welche in der Klimamodellierung nur selten

zum Einsatz kommen. Gründe und Nachteile dieser Praxis werden diskutiert. [eigene Beiträge]

- In **Kapitel 6** werden Anforderungen an eine Lösung für eine anwendungsorientierte Unterstützung der Unittesterstellung für Klimamodelle formuliert. [eigene Beiträge]
- In **Kapitel 7** werden exemplarisch fünf gängige Methoden zur automatischen Testgenerierung beschrieben und diskutiert, inwieweit sich diese für eine anwendungsorientierte Lösung eignen. [Literatur]
- Als Alternative wird in **Kapitel 8** der Capture-&-Replay-Ansatz vorgestellt. Es werden zunächst die allgemeinen Prinzipien dieses Ansatzes erläutert und anschließend diskutiert, wie dieser zur Erzeugung von Unittests für Klimamodelle eingesetzt werden kann. Entwurfsoptionen bei der Konstruktion eines entsprechenden Softwarewerkzeugs werden erörtert. Zudem werden bereits bestehende Umsetzungen beleuchtet. [überwiegend eigene Beiträge]
- In **Kapitel 9** wird das Softwarewerkzeug FortranTestGenerator, welches den Capture-&-Replay-Ansatz für die unterstützte Erstellung von Unittests umsetzt, vorgestellt. [eigene Beiträge]
- **Kapitel 10** diskutiert Grenzen von Methode und Werkzeug. [eigene Beiträge]
- **Kapitel 11** beschreibt die Erprobung des FortranTestGenerators mit Hilfe von Experimenten zur Funktionsfähigkeit und Nützlichkeit sowie einer Benutzerstudie zum Praxiseinsatz des Werkzeugs. [eigene Beiträge]
- In **Kapitel 12** werden die Ergebnisse dieser Arbeit und ihr wissenschaftlicher Beitrag zusammengefasst sowie Anknüpfungspunkte für weitere Forschungsarbeiten aufgezeigt. [eigene Beiträge]

1.6 Schreibweisen und Typographie

- Eine Dissertation ist nicht nur ein Werk wissenschaftlicher Literatur, sondern auch eine Prüfungsleistung, mit der im Rahmen der Promotion „die Befähigung zu selbstständiger, vertiefter wissenschaftlicher Arbeit [...] zu dokumentieren“ ist (PromO, 2012, §7, Abs. 1). Da hiermit also auch der persönliche wissenschaftliche Beitrag herausgestellt werden soll, erscheint eine Verwendung der ansonsten in wissenschaftlichen Texten üblichen Pluralform unangebracht. Daher werde ich, wo es sich nicht vermeiden lässt, über meine eigene Person, den Autor dieser Arbeit, in der ersten Person Singular schreiben.

- Für Rollennamen und nicht genau abgrenzbare Personengruppen, für die keine geschlechtsneutralen Formen, z.B. durch substantivierte Partizipien, wie etwa „die Studierenden“, zur Verfügung stehen, wird das generische Femininum mit Binnen-I verwendet. Im Sinne der Lesbarkeit werden zugehörige Artikel und Pronomen ausschließlich in der weiblichen Form verwendet, z.B.: „Die EntwicklerIn testet ihren Code.“ Bei eindeutig abgegrenzten Personengruppen wird die jeweils passende weibliche und/oder männliche Form verwendet, z.B.: „Die interviewten Wissenschaftlerinnen und Wissenschaftler“. Für zusammengesetzte Wörter, die Rollennamen oder Personengruppen enthalten, wird die jeweils gebräuchliche Form, i.d.R. mit generischem Maskulinum, verwendet, z.B.: „benutzerfreundlich“. Gleiches gilt auch für aus dem Englischen übernommene, geschlechtsneutrale Fachbegriffe, die im deutschen Kontext zwar männlich klingen und entsprechend verwendet werden, eine Feminisierung das Wort jedoch entstellen würde, z.B.: „Gatekeeper“. Durch generische Formen und ggf. auch durch weibliche und männliche Form bezeichnete Personengruppen schließen selbstverständlich auch Personen ein, die weder weiblich noch männlich sind.
- In der Annahme, dass die LeserInnen dieser Arbeit der englischen Sprache mächtig sind, werden wörtliche Zitate aus englischen Texten im Originalwortlaut abgedruckt und nicht übersetzt.
- Literaturverweise werden in serifenloser Schrift gesetzt. Dabei werden die Quellen indirekter Zitate mit dem Zusatz „vgl.“ gekennzeichnet, Verweise zu weiterführender Literatur dagegen mit „siehe“. Kein Zusatz wird vorangestellt vor Quellenangaben direkter Zitate, Referenzen von im Fließtext benannter Werke oder wenn für einen einzelnen Fachbegriff auf das Werk verwiesen wird, in dem dieser etabliert wurde.
- Wichtige Fachbegriffe oder Namen werden durch *kursive Schrift* hervorgehoben.
- Die Programmiersprache Fortran spielt in dieser Arbeit eine zentrale Rolle. Viele Begriffe haben im Fortran-Universum jedoch eine andere Bedeutung als in anderen Programmiersprachen, so heißen etwa nichtveränderliche Variablen, ansonsten als Konstanten bekannt, in Fortran PARAMETER, während Parameter in Fortran ARGUMENTE heißen. Diese Fortran-spezifische Begriffe werden in Abschnitt 2.4 erläutert und in der gesamten Arbeit, wie hier bereits geschehen, durch KAPITÄLCHEN gekennzeichnet.
- Für Quellcodeelemente wie etwa Variablenbezeichner oder Schlüsselwörter einer Programmiersprache wird der `Schreibmaschinenstil` verwendet.
- Zeilennummern in Quelltextbeispielen dienen einzig der Identifizierung einzelner Zeilen zur Referenzierung im Fließtext und stellen keine logische Nummerierung innerhalb realer oder fiktiver Quelltextdateien dar. Daher führt das

Auslassen einzelner Codeabschnitte (markiert durch „⋮“) nicht zu Sprüngen in der angezeigten Zeilennummerierung.

Beispiel 1.1: Zeilennummern

```
1  MODULE example
2
3  ⋮
4
5  END MODULE example
```

Kapitel 2

Klimamodellierung

Zu den größten wissenschaftlichen Fortschritten, die im letzten Jahrhundert durch das Aufkommen der Computertechnik ermöglicht wurden, gehört die Entwicklung globaler Wetter- und Klimamodelle auf Basis numerischer Simulationen. Dadurch sind heute zum einen Wettervorhersagen für mehrere Tage im Voraus möglich, zum anderen kann durch diese Modelle der Einfluss unterschiedlicher Faktoren auf das Erdklima untersucht werden (vgl. Lynch, 2008). Ein Beispiel hierfür ist der Einfluss von CO_2-Emissionen auf den anhaltenden Klimawandel. Da aufgrund seiner Größe und der relevanten Zeitskalen das *Klimasystem* der Erde nicht durch Experimente in der realen Welt erforscht werden kann, spielen *Simulationen* mit Hilfe von Computermodellen hier eine entscheidende Rolle (vgl. Edwards, 2011).

Diese Arbeit beschäftigt sich mit der softwaretechnischen Praxis des Testens derartiger Modelle. Um diese zu verstehen, ist ein grundlegendes Verständnis des Anwendungsbereichs notwendig. Dieses Kapitel gibt daher eine Einführung in die wissenschaftlichen Methoden und softwaretechnische Praxis der Klimamodellierung.

Zunächst werden dazu einige grundlegende Begriffe definiert und erläutert (Abschnitt 2.1). Danach wird ein kurzer Abriss über die historische Entwicklung der Klimamodellierung und deren Einfluss auf die heutige Praxis gegeben (Abschnitt 2.2). Abschnitt 2.3 betrachtet die Softwarearchitektur heutiger Klimamodelle. Eines der wichtigsten Werkzeuge in der Entwicklung von Klimamodellen stellt die Programmiersprache Fortran dar. Abschnitt 2.4 widmet sich den Besonderheiten dieser Programmiersprache und erläutert wichtige Begriffe. In Abschnitt 2.5 werden schließlich weitere wichtige Technologien vorgestellt.

2.1 Grundlagen

Im Folgenden werden die für diese Arbeit relevanten Begriffe aus der Klimamodellierung eingeführt. Für zentrale Fachbegriffe werden formelle Definitionen gegeben.

2.1.1 Wetter

Es gibt verschiedene Definitionen von Wetter und Klima. Üblicherweise versteht man unter Wetter den Zustand der Atmosphäre zu einem bestimmten Zeitpunkt, ausgedrückt in meteorologischen Größen (Elementen) wie Temperatur, Luftdruck, etc. So definiert die *World Meteorological Organisation (WMO)* im *International Meteorological Vocabulary* (WMO, 1992, S. 672 u. 386):

Definition 2.1: Wetter (weather)

„State of the atmosphere at a particular time, as defined by the various meteorological elements.“

Definition 2.2: Meteorologisches Element (meteorological element)

„Atmospheric variable or phenomenon which characterizes the state of the weather at a specific place at a particular time (e.g., air temperature, pressure, wind, humidity, thunderstorm and fog).“

2.1.2 Klima

Eine häufig zitierte Definition für Klima findet sich in den Berichten des IPCC. Danach versteht man unter Klima im engeren Sinne die Beschreibung des durchschnittlichen Wetters in einem gegebenen Zeitraum durch Angabe von statistischen Mittelwerten und Schwankungen (IPCC, 2013, S. 1450):

Definition 2.3: Klima (climate)

„Climate in a narrow sense is usually defined as the average weather, or more rigorously, as the statistical description in terms of the mean and variability of relevant quantities over a period of time ranging from months to thousands or millions of years. The classical period for averaging these variables is 30 years, as defined by the World Meteorological Organization. The relevant quantities are most often surface variables such as temperature, precipitation and wind. Climate in a wider sense is the state, including a statistical description, of the climate system.“

Die erwähnte 30-Jahr-Periode ergibt sich aus den Empfehlungen der WMO an ihre Mitgliedsstaaten, die nationalen klimatologischen Daten für Zeiträume von eben 30 Jahren zu berechnen und zu veröffentlichen, beginnend ab dem Jahr 1901 (WMO,

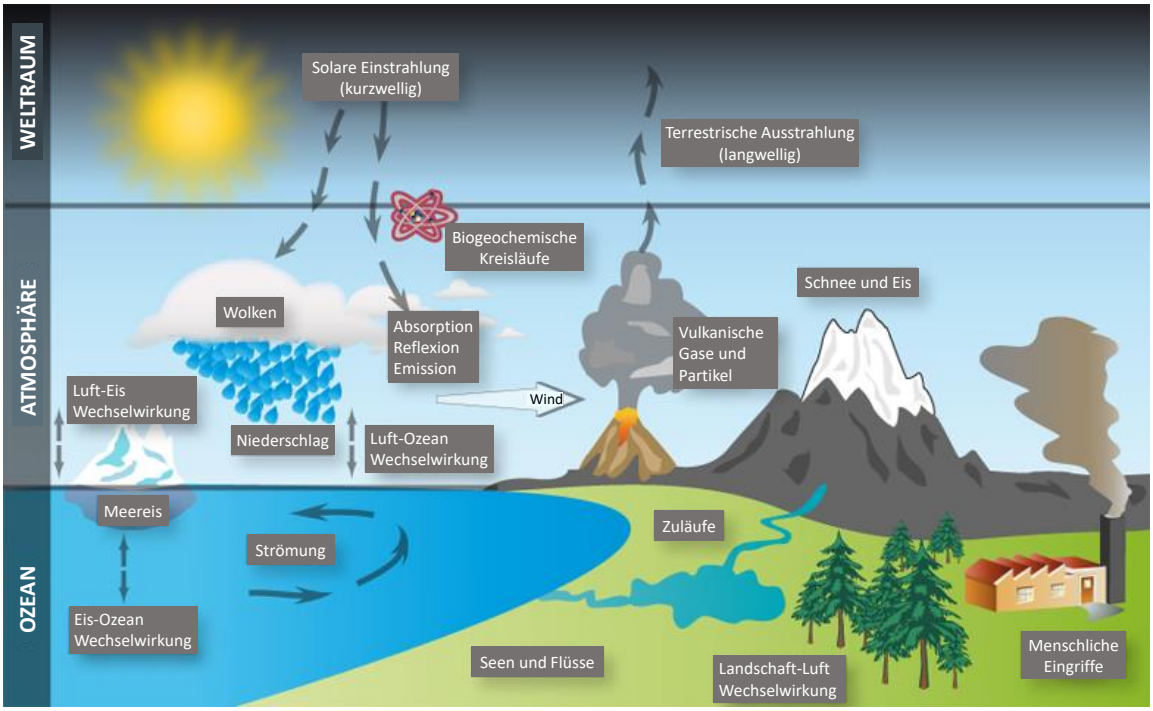


Abbildung 2.1: Vereinfachte Darstellung des Klimasystems (DWD, Klimawandel)

2011, S. 4.15). Laut der Definition 2.3 kann man unter Klima im weiteren Sinne auch den Zustand des Klimasystems verstehen. Das Klimasystem der Erde besteht aus dem Zusammenspiel der fünf Hauptkomponenten *Atmosphäre* (die gasförmige Hülle der Erde), *Hydrosphäre* (Flüsse, Seen und insbesondere Ozeane), *Kryosphäre* (Schnee- und Eisflächen, gefrorener Boden sowie Permafrostgebiete), *Lithosphäre* (Landoberfläche und Meeresböden) und *Biosphäre* (lebende Organismen und Ökosysteme) sowie äußerer Einflüsse wie Sonnenstrahlung, Vulkanausbrüche oder menschliche Einwirkungen. Abbildung 2.1 zeigt eine vereinfachte, schematische Darstellung dieses Systems. Das IPCC definiert (IPCC, 2013, S. 1451):

Definition 2.4: Klimasystem (climate system)

„The climate system is the highly complex system consisting of five major components: the atmosphere, the hydrosphere, the cryosphere, the lithosphere and the biosphere, and the interactions between them. The climate system evolves in time under the influence of its own internal dynamics and because of external forcings such as volcanic eruptions, solar variations and anthropogenic forcings such as the changing composition of the atmosphere and land use change.“

2.1.3 Klimamodell

Unter dem Begriff *Klimamodell* wird in dieser Arbeit gemäß der Definition 1.3 ein Computerprogramm zur Simulation des Klimasystems verstanden. Solch ein Computermodell implementiert numerische Gleichungen, die die physikalischen, chemischen und biologischen Eigenschaften, Prozesse und Wechselwirkungen der Komponenten des Klimasystems beschreiben (vgl. Abbildung 1.1).

Da diese numerischen Beschreibungen erst durch die Implementation in Form eines Computerprogramms ausführbar gemacht werden, erfolgt in der Literatur häufig eine Gleichsetzung dieser beiden Ebenen. So wird der Begriff Klimamodell außerhalb dieser Arbeit i.d.R. als „numerische Repräsentation“ definiert, wie etwa in der Definition des IPCC, 2013, S. 1450:

> „***Climate model (spectrum or hierarchy):*** *A numerical representation of the climate system based on the physical, chemical and biological properties of its components, their interactions and feedback processes, and accounting for some of its known properties. [...] Climate models are applied as a research tool to study and simulate the climate, and for operational purposes, including monthly, seasonal and interannual climate predictions.*“

Die Formulierungen „research tool“, „simulate“ und „operational purposes“ deuten jedoch daraufhin, dass die Autoren auch hier eher an eine ausführbare Form der numerischen Repräsentation, also letztendlich an ein Computerprogramm, gedacht haben. Es ließe sich einwenden, dass auch ein Computermodell nichts anderes sei als eine numerische Repräsentation, dies verkennte jedoch die Tatsache, dass ein ausführbares Klimamodell aus deutlich mehr besteht als aus einer Sammlung in Fortran übersetzter numerischer Algorithmen. Wesentlich sind zum Beispiel Designentscheidungen bzgl. der funktionalen Dekomposition, Schnittstellen, Parallelisierung oder Infrastrukturcode für technische Aspekte wie der Interprozesskommunikation oder der Ein- und Ausgabe. Dies alles führt dazu, dass die Erstellung eines Klimamodells ein komplexer Softwareentwicklungsprozess ist und nicht nur das bloße „Herunterprogrammieren“ numerischer Algorithmen.

2.1.4 Modelltypen

Für verschiedene Klimamodelle lassen sich entlang verschiedener Dimensionen in Kategorien einteilen. Ein Unterscheidungsmerkmal ist beispielsweise die Komplexität der Modelle. Zur Erforschung einzelner physikalischer Prozesse existieren einfache Modelle, die gezielt nur die jeweils interessanten Aspekte abbilden. Ein Beispiel hierfür sind sog. *Energie-Bilanz-Modelle*, welche Veränderungen des Klimas nur auf Basis

einer Analyse des Energiehaushalts der Erde abschätzen. Mit diesen hochgradig idealisierten Modellen lassen sich keine realistischen Bedingungen simulieren (vgl. Goosse, 2015, S. 78f).

Für die Simulation realistischer Bedingungen werden sog. *General Circulation Models (GCM)* verwendet. Es existieren GCMs für die Simulation der Atmosphäre (*AGCM*) und für die Ozeane (*OGCM*). Atmosphärenmodelle enthalten häufig auch Komponenten für die Simulation der Landoberfläche bzw. werden mit entsprechenden Landmodellen verbunden. Gleiches gilt für Seeeismodelle, welche häufig mit Ozeanmodellen verbunden sind. Werden all diese Teilmodelle, d.h. Atmosphäre und Ozean bzw. Atmosphäre, Land, Ozean und Seeeis miteinander verbunden, spricht man von *gekoppelten Modellen*. Gebräuchliche Abkürzungen sind hier sowohl *CGCM* (Coupled) als auch *AOGCM* (Atmosphere-Ocean) (vgl. Heavens, Ward und Mahowald, 2013; Goosse, 2015, S. 77).

Aktuelle Modelle enthalten zudem Komponenten und Teilmodelle für biologische und chemische Prozesse, wie etwa der Landvegetation oder des CO_2-Kreislaufs. Solche Modelle werden *Erdsystemmodelle (Earth System Models, ESM)* genannt (vgl. Heavens, Ward und Mahowald, 2013).

Auf der Komplexitätsskala zwischen den einfachen Modellen und den GCMs liegen die sog. *Earth (System) Models of Intermediate Complexity (EMIC)*. Sie enthalten komplexere Abbildungen des Klimasystems als die einfachen Modelle, aber auch deutlich mehr Vereinfachungen und *Parametrisierungen*, sowie meist gröbere *Auflösungen* als GCMs (Goosse, 2015, S. 77). EMICs ermöglichen die Simulation längerer Zeiträume, da ihr Rechenaufwand geringer ist als der von GCMs (Goosse, 2015, S. 80).

Neben *globalen* Klimamodellen, welche die relevanten Prozesse für die gesamte Erde berechnen, existieren auch *regionale* Modelle, die die Klimaentwicklung für einzelne Gebiete der Erde simulieren können. Diese ermöglichen es, durch eine meist höhere Auflösung auch kleinskalige regionale Phänomene zu erforschen. Das Gebiet, für das ein Modell seine Berechnungen durchgeführt, wird auch *Domäne (domain)* genannt. Regionale Modelle benötigen an den Rändern ihrer Domäne externe Daten. Diese stammen entweder aus Beobachtungsdaten oder häufig auch von einem globalen Modell, welches aber mit geringerer Auflösung betrieben wird (Goosse, 2015, S. 82f).

In dieser Arbeit stehen globale GCMs und gekoppelte Modelle bzw. Erdsystemmodelle im Mittelpunkt.

2.1.5 Gitter und Auflösung

Die Berechnungen in einem Klimamodell erfolgen in einem dreidimensionalen *Gitter*, durch welches die Domäne in mehrere Gitterzellen unterteilt wird. Für diese werden

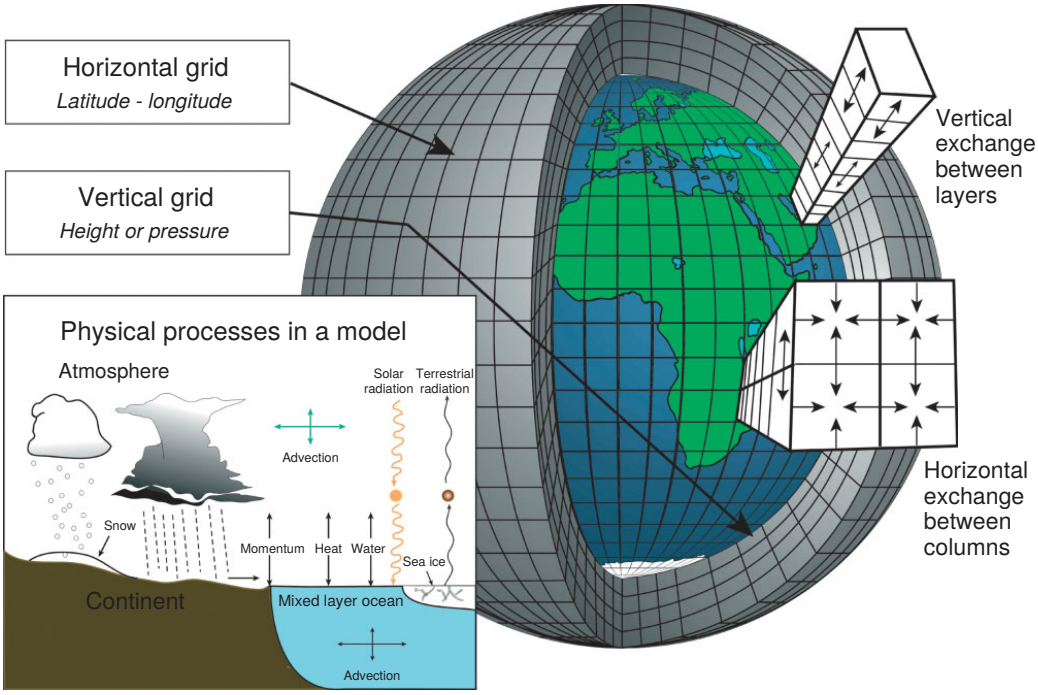
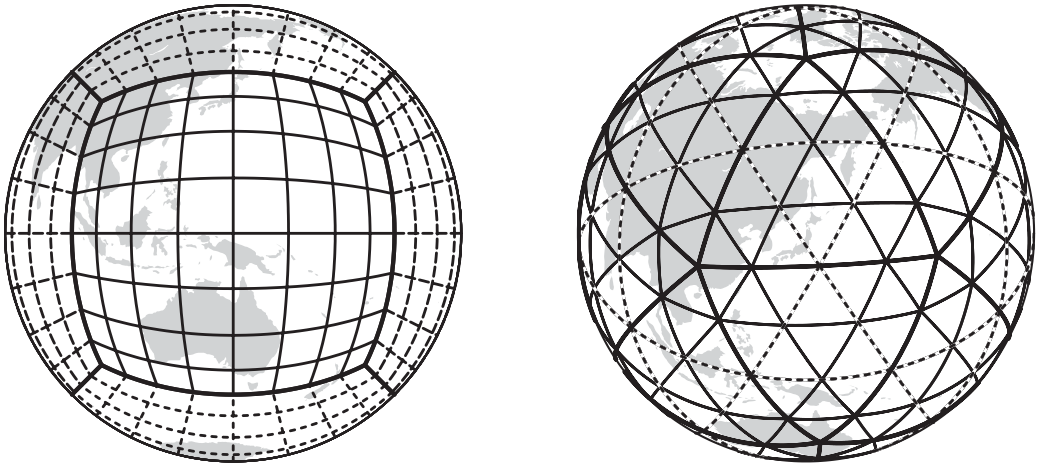


Abbildung 2.2: Kartesische Gitterstruktur von Klimamodellen. Grafik von Courtney Ritz and Trevor Burnham (zitiert nach Edwards, 2011).

die numerischen Gleichungen separat, jedoch unter Berücksichtigung der Nachbarzellen, gelöst. Durch diese Diskretisierung eignen sich Klimamodelle besonders gut für die Berechnung auf parallelen Hochleistungsrechnern. Einzelne Rechnerknoten berechnen dabei jeweils die ihnen zugewiesene Teilmenge von Gitterzellen. Traditionell orientiert sich das Gitter dabei, wie in Abbildung 2.2 dargestellt, an Breiten- und Längengraden. Diese Gitter haben jedoch das Problem, dass die Gitterzellen zu den Polen hin immer kleiner werden, was zu numerischen Problemen führt. Neuere Modelle verwenden daher andere Gitter, die den Globus in möglichst gleichgroße Flächen aufteilen (vgl. Staniforth und Thuburn, 2012). Abbildung 2.3 zeigt zwei Beispiele solcher Gitter.

Die Auflösung eines Modells bezeichnet die Feinheit des Gitters. Je höher die Auflösung, desto kleiner die Gitterzellen. Mit wachsender Auflösung erhöht sich die Genauigkeit eines Modells, aber auch der Rechenaufwand. Typische Auflösungen aktueller globaler Atmosphärenmodelle liegen bei 50 bis 200 km Abstand zwischen einzelnen Gitterpunkten (vgl. Goosse, 2015, S. 81). Es sind jedoch auch Modelle in Entwicklung, die mit deutlich höheren Auflösungen arbeiten (siehe auch Haarsma u. a., 2016; Satoh, Stevens u. a., 2019).

Neben der räumlichen spielt auch die zeitliche Auflösung eine Rolle. Klimamodelle berechnen den Zustand von Atmosphäre, Ozean etc. über die Zeit in diskreten *Zeit-*



(a) Cubed Sphere Grid (b) Icosahedral Grid

Abbildung 2.3: Zwei neuere Gitterarten (Williamson, 2007)

schritten festgelegter Länge. Der Zustand zu einem Zeitpunkt ergibt sich dabei aus dem vorherigen Zustand (bzw. einer begrenzten Zahl vorheriger Zustände) und den jeweiligen Randbedingungen. Auch hier gilt in der Regel: je höher die zeitliche Auflösung, d.h. je kürzer die Zeitschritte, desto genauer die Berechnungen und desto höher der Rechenaufwand. Je nach Experiment werden in der Praxis Zeitschrittlängen von wenigen Sekunden bis zu Jahren verwendet (vgl. Goosse, 2015, S. 73ff).

Räumlich und zeitliche Auflösung sind nicht unabhängig voneinander. Grundsätzlich gilt, dass je höher die räumliche Auflösung eines Modells ist, desto höher muss auch die zeitliche Auflösung sein, damit innerhalb eines Zeitschritts keine Zustandsinformation eine Gitterzelle überspringt (vgl. McGuffie und Henderson-Sellers, 2005, S. 170).

2.1.6 Anfangs- und Randbedingungen

Die Berechnungen von Klimamodellen basieren auf zwei Sorten von Eingabedaten: *Anfangs- und Randbedingungen*. Die Anfangsbedingungen (*initial conditions*) beschreiben den Zustand des Klimasystems zu Beginn des zu simulierenden Zeitraums. Sie bestehen somit aus den initialen Werten der Zustandsvariablen. Die Randbedingungen (*boundary conditions*) enthalten die Größen, welche über die Zeit das Klima beeinflussen, aber dem Modell vorgegeben und von diesem nicht selbst berechnet werden, wie etwa die Topographie von Land und Ozeanen oder die Solarkonstante. Ändern sich einzelne Randbedingungen über die Zeit, werden diese Größen als

Antriebskräfte (forcings) bezeichnet (vgl. Easterbrook, 2010a). Ein Beispiel für eine Antriebskraft ist etwa ein sich verändernder menschlicher CO_2-Ausstoß. Anfangs- und Randbedingungen werden von Klimamodellen als vorbereitete Daten aus Dateien ausgelesen oder von anderen Modellen bzw. Modellkomponenten übernommen.

Ob eine Größe eine Zustandsvariable, eine Randbedingung oder eine Antriebskraft ist, ist je nach Modell(komponente) und Experiment unterschiedlich. So ist beispielsweise die Topographie von Eisschilden in herkömmlichen Klimasimulationen, die sich über Jahrzehnte oder Jahrhunderte erstrecken, konstant, in Modelle, die längere Zeiträume simulieren, können Änderungen der Eisschildtopographie als Antriebskraft eingehen (vgl. Goosse, 2015, S. 75). Zudem sind beispielsweise die Oberflächentemperaturen des Ozeans für ein Atmosphärenmodell eine Antriebskraft, für ein Ozeanmodell wiederum eine Zustandsvariable. Für gekoppelte Modelle gilt, je mehr Komponenten es enthält, je mehr Zustandsgrößen es somit selbst berechnen kann, desto weniger Randbedingungen und Antriebskräfte benötigt es als Eingabedaten.

GCMs basieren auf nichtlinearen partiellen Differentialgleichungen, d.h. sie beschreiben chaotische Systeme. Für kurze Simulationszeiträume, wie in der Wettervorhersage, bedeutet dies eine starke Abhängigkeit von den Anfangsbedingungen, d.h. kleine Unterschiede können zu sehr unterschiedlichen Ergebnissen führen. Wettervorhersagen sind daher umso genauer, je präziser die Messdaten sind, die als Anfangsbedingungen verwendet werden. Für längere Klimasimulationen ist die Abhängigkeit von den Anfangsbedingungen nicht so stark, da diese kaum Einfluss auf langfristige statistische Trends haben. Eine umso wichtigere Rolle spielen hier die Randbedingungen (vgl. McGuffie und Henderson-Sellers, 2005, S. 11; Easterbrook, 2010a).

2.1.7 Simulation und Konfiguration

Im Kontext dieser Arbeit werden die Begriffe Simulation und Experiment synonym wie folgt verwendet:

Definition 2.5: Simulation/Experiment

Ausführung eines Klimamodells in einer wissenschaftlich relevanten Konfiguration.

Die Begriffe stehen dabei sowohl für einzelne Ausführungen wie auch für Klassen von Ausführungen, die durch ihre jeweiligen Konfigurationen und wissenschaftlichen Fragestellungen definiert sind.

Während einer Simulation durchläuft ein Klimamodell die Abbildung 2.4 dargestellten Phasen:

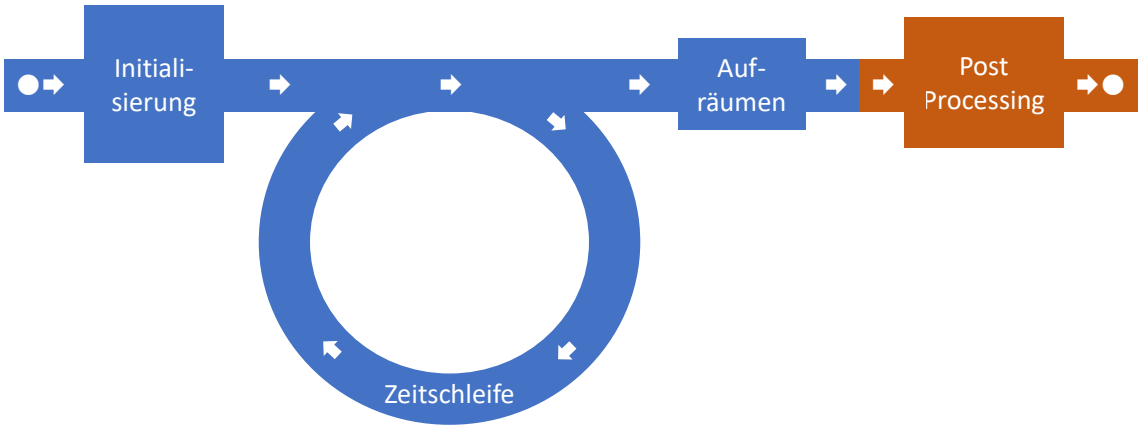


Abbildung 2.4: Phasen einer Simulation

Initialisierung In der Initialisierungsphase werden Daten aus Dateien eingelesen, die beispielsweise Informationen enthalten zur verwendeten Gitterstruktur, zur Topografie oder zu Anfangs- und Randbedingungen.

Zeitschleife Die Zeitschleife ist der Hauptteil der Simulation. In der Konfiguration des Modells wird festgelegt, welcher Zeitraum berechnet werden soll und in welchen Zeitschritten. In der Zeitschleife wird dann Zeitschritt für Zeitschritt der Zustand des Klimasystems berechnet bis das Ende der gewünschten Simulationszeit erreicht ist. Am Ende der Zeitschleife und, je nach Konfiguration, auch währenddessen werden die vom Modell berechneten Daten in Dateien geschrieben. Zudem müssen während der Zeitschleife ggf. Antriebskraftdaten eingelesen werden.

Aufräumen In der Aufräumphase werden abschließende Arbeiten erledigt, wie z.B. Speicher freigeben oder Dateicaches leeren.

Postprocessing Beim Postprocessing werden die vom Modell produzierten Daten aufbereitet, z.B. indem Visualisierungen in Form von Diagrammen erstellt werden. Abbildung 2.5 zeigt typische Beispiele für solche Diagramme. Das Postprocessing ist meist nicht Teil des eigentlichen Klimamodellprogramms, sondern wird mit Hilfe externer Programme und Skripte erledigt.

Klimamodelle enthalten unzählige Konfigurationsmöglichkeiten, dadurch lassen sich viele verschiedene Simulationen durchführen. So lassen sich beispielsweise Teilmodelle an- und ausschalten, numerische Parameter setzen, für einzelne Aspekte aus verschiedenen Berechnungsmethoden auswählen, die Gitterauflösung festlegen, Zeitraum- und Zeitschrittlänge bestimmen, Datenquellen für Anfangs- und Randbedingungen festlegen usw. usf. Einige dieser Konfigurationen werden zur Übersetzungszeit, ande-

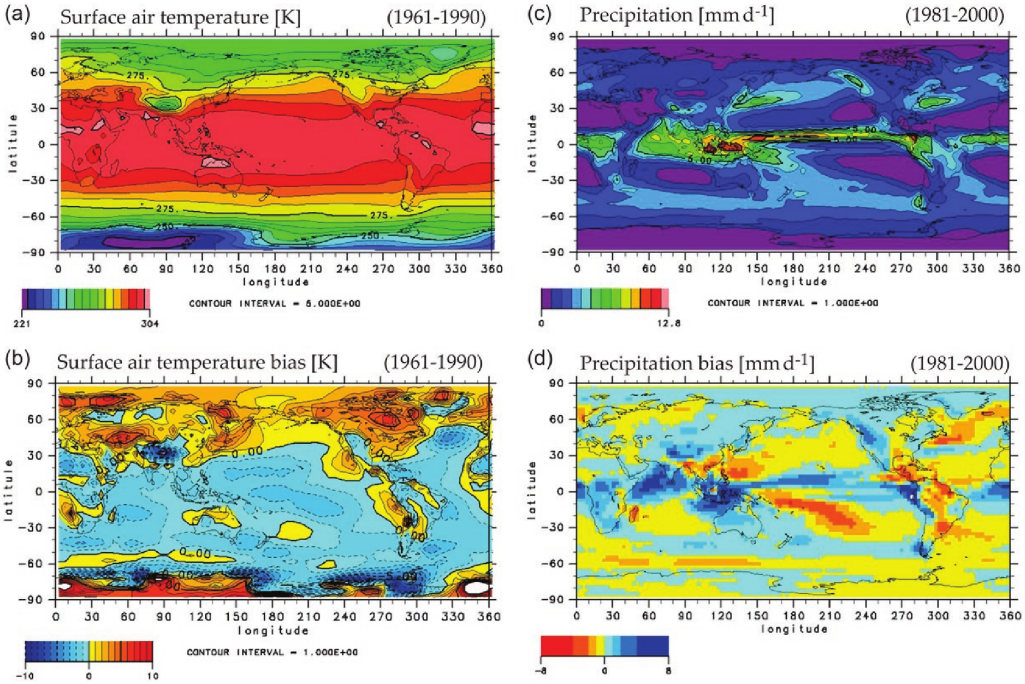


Abbildung 2.5: Beispiele für Diagramme zur Visualisierung von Klimamodelldaten (Watanabe u. a., 2011)

re zur Laufzeit festgelegt. Für eine Simulation werden somit i.d.R. ein kompiliertes Klimamodell, Konfigurationsdateien und Dateien mit Eingabedaten benötigt.

Einige Klimamodelle enthalten besondere Testmodi, mit denen i.d.R. technische Aspekte getestet werden. Führt man ein Klimamodell in einem solchen Modus aus, spricht man normalerweise nicht von „Simulation“ oder „Experiment“.

Häufig benötigen Klimamodelle einige Zeitschritte bis das berechnete Klima einen stabilen Zustand erreicht hat. Diesen Zeitraum nennt man *Spin-Up-Phase*. Heutige Modell starten ihre Simulationen häufig nicht mit präzisen Anfangsbedingungen, sondern aus einem Ruhezustand heraus, aus dem das Modell im Laufe der Spin-Up-Phase ein realistisches Klima entwickelt (vgl. Edwards, 2010, S. 148f).

2.1.8 Ungenauigkeiten und Unsicherheiten

Wie jedes Modell bilden auch Klimamodelle die Realität nicht genau ab, sondern nur annäherungsweise. Approximationen finden dabei auf jeder Ebene der Modellbildung (vgl. Abbildung 1.1) statt, angefangen beim qualitativen physikalischen Modell, welches nur die wichtigsten Einflussfaktoren erfasst und von anderen abstrahiert, bis hin

zur digitalen Berechnung auf Computersystemen, welche über eine begrenzte Genauigkeit verfügen (vgl. Müller, 2010; Hinsen, 2015). Modellergebnisse sind daher zum einen mit *Ungenauigkeiten* behaftet, da sie die Realität nicht exakt abbilden können, und zum anderen mit *Unsicherheiten*, da bei der bei Simulation zukünftiger Zeiträume und fiktiver Szenarien die Abweichung der Ergebnisse von der Realität unbekannt ist.

Eine wesentliche Quelle dieser Unsicherheiten stellen *Parametrisierungen* dar. Parametrisierung werden verwendet, wenn einzelne physikalische Phänomene innerhalb des Modells numerisch nicht abgebildet werden können, beispielsweise, weil sie auf Skalen stattfinden, die vom Modellgitter nicht aufgelöst werden, weil ihre Berechnung zu aufwendig wäre oder weil schlicht keine mathematische Beschreibung des Phänomens existiert. Parametrisierungen können unterschiedliche Formen annehmen, sie können z.B. aus einzelnen Konstanten bestehen, deren Werte die EntwicklerInnen aus eigenen Erfahrungen oder der Literatur ableiten, aus vereinfachten Berechnungen oder aus größeren Datensätzen, die aus Dateien eingelesen werden.

Parametrisierungen unterliegen häufig *Kalibrierungen*. Dabei werden einzelne Parameter so verändert, dass bei Simulationen ausgewählter historischer Szenarien, die Abweichungen zwischen Modellergebnissen und Messdaten möglichst klein werden. Man spricht dabei auch von *Modell-* bzw. *Parametertuning* oder *-fitting*. Aus wissenschaftstheoretischer Sicht ist dieses Vorgehen nicht unumstritten. Zum einen erhöht dieses Vorgehen die Unsicherheit von Prognosen, da unklar ist, ob gute Modellergebnisse in historischen Szenarien aufgrund guter Modellierung oder allein aufgrund des Parameter-Tunings erreicht wurden. Zum anderen werden Parameter dabei jeweils auf ein bestimmtes Szenario hin kalibriert, dies bedeutet aber nicht, dass sie in anderen Szenarien ebenso gute Ergebnisse liefern (Müller, 2010).

Aus der inhärenten Ungenauigkeit und Unsicherheit folgt, dass es kein „korrektes“ Modell und keine „korrekten“ Modellergebnisse geben kann. Modelle und ihre Ergebnisse können aus wissenschaftlicher Sicht nützlich sein und über eine gewisse Güte verfügen. Diese Güte ergibt sich aus dem Vergleich der Modellergebnisse mit Referenzdaten, beispielsweise historischer Messungen, welche wiederum selbst einer Ungenauigkeit und Unsicherheit unterliegen (vgl. Guillemot, 2010; Parker, 2010).

Eine Methode Ungenauigkeiten und Unsicherheiten zu reduzieren, sind sog. *Ensembles*. Dabei werden Simulationen mehrfach mit variierenden Parametrisierungen durchgeführt. Die Ergebnisse lassen sich anschließend statistisch auswerten, um so zu einem genaueren Ergebnis und/oder einer Abschätzung des Unsicherheitsbereichs zu kommen (vgl. Edwards, 2010, S. 352ff; Parker, 2010).

2.1.9 Rollen

Klimamodelle werden in der Regel von größeren Teams von *WissenschaftlerInnen* und *ProgrammiererInnen* entwickelt. Diese heißen im Kontext dieser Arbeit *EntwicklerInnen*. Meist bestehen Entwicklungsteams einerseits aus wechselndem Personal, das in Rahmen typischer befristeter Forschungsstellen angestellt ist, und einem Kernteam aus längerfristig bzw. dauerhaft angestellten EntwicklerInnen und Leitungspersonal. Die zweite Gruppe hat in der Regel mehr Erfahrung, mehr technische Kenntnisse und trifft strategische Entscheidungen, sofern diese nicht von außen, z.B. von der Institutsleitung vorgegeben werden.

Eine strikte Unterscheidung zwischen WissenschaftlerInnen und ProgrammiererInnen gibt es in der Regel nicht. Diese ist mehr eine Frage der persönlichen Einschätzung des eigenen Arbeitsschwerpunkts, ob dieser mehr in der wissenschaftlichen oder technischen Weiterentwicklung eines Modells liegt. In der Regel haben sowohl WissenschaftlerInnen als auch ProgrammiererInnen einen Hintergrund in naturwissenschaftlichen Disziplinen wie Meteorologie, Ozeanografie, Physik, Mathematik etc. Eine formale Ausbildung in Informatik oder Softwareentwicklung findet sich äußerst selten.

EntwicklerInnen verwenden ihre Modelle selbst für eigene Forschungsarbeiten und die Entwicklung wird maßgeblich durch die eigenen Forschungsinteressen der EntwicklerInnen bzw. ihrer Institute getrieben. Daneben gibt es weitere WissenschaftlerInnen, die Klimamodelle für ihre Forschungsprojekte verwenden, ohne selbst zum eigentlichen Entwicklungsteams zu gehören. Dies können zum Beispiel Doktoranden des eigenen Instituts sein oder MitarbeiterInnen und Studierende anderer Forschungseinrichtungen. Diese Personengruppen werden von den KernentwicklerInnen häufig auch als *BenutzerInnen* bezeichnet. In der Regel müssen diese „BenutzerInnen“ eigene Entwicklungsarbeit leisten, z.B. um selbst entwickelte Berechnungsverfahren einzubauen oder das benutzte Modell mit einem eigenen Modell zu koppeln. In manchen Fällen übernimmt das Entwicklungsteam auf diese Weise entstandene Erweiterungen oder Änderungen des Modellcodes.

Durch diese Zusammenarbeit ist schwierig, genaue Zahlen der an einem Modell beteiligten EntwicklerInnen zu ermitteln. Geschätzt werden kann, dass die Entwicklungsteams von großen ESMs Größen von über einhundert Beteiligten erreichen können.

2.2 Historische Entwicklung

Wetter- und Klimammodellierung blicken auf eine über 70-jährige Tradition zurück. Heutige Praktiken können nicht unabhängig von diesem historischen Kontext betrachtet werden. Daher gibt dieser Abschnitt eine kurze Zusammenfassung der wich-

tigsten Aspekte der geschichtlichen Entwicklung der Klimamodellierung. Ausführlichere Abhandlungen finden sich beispielsweise in Lynch, 2008 oder Edwards, 2011.

2.2.1 Entstehung

Die Entwicklung von Wetter- und Klimamodellen ist eng mit der Entwicklung des Computers verbunden. Die mathematischen (Abbe, 1901; Bjerknes, 1904) und numerischen Grundlagen (Richardson, 1922) wurden jedoch schon vor dessen Erfindung gelegt. Bemerkenswert in dem Zusammenhang ist unter anderem die Beschreibung einer fiktiven „Forecast Factory“ von Lewis Fry Richardson (1922), die im Aufbau der Architektur heutiger Computermodelle im Hinblick auf die Verteilung auf Parallelrechnerknoten sehr nahekommt (vgl. Lynch, 2008). Abbildung 2.6 zeigt eine künstlerische Darstellung dieser Fantasie.

> *„Imagine a large hall like a theatre, except that the circles and galleries go right round through the space usually occupied by the stage. The walls of this chamber are painted to form a map of the globe. The ceiling represents the north polar regions, England is in the gallery, the tropics in the upper circle, Australia on the dress circle and the antarctic in the pit. A myriad computers are at work upon the weather of the part of the map where each sits, but each computer attends only to one equation or part of an equation. The work of each region is coordinated by an official of higher rank. Numerous little ‚night signs‘ display the instantaneous values so that neighbouring computers can read them.“* (Richardson, 1922, S. 219)

Zum Durchbruch gelangte die *numerische Wettervorhersage (Numerical Weather Prediction, NWP)* in den späten 1940er bis frühen 1950er Jahren durch die Erfindung der Digitalrechner. 1945 veröffentlichte John von Neumann seine einflussreiche Arbeit „First Draft of a Report on the EDVAC“, in der er erstmalig das heute als von-Neumann-Architektur bekannte Konzept des speicherprogrammierbaren Rechners beschrieb (Neumann, 1945). Von 1946 bis 1952 leitete er die Entwicklung eines auf dieser Architektur aufbauenden Rechners am *Institute for Advanced Studies (IAS)* in Princeton, USA. Von Neumann erkannte die Meteorologie als ideales Anwendungsgebiet für die Computertechnik (auch vor dem Hintergrund des militärischen Nutzens) und initiierte bereits zu Beginn der Arbeiten am IAS-Rechner ein meteorologisches Forschungsprojekt mit dem Ziel, eine numerische Wettervorhersage auf diesem Rechner ausführen zu können. Die erste computergestützte numerische Wettervorhersage gelang den Forschern 1950 (Charney, Fjörtoft und Neumann, 1950), jedoch noch nicht auf der IAS-Maschine, sondern auf der *ENIAC (Electronic Numerical Integrator and Computer)* in Aberdeen (vgl. Nebeker, 1995; Lynch, 2008).

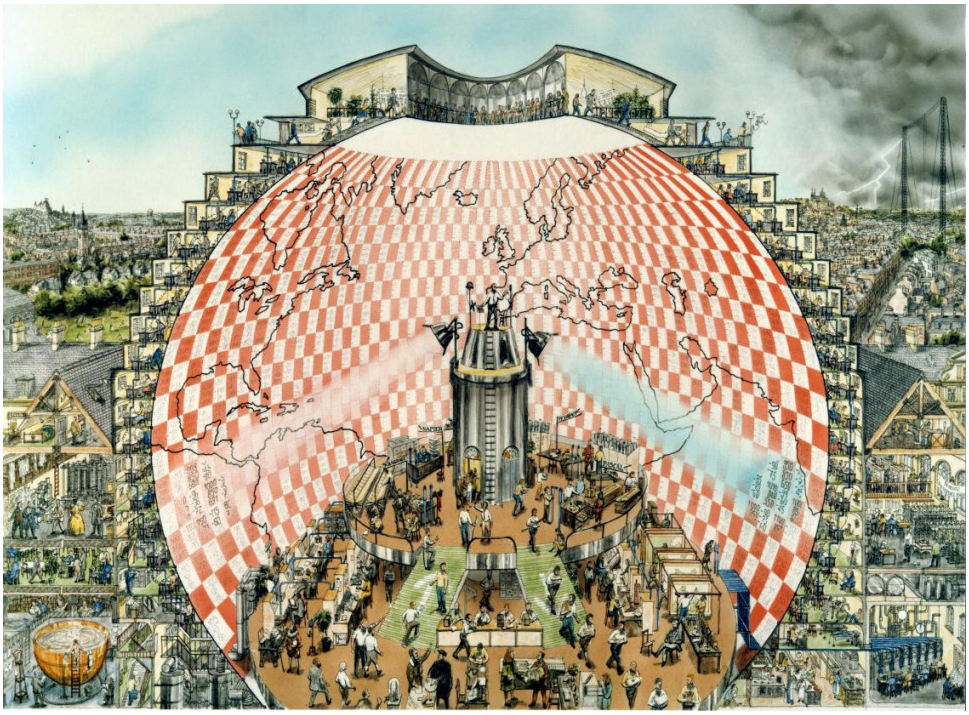


Abbildung 2.6: Künstlerische Darstellung von Richardsons Forecast Factory von Stephen Conlin, 1986 (zitiert nach Lynch, 2016)

Diese frühen Modelle konnten nur relativ kurze Zeiträume um die 24 Stunden simulieren und waren so in der Lage aus den Messdaten eines Tages das Wetter des nächsten Tages zu berechnen. Entwicklungen des Klimas konnten damit nicht erforscht werden. 1955 gelang Norman Philipps, ebenfalls vom IAS in Princeton, erstmalig eine Langzeitsimulation der atmosphärischen Zirkulation (Phillips, 1956). Sein Modell gilt als der erste Prototyp eines General Circulation Models. In den Folgejahren begannen Forschungsinstitute rund um die Welt mit der Entwicklung von GCMs (vgl. Lynch, 2008). Die ersten Ergebnisse eines gekoppelten Atmosphäre/Ozean-Modells wurden 1969 von Syukuro Manabe und Kirk Bryan veröffentlicht (Manabe und Bryan, 1969; vgl. Edwards, 2010).

2.2.2 Klimawandel und IPCC

In den 1820er Jahren beschrieb Joseph Fourier das als *Treibhauseffekt* bekannte Phänomen, dass die Atmosphäre Wärme zurückhalten kann und dadurch die Temperatur der Erde höher ist als sie ohne Atmosphäre wäre (Fourier, 1822, vgl. Edwards, 2010). Einige Jahrzehnte später wurde der Einfluss von Wasserdampf und Kohlendioxid auf die Erderwärmung erkannt (vgl. Dessler, 2011, S. 198). 1896 schätzte der Schwede Svante

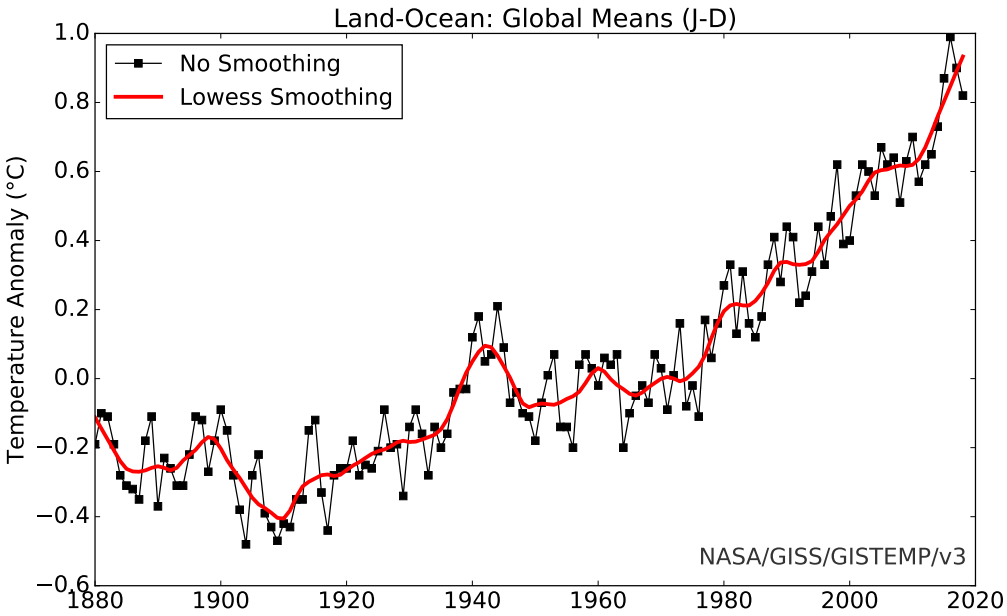


Abbildung 2.7: Globale durchschnittliche Temperaturanomalien bezogen auf die mittelere Temperatur der Jahre 1951-1980. Die rote Linie zeigt das laufende 5-Jahres-Mittel. Erstellt mit dem Custom Plotter der NASA GISS Surface Temperature Analysis (GISTEMP, 2019).

Arrhenius, dass eine Verdopplung des CO_2-Gehalts zu einem Temperaturanstieg von 5-6° Celsius führt (Arrhenius, 1896; vgl. Uppenbrink, 1996).

Zu Beginn des 20. Jahrhunderts kam es zu einem Anstieg der globalen Durchschnittstemperaturen (siehe Abbildung 2.7) und der Brite Guy Stewart Callendar (1938) berechnete, dass menschliche CO_2-Emissionen durch das Verbrennen fossiler Energieträger hierfür verantwortlich sind (vgl. Dessler, 2011, S. 199f). Zwar blieb die globale Durchschnittstemperatur zwischen den 1940er- und 1970er-Jahre weitestgehend konstant, dennoch hatte sich zum Ende der 1970er-Jahre die Erkenntnis, dass der von der Menschheit verursachte Anstieg der CO_2-Gehalts der Atmosphäre zu einer Erwärmung der Erde führt, in der Wissenschaft weitestgehend durchgesetzt (vgl. Dessler, 2011, S. 206). In einem Bericht des US-amerikanischen *National Research Council* heißt es:

> *„The conclusions of this brief but intense investigation may be conforting to scientists but disturbing to policymakers. If carbon dioxide continues to increase, the study group finds no reason to doubt that climate changes will result and no reason to believe that these changes will be negligible. [...] A wait-and-see policy may mean waiting until it is too late.“*
> (NRC, 1979, S. viii)

Nach einem erneuten Anstieg der globalen Durchschnittstemperaturen in den 1980er-Jahren gründete sich 1988 das *Intergovernmental Panel on Climate Change (IPCC)*, auch *Weltklimarat* genannt. Es soll politischen Entscheidungsträger regelmäßig wissenschaftliche Bewertungen des Klimawandels, seiner Auswirkungen und Risiken liefern, sowie Möglichkeiten zur Anpassung und Minderung aufzeigen (vgl. ipcc.ch). Wichtigstes Instrument sind hier neben Beobachtungsdaten die Simulationsdaten, die von verschiedenen Modellen aus der ganzen Welt geliefert werden.

Einen wesentlichen Teil der Daten liefert u.a. das *Coupled Model Intercomparison Project (CMIP)*. Dieses legt standardisierte Experimente fest, die von allen beteiligten Forschungsinstituten mit Hilfe ihrer jeweiligen Modelle durchgeführt werden. Hierzu gehören u.a. Zukunftsprognosen bezogen auf Szenarien unterschiedlicher Mengen von CO_2-Emissionen (siehe auch Moss u. a., 2010). Der aktuelle Durchgang CMIP6 enthält nicht nur Experimente für gekoppelte GCMs und ESMs, sondern umfasst eine Vielzahl von MIPs für unterschiedliche Modelltypen (siehe auch Eyring u. a., 2016). Die MIP-Experimente haben für die beteiligten Forschungsinstitute maßgeblichen Einfluss auf die inhaltliche und zeitliche Planung der Modellentwicklung.

Der Vergleich mit anderen Modellen ermöglicht den EntwicklerInnen zum einen ihr eigenes Modell zu evaluieren, zum anderen lassen sich die Ergebnisse eines Experiments, das von mehreren Modelle durchgeführt wurde, statistisch auswerten, beispielsweise um Unsicherheiten einzuschätzen. Die Kombination von Ergebnissen mehrerer Modelle nennt man auch *Multi-Modell-Ensembles*. Diese ermöglichen in vielen Fällen bessere Vorhersagen als einzelne Modelle (vgl. Tebaldi und Knutti, 2007).

2.2.3 Community

Zwar hat das IPCC 195 Mitgliedsländer (vgl. ipcc.ch), da die Entwicklung der Modelle sowie der Betrieb geeigneter Hochleistungsrechner jedoch mit hohen Kosten verbunden sind, stammen die Modelle vorwiegend aus den entwickelten Industrieländern. In den letzten IPCC-Report sind die Ergebnisse von gekoppelten GCMs und ESMs aus Australien, China, Deutschland, Frankreich, Großbritannien, Italien, Japan, Kanada, Korea, Norwegen, Russland, den USA sowie eines Konsortiums verschiedener europäischer Länder eingeflossen (vgl. IPCC, 2013, S. 854ff).

Trotz der gemeinsamen physikalischen Basis unterscheiden sich einzelne Klimamodelle in vielerlei Hinsicht. So werden verschiedene Gitterstrukturen oder numerische Verfahren eingesetzt oder unterschiedliche Schwerpunkte auf einzelne physikalische Prozesse gesetzt. Beeinflusst wird dies zum einen durch die wissenschaftliche Ausrichtung des jeweiligen Instituts, aber auch durch technologische Gegebenheiten, wie z.B. der verfügbaren Rechnertechnologie.

Jedoch entstehen neue Klimamodelle in der Regel nicht im luftleeren Raum, sondern auf Basis bereits existierender Modelle. Dabei existieren unterschiedliche Verbindungen. So kann ein neues Modell ganze Codeteile eines anderen übernehmen oder ein einzelnes mathematisches Schema. Auch gibt es personellen Austausch zwischen den Instituten, wodurch es zum Transfer von Wissen und Erfahrung in der Modellentwicklung oder wiederum von Code kommen kann (vgl. Edwards, 2010, S. 167ff).

Bei gekoppelten GCMs und ESMs ist es zudem nicht selten, dass ganze Teilmodelle von anderen Forschungsinstituten übernommen werden, z.B. wenn kein entsprechendes eigenes Teilmodell verfügbar ist. So zeigt z.B. die Auflistung der AOGCMs und ESMs in IPCC, 2013, S. 854ff, das wenigstens sechs Modellfamilien, u.a. aus Australien und China, auf dem Ozeanmodell des US-amerikanischen GFDL basieren. Aufgrund dieser mannigfaltigen Zusammenarbeit ist anzunehmen, dass es auch einen Austausch bezogen auf Softwareentwicklungspraktiken innerhalb der Community gibt. Eine Gemeinsamkeit ist beispielsweise die Verwendung von Fortran als Hauptprogrammiersprache (siehe auch Easterbrook, 2010b; Méndez, Tinetti und Overbey, 2014)[4].

2.2.4 Evolution

Der Fortschritt im Bereich der Klimamodellierung findet im Wesentlichen innerhalb von zwei Dimensionen statt: Zum einen können mit der Zeit immer mehr physikalische Prozesse innerhalb der Modelle abgebildet werden. So werden immer mehr Teilmodelle miteinander gekoppelt und innerhalb der Teilmodelle können immer mehr physikalischer Größen direkt berechnet werden, was die Notwendigkeit von Parametrisierungen verringert. Abbildung 2.8 zeigt diese Entwicklung von den 1970er Jahren bis zum vierten Sachstandsberichts des IPCC im Jahr 2007 (AR4).

Zum anderen verfeinert sich die Auflösung der Modelle. Während die Modelle zur Zeit des ersten Sachstandsberichts des IPCC (FAR) im Jahr 1990 noch eine typische Gitterzellenbreite von 500 km hatten, lag diese beim vierten Bericht bei etwa 110 km (siehe Abbildung 2.9). Neueste Modelle arbeiten mit sehr hohen Auflösungen von zum Teil unter 5 km Gitterzellenbreite, um die Simulation von Wolken zu ermöglichen (vgl. Satoh, Stevens u. a., 2019). Da jedoch der Rechenaufwand mit steigender Auflösung zunimmt, hängt die Wahl der Auflösung auch vom gewünschten Simulationszeitraum ab.

Neben der Entwicklung effizienterer numerischer Verfahren (siehe auch Williamson, 2007) ist die wachsende Leistung der eingesetzten Computer ein wesentlicher Treiber

[4] Dies betrifft zumindest die komplexen GCMs und ESMs, die den aktuellen Stand der Wissenschaft abbilden. Es existieren jedoch einige sog. *Simple Climate Models* u.a. in C++ (siehe z.B. Hartin u. a., 2015) oder Python (siehe z.B. pySCM, 2014; Hausfather, 2016; Smith u. a., 2018). Diese dienen im Wesentlichen Lehrzwecken oder um größere Modelle zu emulieren.

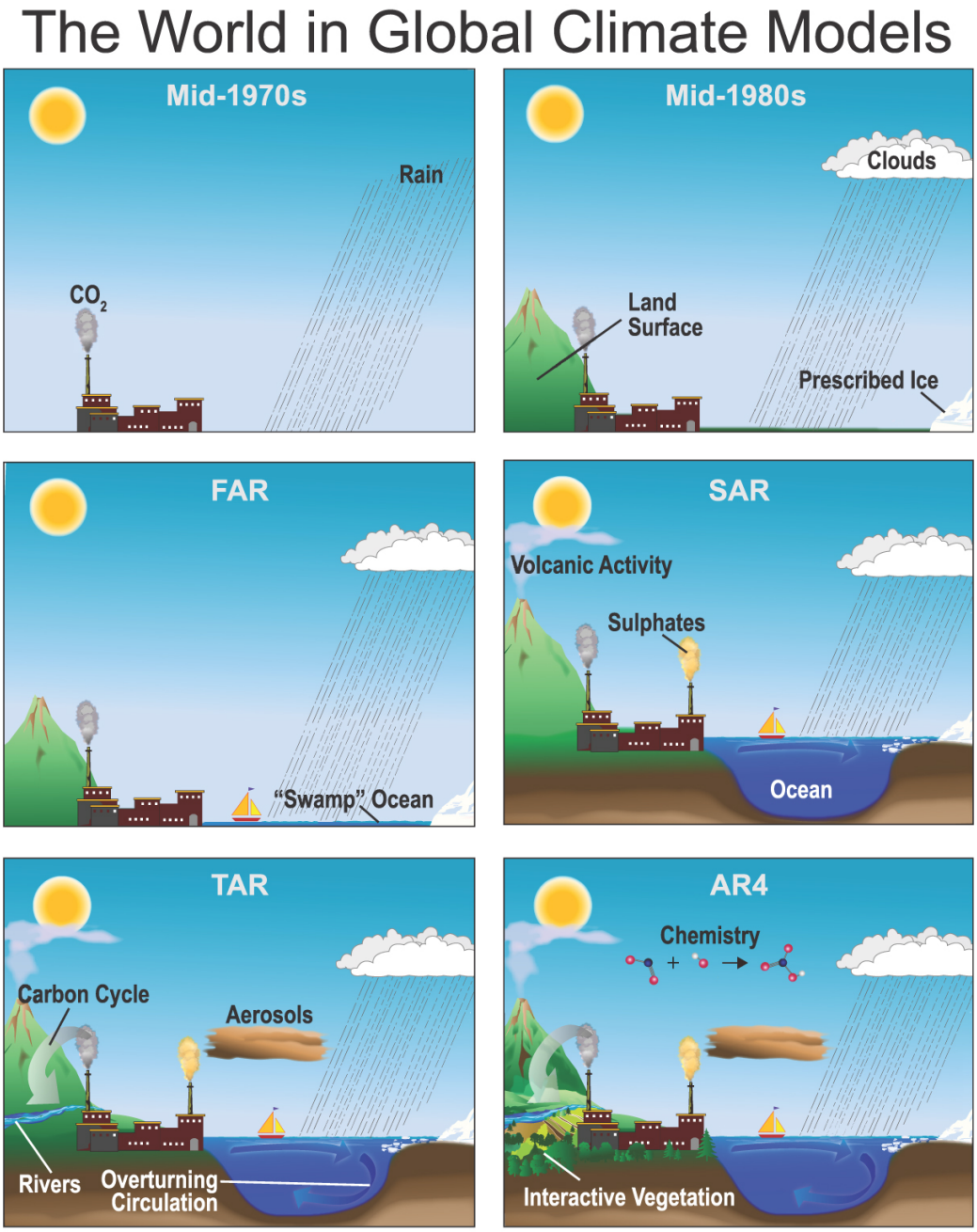


Abbildung 2.8: Historische Entwicklung der in Klimamodellen typischerweise berücksichtigten physikalischen Prozesse (IPCC, 2007). Die Abkürzungen FAR, SAR, TAR und AR4 stehen jeweils für den First, Second und Third Assesment Report bzw. Assesment Report 4 des IPCC.

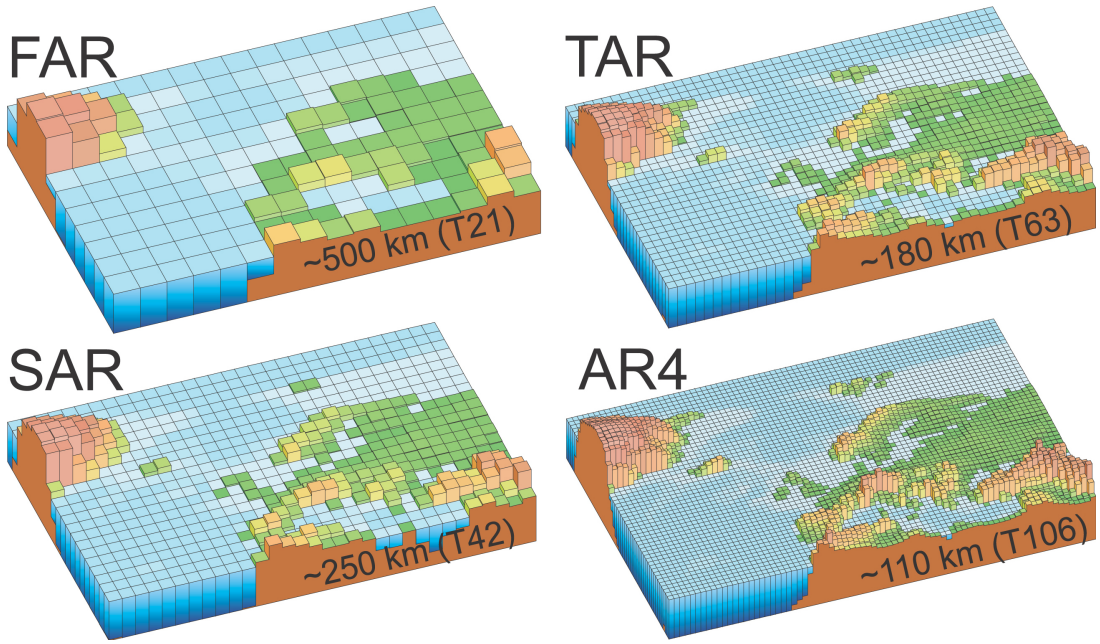


Abbildung 2.9: Historische Entwicklung der typischen Auflösung von Klimamodellen (IPCC, 2007). Abkürzungen wie in Abbildung 2.8.

dieser Entwicklung. Insbesondere die Ablösung spezialisierter Hochleistungsrechner durch massiv-parallele Computer gebaut aus handelsüblichen Komponenten in den 1990er-Jahren hat starken Einfluss auf die Entwicklung von Klimamodellen genommen. Allerdings haben sowohl diese Veränderung als auch die Entwicklung von gekoppelten Modellen die softwaretechnischen Anforderungen der Modellentwicklung stark erhöht:

> *„This led to a dramatic transition in the way climate models were built. Software engineering became a key issue: How do you couple a land model built by one group of scientists to an atmosphere built by a different team, to an ocean built by yet a third, living on another continent, all while respecting numerical constraints on coupling, such as the conservation of mass and energy across the whole system? Extracting performance from chips not primarily built for number crunching also proved to be a daunting task. For the first time, scientists turned to professional programmers to help saddle their beasts.“* (Balaji, 2013)

Aktueller Trend ist der Einsatz von Grafikprozessoren *(graphics processing unit, GPU)* und heterogener Computerarchitekturen. Die wachsende Komplexität auf Software- wie auf Hardwareebene macht eine stärke Zusammenarbeit zwischen WissenschaftlerInnen und anderen ExpertInnen notwendig (vgl. Lawrence, Rezny u. a., 2018).

2.3 Softwarearchitektur

Trotz bestehender Unterschiede lassen sich – neben der Verwendung von Fortran als Hauptprogrammiersprache – weitere Gemeinsamkeiten in der Softwarearchitektur vieler Klimamodelle identifizieren. Diese werden im Folgenden diskutiert.

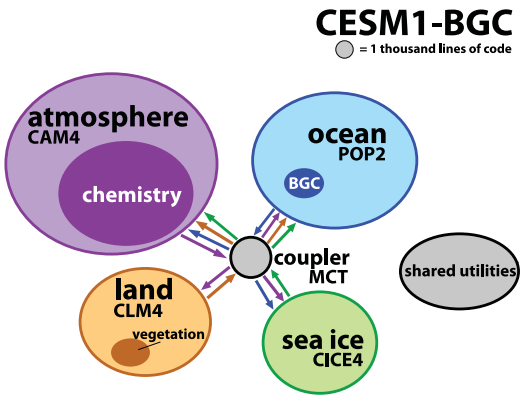
2.3.1 Modularisierung

Klimamodelle zeichnen sich durch eine starke Modularisierung aus, was im Wesentlichen in der interdisziplinären Arbeitsteilung begründet liegt und diese wiederum erst ermöglicht. Gekoppelte GCMs bestehen auf oberster Ebene aus weitgehend unabhängigen Teilmodellen für Atmosphäre, Ozean, Landoberfläche und Meereis. Diese sind in der Regel auch eigenständig ohne die anderen Komponenten lauffähig, wobei einige Landmodelle stärker an ihre jeweiligen Atmosphärenmodelle gekoppelt sind als andere, gleiches gilt für die Verbindung zwischen Eis- und Ozeanmodell. ESMs bestehen aus weitere Teilmodellen, etwa für den Kohlenstoffkreislauf.

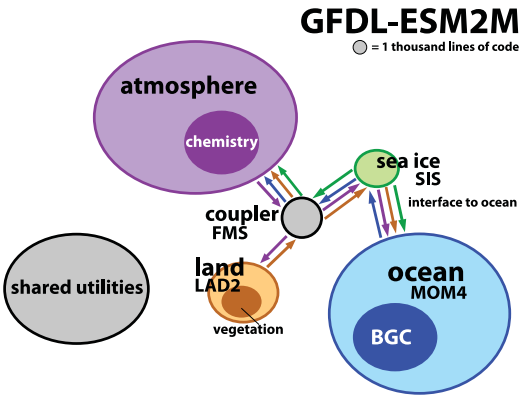
Die Schnittstelle zwischen den Teilmodellen bilden sogenannten *Koppler*. Diese Softwarekomponenten sind für die Umrechnung der *Flüsse* (Energie- und Massentransport) und die Steuerung der Interaktion zwischen den Teilmodellen verantwortlich. Da die Teilmodelle oftmals mit unterschiedlichen Gitterstrukturen arbeiten, müssen vom Koppler Interpolationen vorgenommen werden, um die Flüsse vom einen zum anderen Modell übermitteln zu können.

Grafische Darstellungen der Softwarearchitektur sind eher unüblich in der Klimamodellierungscommunity. Jedoch haben Kaitlin Alexander und Steve Easterbrook (2015) die Architektur einiger gekoppelter Modelle des fünften CMIP-Durchgangs (CMIP5) visualisiert (Abbildung 2.10). In den Darstellungen sind jeweils die bereits erwähnten Teilmodelle und ihr Zusammenspiel über die Koppler-Komponente erkennbar. Wie man sieht, sind Land- und Eismodelle in den amerikanischen Modellen (Abbildungen 2.10a–2.10c) jeweils eigenständige Komponenten und mit Atmosphäre und Ozean über den Koppler verbunden, während in den dargestellten europäischen Modellen (Abbildungen 2.10d–2.10f) das Landmodell vom jeweiligen Atmosphärenmodell gesteuert wird und das Seeeis- durch das Ozeanmodell.

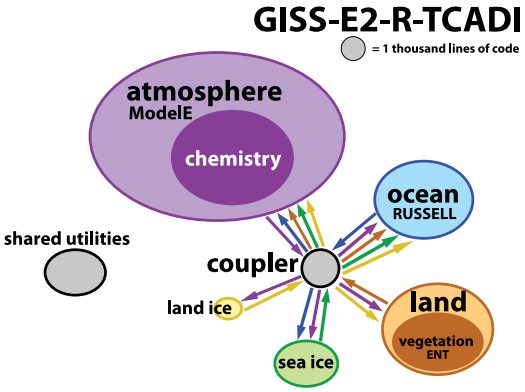
Ebenfalls zu erkennen in den Grafiken von Alexander und Easterbrook ist, dass die meisten Klimamodelle einen nicht unwesentlichen Teil an *Infrastrukturcode (shared utilities)* enthalten. Hierbei handelt es sich meist um Module, die technische Aspekte wie die Interprozesskommunikation oder die Ein- und Ausgabe kapseln, aber auch *numerische Gleichungslöser (solver)*, die von mehreren Teilmodellen verwendet werden.



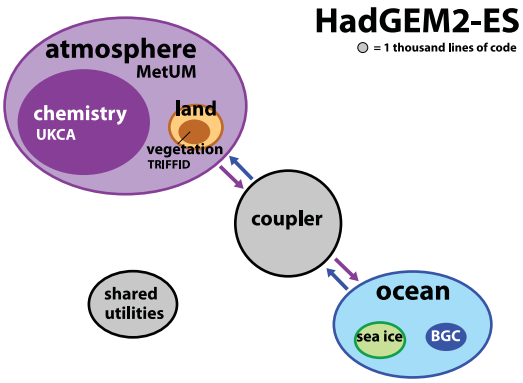
(a) CESM1-BGC vom NCAR, USA



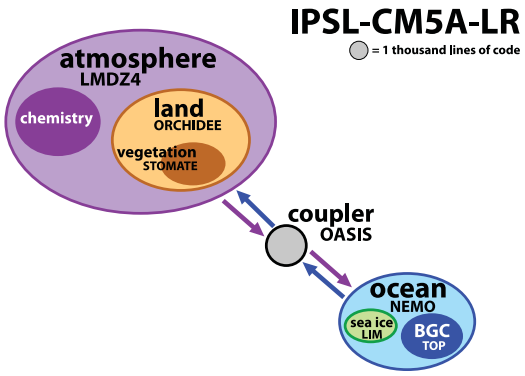
(b) GFDL-ESM2M vom GFDL, USA



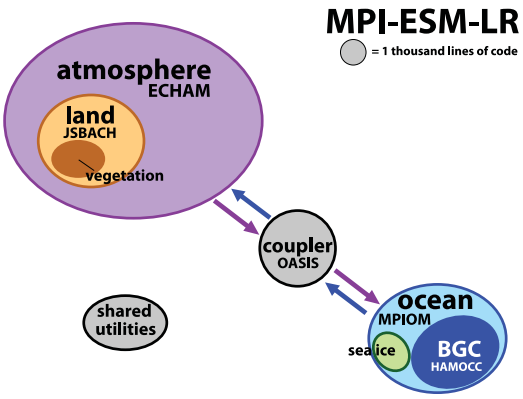
(c) GISS-E2-R-TCADI vom NASA GISS, USA



(d) HadGEM2-ES vom Met Office, UK



(e) IPSL-CM5A-LR vom IPSL, Frankreich



(f) MPI-ESM-LR vom MPI-M, Deutschland

Abbildung 2.10: Architekturdiagramme verschiedener Klimamodelle (Alexander und Easterbrook, 2015). Komponenten/Teilmodelle sind durch Ellipsen dargestellt, die Datenflüsse zwischen den Komponenten durch Pfeile, jeweils in der Farbe ihrer Ursprungskomponente. Die Komponenten sind in der Größe proportional zur Codegröße dargestellt, jedoch nur innerhalb eines Modells, zwischen den Modellen ist die Darstellung nicht proportional. Zur Skalierung ist jeweils eine Vergleichsfläche entsprechend 1.000 Zeilen Code angegeben.

Abhängigkeiten zu solchem Infrastrukturcode können den Austausch einzelner Modellkomponenten zwischen den Forschungsinstituten erschweren. Dieser ist nicht unüblich, häufig aber mit umfangreichen Anpassungen verbunden (vgl. Edwards, 2010; Alexander und Easterbrook, 2015). Es gab in der Vergangenheit Bemühungen einheitliche Frameworks über Institutsgrenzen hinaus zu schaffen (siehe z.B. Hill u. a., 2004; PRISM, 2015), jedoch haben sich diese bis heute nicht durchsetzen können. Die meisten größeren Institute pflegen eigene Infrastrukturbibliotheken und -frameworks.

Auch innerhalb der jeweiligen Teilmodelle lassen sich wiederkehrende modulare Strukturen finden. So bestehen Atmosphären- und Ozeanmodelle in der Regel aus einem *dynamischen Kern* und der sogenannten *Physik*. Im dynamischen Kern sind die grundlegenden Gleichungen der Hydro- und Thermodynamik zur Beschreibung der allgemeinen Zirkulation implementiert. Die Physik beschreibt die physikalischen Prozesse, die durch diese Gleichungen nicht abgedeckt werden, wie etwa der Strahlungshaushalt, Konvektion oder Diffusion. Einige davon können mathematisch sehr genau beschrieben werden, andere nur parametrisiert werden. Oftmals existierende für einzelne Aspekte der Physik mehrere austauschbare Implementationen. Welche jeweils verwendet wird, wird über die Modellkonfiguration festgelegt.

Zwar sind Klimamodelle häufig stark modularisiert, dabei wird jedoch nicht immer das *Geheimnisprinzip (information hiding)* beachtet. Dieses besagt, dass Module einzelne Designentscheidungen kapseln und diese untereinander verbergen sollen (vgl. Parnas, 1972). Innerhalb von Klimamodellen gibt es jedoch häufig indirekte und implizite Abhängigkeiten zwischen einzelnen Modulen, beispielweise über die exzessive Verwendung von globalen Variablen oder Annahmen zur Reihenfolge oder Art und Weise bestimmter Berechnungsschritte. Teilweise sind diese gegenseitigen Abhängigkeiten fachlich begründet, da im Klimasystem viele Wechselwirkungen zwischen einzelnen Komponenten und Prozesse existieren.

2.3.2 Konfigurationen

Klimamodelle sind in hohem Maße konfigurierbar (siehe auch Abschnitt 2.1.7). Mögliche Konfigurationsoptionen sind z.B.:

- Simulationszeitraum und Zeitschrittlänge
- Aktivierung einzelner Teilmodelle
- Auswahl einzelner alternativer bzw. optionaler Physikimplementationen
- Gitterauflösung
- Pfade für Dateien mit Gitterinformationen, Anfangs- und Randbedingungen

- Konstante Parameter
- Gewünschte Ausgabe (Variablen und Zeitpunkte)
- Technologieauswahl zur Anpassung an die jeweilige Rechnerplattform

Einige dieser Konfigurationen finden bereits zur Übersetzungszeit statt, wobei *Compiler-Flags* zum Einsatz kommen, die vom *Präprozessor* (siehe auch Abschnitt 2.5.1) ausgewertet werden, andere Konfigurationen werden zur Laufzeit eingelesen, beispielsweise mit Hilfe von Namelists (siehe Abschnitt 2.4.8) oder XML-Dateien. Die starke Konfigurierbarkeit ist neben den komplizierten physikalisch-mathematischen Strukturen eine der wesentlichen Quellen von Komplexität innerhalb von Klimamodellen. So bezieht sich ein Großteil der Fallunterscheidungen im Code von Klimamodellen auf Konfigurationsoptionen.

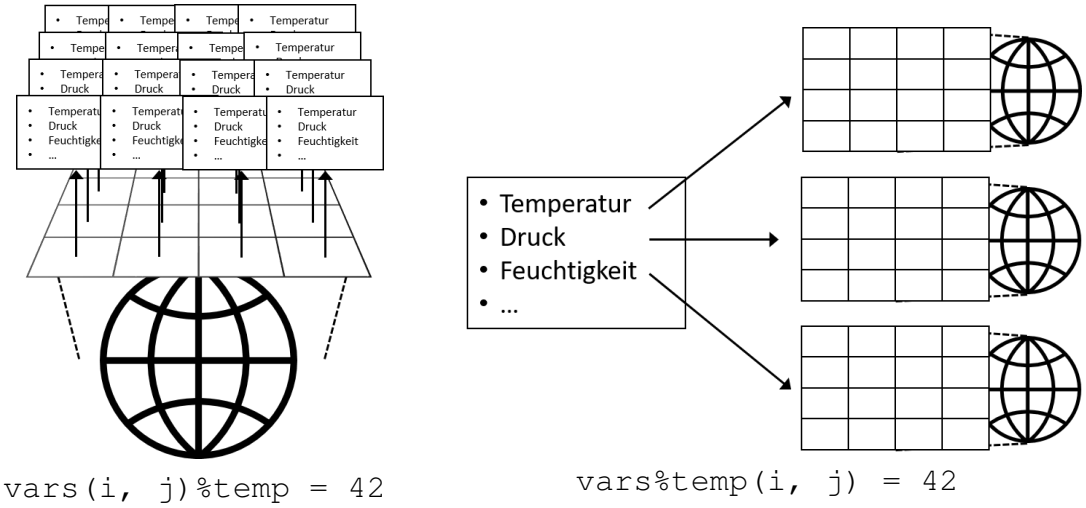
Viele Konfigurationsoptionen dienen auch der *Portabilität* zwischen mehreren Rechnerplattformen. Je nach Plattform oder Compiler werden ggf. andere Bibliotheken eingebunden, Datenstrukturen an Cachegrößen angepasst, Arraydimensionen vertauscht oder ganze Module oder Prozeduren ausgetauscht.

2.3.3 Kontrollfluss und Datenstrukturen

Der Kontrollfluss eines Klimamodells besteht im Wesentlichen aus einer Schleife, in der von einem Zeitschritt zum nächsten die Zustandsvariablen des jeweiligen Teilmodells (z.B. der Atmosphäre) neu berechnet werden (siehe auch Abschnitt 2.1.7). Innerhalb eines Zeitschritts werden nacheinander die Routinen der einzelnen Modellkomponenten aufgerufen. Die Reihenfolge, in der dies geschieht, wird dabei sowohl fachlich wie auch technisch bestimmt.

Die Zustandsvariablen des Modells liegen in mehrdimensionalen Arrays vor, welche i.d.R. alle Werte der dreidimensionalen Domäne enthalten. Dabei gibt es eine eindeutige Abbildung von den Gitterpunkten des Modells auf die Indizes der Arrays. Bei kartesischen Gittern (Abbildung 2.2), ergibt sich diese recht natürlich: unter Berücksichtigung der Gitterauflösung entspricht dabei eine Dimension der geographischen Breite, eine der Länge und eine der Höhe. Nachbarpunkte lassen sich durch Inkrementierung bzw. Dekrementierung eines Index finden. Bei anderen Gittern (Abbildung 2.3) ist diese Abbildung komplizierter.

Mit Hilfe von benutzerdefinierten *Verbunddatentypen* lassen sich in Fortran mehrere zusammengehörige Variablen gruppieren (siehe auch Abschnitt 2.4.5). Dabei gibt es grundsätzlich zwei Möglichkeiten der Anordnung von Arrays. Bei der ersten Variante, auch *Array of Structs* genannt (Abbildung 2.11a), wird eine Datenstruktur definiert, welche die Einzelwerte der Zustandsvariablen einer Gitterzelle zusammenfasst, und



(a) Array of Structs

(b) Struct of Arrays

Abbildung 2.11: Mögliche Anordnung von Arrays in Verbunddatentypen.

dann mehrere Exemplare dieser Datenstruktur in einem Array angeordnet. Bei der zweiten Variante, *Struct of Arrays* (Abbildung 2.11b), existiert nur ein Exemplar der Datenstruktur, welche wiederum mehrere Arrays, für jede Zustandsvariable eines, enthält.

In Atmosphären- und Ozeanmodellen ist, auch aus Performancegründen, die zweite Variante üblicher. In sehr vielen Routinen finden sich Schleifen über sämtliche Gitterpunkte, um die Werte einer Variable für den gesamten Globus bzw. den Ausschnitt, für den der aktuelle Prozess zuständig ist, zu berechnen und zu aktualisieren. Dabei werden häufig auch die entsprechenden Daten von Nachbarpunkten benötigt. Dieser Aufbau macht es schwieriger, zu Testzwecken die Gleichungen des Modells für nur einen einzigen Gitterpunkt bzw. eine Gitterzelle oder eine Säule zu berechnen. Dennoch bieten einige Modelle diese Möglichkeit.

In Landmodellen steht hingegen weniger die Wechselwirkung zwischen benachbarten Zellen im Vordergrund, dafür umso mehr das unterschiedliche Verhalten verschiedener Oberflächentypen und deren Wechselwirkung mit der Atmosphäre. In einigen Modellen lässt sich daher eine eher objektorientierte Struktur nach dem Array-of-Structs-Schema wiederfinden.

2.3.4 Parallelisierung

In Klimamodellen finden sich zwei Arten von Parallelisierung: *Domain Decomposition* und *Task Parallelisation*. Domain Decomposition bedeutet, dass die Domäne,

d.h. die Menge der Gitterzellen bzw. -punkte gleichmäßig auf einzelne Prozesse aufgeteilt wird. Jeder Prozess führt dieselben Berechnungen für die ihm zugewiesene, i.d.R. zusammenhängende Teilmenge an Gitterzellen/-punkten durch. Für Berechnungen, die Daten aus Nachbarzellen benötigen, müssen Prozesse ggf. miteinander kommunizieren, um Daten untereinander auszutauschen.

Bei der Task Parallelisation, auch *Functional Decomposition* genannt, werden unterschiedliche Berechnungen parallel ausgeführt. Diese Form der Parallelisierung findet in Klimamodellen meist auf Ebene der gekoppelten Teilmodelle statt, z.B. indem das Atmosphärenmodell parallel zum Ozeanmodell läuft. Es gibt allerdings auch einzelne Bemühungen diese Form der Parallelisierung innerhalb von Teilmodellen zu ermöglichen (siehe z.B. Balaji, Benson u. a., 2016; Behrens u. a., 2018).

2.3.5 I/O

Ein nicht unwesentlicher Teil des Codes in Klimamodellen dient der Steuerung der Ein- und Ausgabe *(input/output, I/O)* von Modelldaten. Zu den Eingabedaten gehören u.a. Gitterstrukturen sowie Anfangs- und Randbedingungen. Diese werden in der Regel zum Start einer Simulation eingelesen. Zudem müssen Antriebskraftdaten auch während einer Simulation eingelesen werden.

Auch wenn *Datenassimilation* zum Einsatz kommt, müssen in regelmäßigen Abständen während einer Simulation Daten eingelesen werden. Datenassimilation dient der Korrektur der Simulationsergebnisse mit Hilfe von Beobachtungsdaten und kommt hauptsächlich in der numerischen Wettervorhersage zum Einsatz. Es gibt jedoch auch Anwendungsfälle in der Klimamodellierung (vgl. Carrassi u. a., 2018).

Die Ausgabe der Modelldaten erfolgt am Ende der Simulation und währenddessen zu (in der Konfiguration) festgelegten Zeitschritten. Neben den Daten für die wissenschaftlichen Auswertung der Simulation werden in regelmäßigen Abständen (*Checkpoints*) auch sog. *Restart*-Dateien geschrieben. Diese enthalten sämtliche Daten, um ein Modell wieder in den gleichen Zustand wie zum Ausgabezeitpunkt zu versetzen. Dies ermöglicht es, unterbrochene Simulationen fortsetzen zu können, ohne ganz von vorn beginnen zu müssen. Unterbrechungen können beispielsweise durch Hardware- oder Softwaredefekte entstehen oder durch die Rechenzentrumsadministration veranlasst werden.

2.4 Fortran

Alle großen Klimamodelle sind in *Fortran* geschrieben (siehe auch Abschnitt 2.2.3). Dies hat verschiedene Gründe. Zum einen ist es historisch bedingt und die Sprache in

der Community etabliert. Dies schlägt sich im vorhandenen Code wieder, der selten komplett durch neuen abgelöst wird, sondern sich mit der Zeit weiterentwickelt, und in den Programmierkenntnissen der EntwicklerInnen.

Zum anderen hat Fortran einige Vorteile gegenüber anderen Programmiersprachen. Grundsätzlich stehen im Hochleistungsrechnen aufgrund der Unterstützung durch Rechner- und Compilerhersteller und der erreichbaren Leistung nicht sehr viele Sprachen zur Verfügung, im Wesentlichen nur Fortran sowie C und C++ (vgl. Basili u. a., 2008). Fortran kommt dabei fast ausschließlich in diesem Bereich zum Einsatz und ist daher auch speziell auf numerische Anwendungen ausgerichtet, während C und C++ ihren Hintergrund eher in der Systemprogrammierung haben. Insbesondere bietet Fortran eine einfache Syntax für den Umgang mit (mehrdimensionalen) Arrays und für arithmetische Operationen. Seit dem Standard von 2003 bietet Fortran auch Sprachkonstrukte für die objektorientierte Programmierung.

Fortran gilt als die erste realisierte und breit eingesetzte höhere Programmiersprache der Welt. Sie entstand bei IBM auf Basis einer Idee von John Backus und wurde erstmals 1957 veröffentlicht (vgl. Neumann, 2016). Seitdem ist die Sprache weiterentwickelt und immer wieder neu standardisiert worden. So existieren die Standards FORTRAN 66, FORTRAN 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008 und Fortran 2018 (vgl. Metcalf, 2011; Neumann, 2018). Jedoch verhält sich kaum ein Fortran-Compiler standardkonform. So unterstützen bis heute nicht alle Compiler sämtliche Features der Standards 2003 und 2008 (vgl. Fortran Wiki, 2019a; Fortran Wiki, 2019b). Außerdem werden Sprachkonstrukte, die laut aktuellem Standard obsolet sind, weiterhin von den meisten Compilern unterstützt. In Klimamodelle findet sich ein breites Spektrum von alten und neuen Sprachkonstrukten (vgl. Méndez, Tinetti und Overbey, 2014).

Im Fortran-Universum haben einige Begriffe eine andere Bedeutung als man es von anderen Programmiersprachen gewohnt ist (siehe auch Abschnitt 1.6). Im Folgenden werden die für diese Arbeit relevanten Fortran-Begriffe sowie grundlegende Sprachkonzepte erläutert.

2.4.1 Prozeduren

Fortran gehört zu den *prozeduralen Programmiersprachen*. Fortran-Systeme sind also in *Prozeduren* organisiert, wobei zwischen Subroutinen und Funktionen unterschieden wird. Als Subroutinen werden Prozeduren ohne Rückgabewert bezeichnet, als Funktionen solche mit Rückgabewert.

Parameter, also Werte, die an eine Prozedur übergeben werden, heißen in Fortran Argumente, wobei die formalen Parameter, also die Platzhaltervariablen innerhalb

einer Prozedur, als Dummyargumente bezeichnet werden und die aktuellen Parameter, d.h. die Ausdrücke, die beim Aufruf an die Dummyargumente gebunden werden, einfach als Argumente. Argumente im weiteren Sinne sind also aktuelle und/oder formale Parameter, Argumente im engeren Sinnen meinen aktuelle Parameter. Im weiteren Sinne wird der Begriff in dieser Arbeit nur verwendet, wenn die Bedeutung im jeweiligen Kontext eindeutig ist.

Argumente werden in Fortran grundsätzlich per Referenz übergeben. Dummyargumente können mit dem `INTENT`-Schlüsselwort als Eingangsvariable (`in`), Ausgangsvariable (`out`) oder Zweiwegevariable (`inout`) definiert werden. Entsprechend heißen diese auch In-Argumente, Out-Argumente oder In/Out-Argumente. Mit Hilfe von Out- und In/Out-Argumenten können auch Subroutinen Werte zurückliefern. Mit dem Schlüsselwort `OPTIONAL` können Dummyargumente als optional deklariert werden.

Abbildung 2.12 zeigt den typischen Aufbau einer Prozedur am Beispiel einer Subroutine. Alle Variablen, egal ob Dummyargumente oder herkömmliche lokale Variablen, müssen zu Beginn einer Prozedur deklariert werden. Danach folgen die Anweisungen der Prozedur.

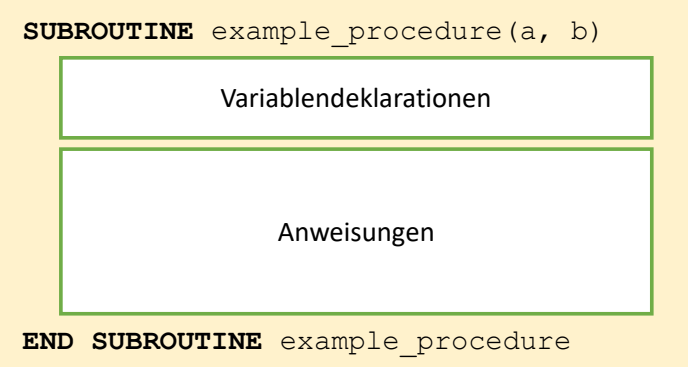


Abbildung 2.12: Aufbau einer Prozedur (hier Subroutine)

Beispiel 2.1: Funktion

```
1  FUNCTION erhoehe_func(akk, delta)
2
3    REAL, INTENT(inout) :: akk
4    REAL, INTENT(in) :: delta
5    REAL :: erhoehe_func
6
7    akk = akk + delta
8    erhoehe_func = delta / akk
9
10 END FUNCTION erhoehe_func
```

In der FUNKTION `erhoehe_func` wird eine Variable `akk`, die als IN/OUT-ARGUMENT übergeben wird, um den Wert des IN-ARGUMENTS `delta` erhöht. Der Quotient aus `delta` und dem neuen Wert von `akk` wird als Ergebnis der FUNKTION zurückgegeben.

Beispiel 2.2: Subroutine

```
1  SUBROUTINE erhoehe_subr(akk, delta, rel)
2
3    REAL, INTENT(inout) :: akk
4    REAL, INTENT(in) :: delta
5    REAL, INTENT(out) :: rel
6
7    akk = akk + delta
8    rel = delta / akk
9
10 END SUBROUTINE erhoehe_subr
```

Die SUBROUTINE `erhoehe_subr` verhält sich ähnlich wie die vorangegangen FUNKTION. Der Quotient aus `delta` und dem neuen Wert von `akk` ist hier jedoch nicht der Funktionsrückgabewert, sondern wird in dem OUT-ARGUMENT `rel` gespeichert.

Beispiel 2.3: Aufruf von Funktion und Subroutine

```
1  REAL :: akk_f, akk_s
2  REAL :: rel_f, rel_s
3
4  akk_f = 42.0
5  rel_f = erhoehe_func(akk_f, 23.0)
6  PRINT *, akk_f, rel_f                  !   65.0000000      0.353846163
7
8  akk_s = 109.0
9  CALL erhoehe_subr(akk_s, 23.0, rel_s)
10 PRINT *, akk_s, rel_s                  !   132.000000      0.174242422
```

Zu sehen ist die unterschiedliche Syntax für den Aufruf von FUNKTIONEN und SUBROUTINEN. Der Rückgabewert der FUNKTION `erhoehe_func` wird per Zuweisung in der Variable `rel_f` gespeichert (Zeile 5). Das Ergebnis der SUBROUTINE `erhoehe_subr`, die in Zeile 9 mit dem Schlüsselwort `CALL` aufgerufen wird, wird in der als ARGUMENT übergebenen Variable `rel_s` gespeichert. Beide Varianten speichern zusätzlich einen Wert im jeweils ersten DUMMYARGUMENT, welches in beiden Fällen als IN/OUT-ARGUMENT deklariert ist.

Lokale Variablen, die innerhalb von Prozeduren deklariert werden, existieren während der Ausführung ihrer Prozedur und verschwinden, sobald eine Ausführung abgeschlossen ist. Mit dem Schlüsselwort SAVE lassen sich jedoch auch lokale Variablen deklarieren, die ihren Wert über einzelne Ausführungen hinaus behalten. Wird eine lokale Variable bereits bei ihrer Deklaration initialisiert, ist die SAVE-Eigenschaft implizit.

Beispiel 2.4: SAVE-Variable

```
1  FUNCTION zaehle()
2
3    INTEGER, SAVE :: z = 0
4    INTEGER :: zaehle
5
6    z = z + 1
7    zaehle = z
8
9  END FUNCTION zaehle
```

Bei ihrer ersten Ausführung würde die FUNKTION zaehle den Wert 1 liefern. Da die SAVE-Variable z danach ihren Wert behält, wäre das Ergebnis der nächsten Ausführung 2. Da z bereits bei der Deklaration mit dem Wert 0 initialisiert wird, ist die explizite Angabe des Schlüsselworts SAVE hier optional.

2.4.2 Module

Variablen, Prozeduren und benutzerdefinierte Typen (siehe Abschnitt 2.4.5) lassen sich in *Modulen* zusammenfassen. Module dienen in erster Linie der Gruppierung und Kapselung derartiger Programmelemente. Zur Kapselung kann der Zugriff auf Modulinhalte durch andere Programmteile eingeschränkt werden (vollständig: PRIVATE, nur Schreibzugriff verbieten: PROTECTED) bzw. gezielt einzelne (oder alle) Modulinhalte exportiert werden (PUBLIC). Mit dem Schlüsselwort USE lassen sich Elemente aus anderen Modulen importieren. Abbildung 2.13 zeigt den typischen Aufbau eines Fortran-Moduls.

Module enthalten i.d.R. zwischen USE-Anweisungen und Deklarationsblock die Anweisung IMPLICIT NONE. Dies erzwingt, dass alle verwendeten Variablen deklariert werden müssen. Standardmäßig ist Fortran sonst in der Lage, anhand des ersten Buchstabens des Bezeichners nach festgelegten Regeln (z.B. i für INTEGER) den Typ einer Variable abzuleiten. Dies kann jedoch leicht zu Fehlern führen, daher gilt es als guter Stil, dieses Verhalten mit Hilfe von IMPLICIT NONE zu deaktivieren (vgl. Brainerd, 2009, S. 13).

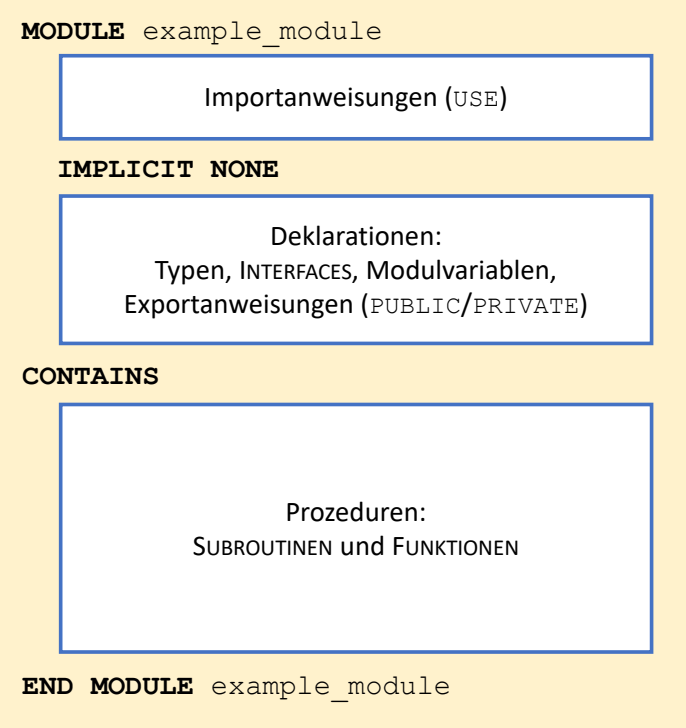


Abbildung 2.13: Aufbau eines Fortran-Moduls

Beispiel 2.5: Modul

```
1  MODULE hello
2
3    IMPLICIT NONE
4    PRIVATE
5
6    CHARACTER(16) :: name
7    PUBLIC :: greet
8
9  CONTAINS
10
11   SUBROUTINE greet()
12     PRINT *, 'Hello ', name
13   END SUBROUTINE greet
14
15 END MODULE hello
```

Das Modul `hello` enthält eine MODULVARIABLE `name` (Zeile 6) und eine SUBROUTINE `greet` (Zeilen 11–13), in der der Inhalt der Variable `name` mit Hilfe der `PRINT`-Anweisung ausgegeben wird. Das Schlüsselwort `PRIVATE` in Zeile 4 gibt an, dass die Inhalte von `hello` grundsätzlich von außen nicht zugreifbar sind. Mit Hilfe des Schlüsselworts `PUBLIC` (Zeile 7) wird die SUBROUTINE `greet` exportiert.

```
16 MODULE other
17
18   USE hello, ONLY: greet
19
20   IMPLICIT NONE
21
22   ⋮
23
24 END MODULE other
```

Im Modul `other` wird nun mit Hilfe der `USE`-Anweisung die SUBROUTINE `greet` aus dem Modul `hello` importiert und steht somit in `other` zur Verfügung.

2.4.3 Dynamische Speicherverwaltung

In der Regel steht die Größe von Variablen in Fortran bereits zur Übersetzungszeit fest. Es lassen sich jedoch auch Arrayvariablen deklarieren, deren Größe erst zur Laufzeit festgelegt wird und deren Speicher entsprechend dynamisch zugewiesen wird.

Beispiel 2.6: Dynamische Zuweisung von Speicher

```
1 INTEGER, ALLOCATABLE :: dyn(:)
2 INTEGER :: g
3
4 g = groesse_berechnen()
5 ALLOCATE(dyn(g))
```

Die Arrayvariable `dyn` wird hier mit dem Schlüsselwort `ALLOCATABLE` deklariert (Zeile 1). Dies bewirkt, dass die Größe zum Zeitpunkt der Deklaration noch nicht feststeht. Mit Hilfe einer hier nicht näher definierten FUNKTION `groesse_berechnen` wird die Größe, die das Array einnehmen soll, bestimmt (Zeile 4) und `dyn` schließlich der entsprechende Speicher zugewiesen (Zeile 5).

2.4.4 Zeiger

Fortran erlaubt es Aliase für Variablen in Form von *Zeigern* zu erstellen. Dazu muss eine Zeigervariable deklariert und mit einer anderen Variable verknüpft werden. Zeiger können drei Zustände annehmen:

undefiniert Diesen Zustand nimmt eine uninitialisierte Zeigervariable ein. Er kann auch eintreten, wenn eine Variable, mit der ein Zeiger verknüpft ist, verschwindet. Wenn beispielsweise ein Zeiger, der auf Modulebene deklariert ist, innerhalb einer Prozedur mit einer lokalen Variable verknüpft wird, ist sein Zustand nach Beendigung der Prozedur undefiniert.

unverknüpft Man kann eine Zeigervariable explizit mit dem speziellen Wert `NULL()` verknüpfen, dann gilt diese als unverknüpft.

verknüpft Eine Zeigervariable ist mit einer anderen Variable verknüpft.

Der Status eines Zeigers lässt sich mit der eingebauten Funktion `ASSOCIATED` abfragen. Diese liefert alle allerdings nur verlässliche Ergebnisse, wenn der Zustand des Zeigers entweder verknüpft oder unverknüpft ist. Für Zeiger im undefinierten Zustand ist auch das Ergebnis der Funktion undefiniert.

Beispiel 2.7: Zeiger

```
1 INTEGER, DIMENSION(:,:), POINTER :: p
2 INTEGER, DIMENSION(4,2), TARGET :: t
3
4 PRINT *, ASSOCIATED(p)                ! ?
5 p => NULL()
6 PRINT *, ASSOCIATED(p)                ! F
7 p => t
8 PRINT *, ASSOCIATED(p)                ! T
```

Mit Hilfe des Schlüsselworts `POINTER` wird die Zeigervariable `p` deklariert. Zunächst ist ihr Zustand undefiniert, die Funktion `ASSOCIATED` liefert noch kein verlässliches Resultat (Zeile 4). Nach der Initialisierung mit `NULL()` liefert `ASSOCIATED(p) .FALSE.` (Zeile 6). Anschließend wird `p` mit der Variable `t` verknüpft, `ASSOCIATED(p)` liefert nun `.TRUE.` (Zeile 8).

2.4.5 Verbunddatentypen

Fortran ist eine streng statisch getypte Programmiersprache. Zusätzlich zu den in die Sprache eingebauten *primitiven Datentypen* (intrinsic type) wie etwa `INTEGER` für Ganzzahlen oder `REAL` für Fließkommazahlen, lassen sich auch benutzerdefinierte Verbunddatentypen (derived types), ähnlich der *Structs* in C, definieren. Damit lassen sich mehrere Datenfelder, Komponenten genannt, zu einer gemeinsamen Datenstruktur zusammenfassen. Die einzelnen Komponenten können dabei entweder einen primitiven Typ oder wiederum einen Verbunddatentyp haben. Der Zugriff auf die einzelnen Komponenten erfolgt mit dem `%`-Operator.

Beispiel 2.8: Verbunddatentyp

```
1  TYPE A
2    REAL :: r
3    LOGICAL :: l
4  END TYPE A
5
6  TYPE B
7    INTEGER :: i
8    TYPE(A) :: a
9  END TYPE B
```

Typ A enthält zwei Komponenten mit den primitiven Datentypen REAL und LOGICAL. Typ B hat eine Komponente des primitiven Typs INTEGER und eine Komponente mit zuvor definierten Verbunddatentyp A.

```
10 TYPE(B) :: b1
11
12 b1%i = 42
13 b1%a%r = 10.9
14 b1%a%l = .TRUE.
```

Hier wird eine Variable b1 mit dem Typ B deklariert und anschließend die einzelnen Komponenten mit Werten belegt.

Die Struktur eines Verbunddatentyps lässt sich auch als Baum darstellen, dessen Blätter die Komponenten mit primitiven Typen sind (Abbildung 2.14). Diese primitiven Komponenten-„Blätter“, hier i, r und l, werden in dieser Arbeit Basiskomponenten genannt. Sie tragen die eigentlichen Nutzdaten der Datenstruktur. Dagegen werden alle Komponenten, die selbst einen Verbunddatentyp haben, wie etwa die Komponente a, Objektkomponenten genannt[5].

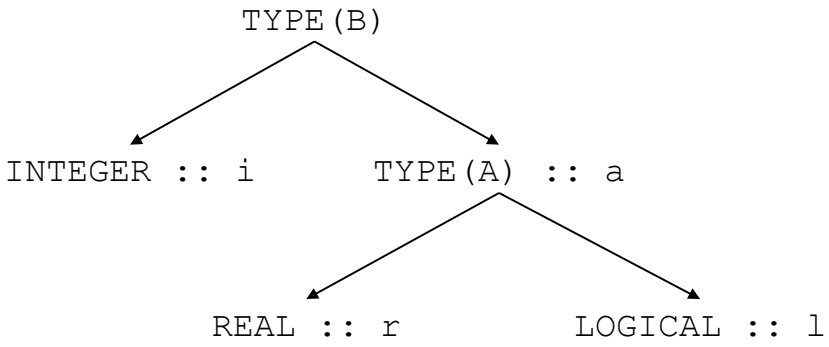


Abbildung 2.14: Baumstruktur der Verbunddatentypen aus Beispiel 2.8

[5]Die Begriffe Basis- und Objektkomponenten sind hier selbst gewählt. Im Fortran-Standard (ISO/IEC DIS 1539-1, 2017) sind mit ultimate components und potential subobject components vergleichbare, jedoch nicht exakt dieselben Begriffe definiert.

Fortran erlaubt einen objektorientierten Programmierstil, u.a. indem Verbunddatentypen Prozeduren zugewiesen werden können. Solche Prozeduren werden TYPGEBUNDENE PROZEDUREN genannt. Diese können ebenfalls mit dem %-Operator an Exemplaren des jeweiligen Typs aufgerufen werden.

Beispiel 2.9: Typgebundene Prozeduren

```
1  MODULE typebound
2
3    IMPLICIT NONE
4
5    TYPE :: C
6      INTEGER :: counter = 0
7    CONTAINS
8      PROCEDURE :: count => increment_c
9    END TYPE C
10
11 CONTAINS
12
13   SUBROUTINE increment_c(this)
14     CLASS(C), INTENT(inout) :: this
15     this%counter = this%counter + 1
16   END SUBROUTINE increment_c
17
18 END MODULE typebound
```

Im Modul `typebound` wird ein Typ `C` definiert, der eine TYPGEBUNDENE PROZEDUR `count` enthält (Zeilen 5–9). `count` ist ein Alias für die SUBROUTINE `increment_c`, die im selben Modul definiert wird (Zeilen 13–16). Das erste DUMMYARGUMENT von `increment_c` muss dabei zwingend vom Typ `CLASS(C)` sein.

```
19 TYPE(C) :: cvar
20 CALL cvar%count()
```

`count` kann nun an einer Variable vom Typ `C` oder einem Untertyp (siehe Abschnitt 2.4.6) aufgerufen werden. Diese wird dabei automatisch als erstes ARGUMENT an `increment_c` übergeben.

2.4.6 Vererbung

Verbunddatentypen können von anderen Verbunddatentypen abgeleitet werden. Der abgeleitete Typ erbt alle Eigenschaften des Obertyps und kann eigene hinzufügen.

Beispiel 2.10: Vererbung

```
1  TYPE, EXTENDS(C) :: D
2    LOGICAL :: active
3  END TYPE D
```

Mit Hilfe des Schlüsselworts EXTENDS wird ein Typ D definiert, der vom Typ C aus Beispiel 2.9 erbt.

```
4  SUBROUTINE cond_incr(darg)
5    TYPE(D), INTENT(inout) :: darg
6    IF (darg%active) THEN
7      CALL darg%count()
8    END IF
9    PRINT *, darg%counter
10 END SUBROUTINE cond_incr
```

Die Subroutine cond_incr enthält ein Dummyargument darg vom Typ D. Abhängig vom Zustand der in D definierten Komponente active wird die vom Typ C geerbte typgebundene Prozedur count an darg aufgerufen (Zeile 7). Anschließend wird der Wert der ebenfalls von C geerbten Komponente counter ausgegeben (Zeile 9).

Zudem ist es möglich, Typen als ABSTRACT zu deklarieren. Von ihnen können keine konkreten Variablen erzeugt werden, jedoch können diese als Typen von Zeigern und ALLOCATABLE-Variablen sowie von Dummyargumenten dienen. Um diese Variablen als *polymorph* zu kennzeichnen, muss bei ihrer Deklaration anstelle von TYPE das Schlüsselwort CLASS verwendet werden. Zur Laufzeit muss diesen Variablen dann ein konkreter Untertyp des abstrakten Typs zugewiesen werden.

Beispiel 2.11: Abstrakter Typ

```
1  TYPE, ABSTRACT :: E
2    INTEGER :: id
3  END TYPE E
4
5  TYPE, EXTENDS(E) :: F
6    CHARACTER(len=16) :: name
7  END TYPE F
```

Ein abstrakter Typ E wird deklariert (Zeilen 1–3) sowie ein konkreter Typ F, der von E erbt (Zeilen 5–7).

```
8  CLASS(E), ALLOCATABLE :: e1
9  ALLOCATE(F::e1)
```

Hier wird eine polymorphe ALLOCATABLE-Variable e1 vom Typ E deklariert.

Anschließend wird bei der Speicherzuweisung `F` als konkreter Typ von `e1` festgelegt.

2.4.7 Parameter

Während Parameter in Fortran Argumente heißen, werden Konstanten, also unveränderliche Variablen, wiederum Parameter genannt.

Beispiel 2.12: Definition eines Parameters

```
1 INTEGER, PARAMETER :: foo = 42
```

2.4.8 Namelists

Fortran bietet die Möglichkeit Variablen in sog. Namelists zu gruppieren, um deren Werte anschließend gebündelt einzulesen, beispielsweise aus einer Datei.

Beispiel 2.13: Namelist

```
1  INTEGER :: foo
2  REAL :: bar, baz
3
4  NAMELIST /beispiel/ foo, bar, baz
5
6  OPEN(1,file='bsp1.dat')
7  READ(1,nml=beispiel)
8  CLOSE(1)
```

Eine Namelist `beispiel` mit drei Variablen wird definiert. Die Inhalte werden anschließend aus der Datei `bsp1.dat` eingelesen werden.

```
9  &beispiel
10   foo = 42
11   bar = 23.0
12   baz = 109.47
13 /
14
```

Inhalt der Datei `bsp1.dat`.

Klimamodelle verwenden häufig Namelists, um Konfigurationen einzulesen. Auch wenn die auf diese Weise gesetzten Variablen technisch gesehen keine Konstanten sind, werden sie meist so verwendet und umgangssprachlich auch als Namelist-Parameter bezeichnet.

2.4.9 Prozedur- und Operatorüberladung

Prozeduren können in Fortran auch überladen werden, d.h. mehrere Prozeduren mit unterschiedlichen Dummyargumenten können über denselben Bezeichner aufgerufen werden. Hierzu muss ein sog. Interface deklariert werden. Dieses legt den gemeinsamen Bezeichner, sowie die zugehörigen Prozeduren fest.

Beispiel 2.14: Prozedurüberladung

```
1  MODULE overload
2
3    IMPLICIT NONE
4
5    INTERFACE square
6      MODULE PROCEDURE square_i
7      MODULE PROCEDURE square_r
8    END INTERFACE square
9
10 CONTAINS
11
12   FUNCTION square_i(i)
13     INTEGER, INTENT(in) :: i
14     INTEGER :: square_i
15     square_i = i * i
16   END FUNCTION square_i
17
18   FUNCTION square_r(r)
19     REAL, INTENT(in) :: r
20     INTEGER :: square_r
21     square_r = r * r
22   END FUNCTION square_r
23
24 END MODULE overload
```

Das Modul `overload` enthält das Interface `square`, das eine gemeinsamen Namen für die beiden Funktionen `square_i` und `square_r` definiert (Zeilen 5–8). Die Funktion `square_i` nimmt ein Argument vom Typ `INTEGER` entgegen (Zeilen 12–16), `square_r` hingegen ein Argument vom Typ `REAL` (Zeilen 18–22).

```
25   PRINT *, square(42)
26   PRINT *, square(109.5)
```

An der Aufrufstelle kann der Compiler anhand der übergebenen Argumente entscheiden, welche Funktion tatsächlich aufgerufen wird. Der Ausdruck `42`

ist ein Literal vom Typ `INTEGER`, somit wird in Zeile 25 tatsächlich `square_i` aufgerufen. Das Literal `109.5` ist hingegen vom Typ `REAL`, somit wird in Zeile 26 `square_r` aufgerufen.

Auf ähnliche Weise können auch Operatoren überladen werden.

Beispiel 2.15: Operatorüberladung

```
1  MODULE overload_add
2
3    IMPLICIT NONE
4
5    INTERFACE operator(+)
6      PROCEDURE add_up_c
7    END INTERFACE
8
9  CONTAINS
10
11   FUNCTION add_up_c(op1, op2)
12     TYPE(C), INTENT(in) :: op1, op2
13     TYPE(C) :: add_up_c
14     add_up_c%counter = op1%counter + op2%counter
15   END FUNCTION
16
17 END MODULE overload_add
```

Das Modul `overload_add` definiert den +-Operator für den Typ `C` aus Beispiel 2.9 (Zeilen 5–7). Realisiert wird die Operation in der Funktion `add_up_c`, die zwei Variablen vom Typ `C` entgegennimmt und diese addiert, indem sie die Komponente `counter` addiert (Zeilen 11–15).

2.4.10 Fixed-Format

Seit der ersten Veröffentlichung unterlag Fortran vielen Veränderungen und Weiterentwicklungen. Der wohl größte Bruch entstand zwischen den Standards FORTRAN 77 und Fortran 90, besonders durch den Übergang vom sog. Fixed-Format zum Free-Format. Der Quelltext musste bis FORTRAN 77 einem festgelegten Format folgen, das noch sehr von der Verwendung mit Lochkarten geprägt war (vgl. Slawig, 2017):

- Zeilen dürfen nicht länger als 72 Zeichen sein.
- Im ersten Zeichen einer Zeile werden Kommentare markiert.
- Zeichen 2-5 sind für Sprungmarken reserviert.

- Zeichen 6 ist für ein Fortsetzungszeichen reserviert, das markiert, dass die Anweisung der vorherigen Zeile hier fortgesetzt wird.
- Der eigentliche Quelltext beginnt am siebten Zeichen.

2.4.11 Common Blocks

Vor der Einführung von Modulen in Fortran 90 stellten sog. Common Blocks die einzige Möglichkeit dar, globale Variablen zu definieren. Common Blocks gelten heute als obsolet und ihre Verwendung ist nicht mehr zu empfehlen (vgl. Chivers und Sleightholme, 2018, S. 755+758). In älterem Code finden sie jedoch noch Verwendung.

Beispiel 2.16: Common-Block (Fixed-Format)

```
1        PROGRAM stepper
2
3        INTEGER counter
4        COMMON /cb/ counter
5
6        counter = 0
7
8        ⋮
9
10       END
11
12       SUBROUTINE step
13       INTEGER zaehler
14       COMMON /cb/ zaehler
15       zaehler = zaehler + 1
16       END
17
18       SUBROUTINE reset
19       INTEGER c
20       COMMON /cb/ c
21       c = 0
22       END
```

In dem Hauptprogramm `stepper` wird ein benannter Common Block `cb` mit einer Variable `counter` definiert (Zeile 4). Auf diese Variable kann über den Common Block auch in den Subroutinen `step` und `reset` zugegriffen werden (Zeilen 14+20).

2.5 Weitere Technologien

Neben Fortran als Hauptprogrammiersprache sind in der Klimamodellierung weitere Technologien sehr verbreitet. Im Folgenden werden die wichtigsten vorgestellt.

2.5.1 Präprozessor

Häufig anzutreffen im Quellcode von Klimamodellen sind *Präprozessordirektiven (Makros)* zur Vorverarbeitung des Quellcodes (siehe auch Méndez, Tinetti und Overbey, 2014). Diese dienen dabei im Wesentlichen zwei Zwecken: der Modellkonfiguration zur Übersetzungszeit und der Auswahl plattformabhängiger Implementationen. Ein Beispiel für den erstgenannten Zweck ist etwa das Einbinden eines Teilmodells in ein gekoppeltes Modell. Wird ein bestimmtes Teilmodell für ein Experiment nicht benötigt, spart es Übersetzungs- und ggf. auch Laufzeit, wenn der entsprechende Code gar nicht erst mitübersetzt wird. Dies kann mit Hilfe von Präprozessordirektiven gesteuert werden. Ein Beispiel für den zweiten Zweck ist etwa das Einbinden einer bestimmten Bibliothek, die auf einer Rechnerplattform verfügbar ist, auf einer anderen jedoch nicht. Mit Hilfe von Präprozessordirektiven kann abhängig von der aktuellen Plattform eine Alternative eingebunden werden.

Einige Fortran-Compiler verwenden für die Quellcodevorverarbeitung den klassischen C-Präprozessor, andere stellen einen dedizierten Fortran-Präprozessor bereit (vgl. Fortran Wiki, 2011). Die Makro-Syntax ist jedoch meist sehr ähnlich.

Beispiel 2.17: Präprozessor

```
1 MODULE mod_atm_model
2
3 #if __LAND_MODEL_ENABLED__
4   USE mod_land_model, ONLY: land_model_interface
5 #endif
```

Zeile 3 und 5 enthalten Präprozessordirektiven. Abhängig davon, ob die Schalteroption `__LAND_MODEL_ENABLED__` an diesen übergeben wird, ist die Zeile 4 im zu kompilierenden Quelltext des Moduls `mod_atm_model` enthalten oder nicht.

2.5.2 MPI

Wichtigste Technologiebasis von Klimamodellen, ebenso wie von anderen HPC-Anwendungen, ist neben der Programmiersprache das *Message Passing Interface (MPI)*, ein Standard zum Nachrichtenaustausch zwischen verteilten Rechnerprozessen, der parallele Berechnungen ermöglicht. MPI bietet grundlegende Operationen für die Punkt-zu-Punkt-Kommunikationen zwischen einzelnen Prozessen sowie für die globale Kommunikation zwischen allen Prozessen in definierten Gruppen.

Punkt-zu-Punkt-Kommunikation wird durch paarweisen Aufruf der Funktionen (in Fortran: SUBROUTINEN) `MPI_Send` und `MPI_Receive` realisiert.

Beispiel 2.18: MPI

```
1  PROGRAM mpi_send_receive
2
3    USE mpi
4
5    INTEGER :: error, rank, content, msg_id, msg_count
6
7    CALL MPI_Init(error)
8
9    CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, error)
10   msg_id = 0
11   msg_count = 1
12
13   IF (rank == 0) THEN
14     content = 42
15     CALL MPI_Send(content, msg_count, MPI_INT, 1, msg_id, &
16                   MPI_COMM_WORLD, error)
17   ELSE IF (rank == 1) THEN
18     CALL MPI_Recv(content, msg_count, MPI_INT, 0, msg_id, &
19                   MPI_COMM_WORLD, MPI_STATUS_IGNORE, error)
20     PRINT *, "Result = ", content
21   END IF
22
23   CALL MPI_Finalize(error)
24
25 END PROGRAM mpi_send_receive
```

Die obligatorischen Aufrufe der SUBROUTINEN `MPI_Init` (Zeile 7) und `MPI_Finalize` (Zeile 23) stehen am Anfang und Ende eines jeden MPI-Programms. Mit Hilfe von `MPI_Comm_rank` (Zeile 9) wird die laufende Nummer des aktuellen Prozesses, auch *Rang (rank)* genannt, in der Variable `rank` gespeichert. In Prozess 0 wird in Zeile 14 die Variable `content` auf den Wert 42 gesetzt und anschließend mit Hilfe von `MPI_Send` der Inhalt dieser Variable an Prozess 1 gesendet (Zeile 15). Prozess 1 wiederum empfängt diese Nachricht mit Hilfe von `MPI_Recv` (Zeile 18), speichert den Inhalt in der Variable `content` und gibt ihn anschließend aus (Zeile 20).

Für die globale Kommunikation stehen mehrere Funktionen zur Verfügung. Die grundlegendsten sind `MPI_Bcast` und `MPI_Gather`. Mit `MPI_Bcast` sendet ein Prozess eine Nachricht an alle anderen Prozesse einer Gruppe. Mit `MPI_Gather` empfängt ein Prozess Nachrichten aller anderen Prozesse einer Gruppe. Ebenfalls erwähnenswert ist `MPI_Reduce`, eine spezielle Form der Gather-Operation, mit der sich Reduktionsoperationen wie Summenbildung, logische UND-Verknüpfung oder Min/Max auf die eingesammelten Daten anwenden lassen (siehe auch MPI, 2015).

MPI ist ein Standard, der von mehreren Bibliotheken implementiert wird. Zu den bekanntesten Open-Source-Vertretern gehören beispielsweise MPICH (mpich.org) oder

Open MPI (openmpi.org), zudem bieten verschiedene Rechner- und Compilerhersteller ihre eigenen MPI-Implementationen an.

2.5.3 OpenMP

Neben MPI kommt in vielen Klimamodellen auch *OpenMP* (*Open Multi-Processing*) zum Einsatz. Während MPI der Kommunikation zwischen verteilten Rechnerknoten dient, lässt sich mit OpenMP parallele Arbeitsteilung zwischen Prozessoren bzw. Threads mit gemeinsamem Speicher realisieren. Mit Hilfe von *Compilerdirektiven*, sogenannten *Pragmas*, lässt sich eine Menge von Programmschritten, beispielsweise zur Abarbeitung einer Schleife, auf die verfügbaren Threads verteilen (siehe auch OpenMP, 2015).

Beispiel 2.19: OpenMP

```
1  SUBROUTINE search(needle, haystack)
2
3    INTEGER, INTENT(in) :: needle, haystack(:)
4    INTEGER :: i
5
6    !$OMP PARALLEL DO
7    DO i = LBOUND(haystack, 1), UBOUND(haystack, 1)
8      IF (haystack(i) == needle) THEN
9        PRINT *, 'Found ', needle, ' by #', OMP_GET_THREAD_NUM()
10     END IF
11   END DO
12   !$OMP END PARALLEL DO
13
14 END SUBROUTINE search
```

Die Subroutine `search` sucht eine gegebene Ganzzahl `needle` in einem Array `haystack` und gibt bei Erfolg eine Meldung auf der Standardausgabe aus. Mit Hilfe des OpenMP-Pragmas `!$OMP PARALLEL DO` (Zeile 6+12) wird die Suche auf die verfügbaren Threads aufgeteilt. Die laufenden Nummern der bei der Suche erfolgreichen Threads sind Teil der Ausgabe (Zeile 9).

2.5.4 GPGPU

Gegen Ende der 2000er Jahre haben Grafikprozessoren (GPUs) Einzug in HPC-Rechner gefunden (vgl. Kindratenko u. a., 2009; Hemsoth, 2016). Da diese dort meist nicht zum Rendern der Grafikausgabe, sondern für diverse Berechnungen verwendet werden, spricht man in diesem Zusammenhang auch von *General-purpose computing on graphics processing units* (*GPGPU*, vgl. Harris, 2014). Auch in der Klimamodellierung und numerischen Wettervorhersage gibt es Bemühungen, Modelle oder einzelne

Modellteile so umzuschreiben, dass sie auf Grafikprozessoren berechnet werden können, um somit die Leistung der Modelle zu erhöhen. Die maßgeblichen Technologien, die dabei zum Einsatz kommen, sind *CUDA*, eine proprietäre Programmiersprachenerweiterung von Nvidia sowie *OpenACC*, welches ähnlich wie OpenMP auf Pragmas aufsetzt, mit denen Codeteile markiert werden können, welche durch Grafikprozessoren beschleunigt werden sollen. In den letzten Jahren wurden einige Forschungsprojekte gestartet, die sich mit Portierungen von Klima- und Wettermodellen auf derartige Technologien beschäftigen (siehe z.B. Govett, Middlecoff und Henderson, 2010; Demeshko u. a., 2013; Norman u. a., 2015). Diese Technologien, welche vor allem Performancegewinne bei sehr hochauflösenden Modellen versprechen, gehören somit noch nicht zum Alltag und sind nicht in allen Klimamodellen zu finden, bekommen aber eine wachsende Bedeutung. In der numerischen Wettervorhersage ist mindestens ein Modell operativ im Einsatz (vgl. MeteoSchweiz, 2016).

2.5.5 Dateiformate

Klimamodelle produzieren riesigen Datenmengen und benötigen ebenfalls große Mengen an Eingabedaten, wie etwa Gitterstrukturen, Anfangs- und Randbedingungen. Diese Daten werden in der Regel in standardisierten Datenformaten abgelegt. Am gebräuchlichsten ist das *NetCDF*-Format (*Network Common Data Form*), ein Datenformat für den Austausch arrayorientierter wissenschaftlicher Daten. Die Softwarebibliotheken zum Lesen und Schreiben von NetCDF-Dateien werden federführend von der US-amerikanischen *University Corporation for Atmospheric Research (UCAR)* entwickelt (siehe auch Unidata, NetCDF). Ebenfalls von Bedeutung sind die Formate HDF4 und HDF5 (*Hierarchical Data Format*) sowie das *GRIB*-Format (GRIdded Binary), welches von der WMO zum Austausch von historischen und prognostischen Wetterdaten verwendet wird (vgl. WMO, 2015).

2.6 Zusammenfassung

Dieses Kapitel gab eine Einführung in den Gegenstandsbereich Klimamodellierung. Dazu wurden zentrale Begriffe definiert und erläutert sowie einen Überblick über die historische Entwicklung der Klimamodellierung gegeben. Eine zentrale Rolle spielt dabei die Entdeckung und Erforschung des menschengemachten Klimawandels. Zwei Aspekte kennzeichnen den Fortschritt in der Klimamodellierung: die Abbildung und Berechnung von immer mehr physikalischen Eigenschaften und Prozessen innerhalb der Modelle sowie eine immer höhere Gitterauflösung. Ermöglicht wird dies auch durch die Entwicklung immer leistungsstärkerer Computer.

Insbesondere das Aufkommen von massiv-parallelen Hochleistungsrechner hatte starken Einfluss die Entwicklung von Klimamodellen und auf deren Softwarearchitektur. Neben der Parallelisierung ist diese gekennzeichnet durch eine ausgeprägte Modularisierung, hohe Konfigurierbarkeit, arraylastige Datenstrukturen sowie der Notwendigkeit große Datenmengen ein- und auszugeben. Hauptprogrammiersprache ist Fortran. Die EntwicklerInnen sind ExpertInnen ihrer jeweiligen wissenschaftlichen Disziplin und verfügen i.d.R. über keine formale Ausbildung in der Softwareentwicklung.

Kapitel 3

Softwaretests

Fehler sind eine unvermeidliche Begleiterscheinung der Softwareentwicklung. Testen ist daher ein wichtiger Bestandteil des Softwarelebenszyklus, um vorhandene Fehler aufzudecken, um sie anschließend zu beseitigen (siehe auch Myers, Sandler und Badgett, 2011; Spillner und Linz, 2012). Dies gilt für die Klimamodelle genauso wie für jede andere Art von Software. Zwar geht von fehlerhaften Klimamodelle keine Gefahr für Leib und Leben aus, wie beispielsweise etwa von der Steuerungssoftware von Flugzeugen, jedoch können sie die Validität wissenschaftlicher Erkenntnisse beeinträchtigen und im schlimmsten Fall zu falschen politischen Entscheidungen führen.

Diese Arbeit beschäftigt sich mit dem Testen von Klimamodellen und insbesondere mit *Unittests* in diesem Umfeld. Der Fokus liegt dabei auf Softwaretests im Sinne der Definition 1.4, wonach unter einem Test das Ausführen eines Stücks Software mit dem Ziel, Fehler aufzudecken, zu verstehen ist. Dieses Kapitel führt die für diese Arbeit relevanten Begriffe aus dem Bereich des Softwaretestens ein. Zentral ist dabei u.a. der Begriff des Unittests, welcher in Abschnitt 3.3 diskutiert wird.

3.1 Testbegriff

In der softwaretechnischen Literatur existiert keine einheitliche Definition davon, was Testen ist. Diese Arbeit beruht auf der Definition 1.4 von Myers, Sandler und Badgett (2011, S. 6). Testen ist demnach das Ausführen eines Stücks Software mit dem Ziel, Fehler aufzudecken. Diese Definition basiert auf der Annahme, dass eine zu testende Software i.d.R. Fehler enthält, von denen man so viele wie möglich finden möchte.

Bill Hetzel (1988) kritisiert diese Definition als zu eng, da es noch viele andere Wege gebe, eine Software zu evaluieren, die sich ebenfalls unter dem Begriff des Testens subsumieren ließen. Für diese Arbeit erscheint Myers' Definition jedoch brauchbar. Sie stellt zwei Aspekte in den Vordergrund: Zum einen geht es um Tests, die im Ausführen der zu testenden Software bzw. eines Teils davon bestehen. Statische Tests,

die nicht aus der Ausführung, sondern der Analyse des Testobjekts bestehen (siehe auch Spillner und Linz, 2012, S. 81ff), werden in dieser Arbeit nicht betrachtet. Zum anderen besteht das Hauptziel in der Aufdeckung von Fehlern. Als Fehler gilt jede Abweichung des *Istverhaltens* einer Software von ihrem *Sollverhalten* (siehe auch Spillner und Linz, 2012, S. 7). Das bedeutet nicht, dass andere Testziele ausgeschlossen werden, jedoch steht das Ziel, Fehler aufzudecken, im Mittelpunkt dieser Arbeit.

Mit „Fehler“ in diesem Sinne sind ausdrücklich nicht statistische Fehler, d.h. die inhärenten Ungenauigkeiten in den Modellergebnissen, gemeint. Zu unterscheiden ist außerdem zwischen *Fehlerursache* und *Fehlerwirkung*. Ist davon die Rede, dass Software Fehler enthält, sind i.d.R. Fehlerursachen in Form von *Programmierfehlern* (*defect* nach Zeller, 2005, S. 2) gemeint. Diese aufzudecken und zu beheben ist die eigentliche Motivation des Testens. Jedoch lassen sich mit Tests im Sinne der o.g. Definition nur Fehlerwirkungen (*failure* nach Zeller, 2005, S. 3) aufdecken, also wahrnehmbare Abweichungen von Ist- und Sollverhalten beim Ausführen der Software. Die Ursache einer Fehlerwirkung kann bei dessen Auftreten offensichtlich sein oder erst durch möglicherweise aufwendiges *Debugging* herausgefunden werden. Nicht immer ist die Ursache einer Fehlerwirkung ein Programmierfehler in der zu testenden Software selbst. Auch fehlerhafte Eingabedaten, Compiler, Bibliotheken, Hardware oder Umgebungssysteme können Fehlerwirkungen verursachen. Da es in dieser Arbeit um das Testen und damit um das Provozieren bzw. Aufdecken von Fehlerwirkungen geht, sind im Folgenden auch diese gemeint, wenn von „Fehlern“ die Rede ist. Ansonsten werden je nach Kontext die Begriffe Fehlerursache oder Programmierfehler verwendet. Dagegen werden statistische Fehler auch als „Fehler“ bezeichnet, jedoch sollte die Bedeutung in diesen Fällen aus dem Kontext heraus deutlich werden.

Das Ausführen einer Software zu Testzwecken kann normalerweise nur stichprobenartig geschehen, da i.d.R. die Menge aller möglichen Eingaben für ein Computerprogramm zu groß ist, als dass vollständiges Testen möglich wäre. Daraus folgt, dass sich mit Testen im o.g. Sinne im Allgemeinen nicht alle Fehler in einer Software aufdecken lassen. Mit Testen kann somit auch nicht die Korrektheit einer Software nachgewiesen werden (vgl. Dijkstra, 1972, Spillner und Linz, 2012, S. 9; Myers, Sandler und Badgett, 2011).

Um einen Fehler, d.h. eine Abweichung vom Sollverhalten festzustellen, ist es notwendig, dass das Sollverhalten bekannt ist. Dies bedeutet nicht, dass dieses in Form einer formalen Spezifikation festgelegt sein muss. Insbesondere für *wissenschaftliche Software* wie etwa Klimamodelle existieren derartige Spezifikationen i.d.R. nicht (siehe auch Abschnitt 3.4). Dennoch bestehen auch für derartige Software implizite und explizite Anforderungen, deren Verletzungen auf Fehler hinweisen, etwa, dass eine Simulation nicht mittendrin abstürzt, dass Ergebnisse in einem gewissen Bereich liegen oder das kürzlich durchgeführte Änderungen keine unerwünschten Nebenwirkungen haben (siehe auch Abschnitt 3.5). Softwaretests helfen derartige Fehler aufzudecken.

Sie sind jedoch nicht geeignet, die wissenschaftliche Nützlichkeit von auf den ersten Blick plausiblen Ergebnissen zu beurteilen. Hierzu bedarf es der *wissenschaftlichen Evaluation* der Modellergebnisse (siehe auch Abschnitt 4.1).

Wenn das Ziel des Testens das Finden von Fehlern ist, ist ein Test gemäß Myers, Sandler und Badgett (2011, S. 6ff) dann erfolgreich, wenn dieses Ziel erreicht wurde:

> *„To our way of thinking, a well-constructed and executed software test is successful when it finds errors that can be fixed. That same test is also successful when it eventually establishes that there are no more errors to be found. The only unsuccessful test is one that does not properly examine the software; and, in the majority of cases, a test that found no errors likely would be considered unsuccessful, since the concept of a program without errors is basically unrealistic.“*

Da eine Verwendung der Formulierung „erfolgreicher Test“ in diesem Sinne doch zu sehr ihrer intuitiven und ansonsten üblichen Verwendung widerspricht und eher zur Verwirrung als zur Klarheit beiträgt, wird in diesem Fall Myers' Sichtweise nicht übernommen. Ein erfolgreicher bzw. bestandener Test bezeichnet im Folgenden somit einen Test, bei dem die zu testende Software keine Abweichung zwischen Ist- und Sollverhalten gezeigt hat.

3.2 Teststufen

Softwaretests lassen sich nach unterschiedlichen Gesichtspunkten klassifizieren (siehe auch Spillner und Linz, 2012, S. 11) Eine übliche Kategorisierung ist die sog. *Teststufe*. Diese beschreibt, auf welcher Granularitätsebene eine Software getestet wird. Unterschieden wird zwischen *Komponententests*, *Integrationstests* und *Systemtests* (siehe auch Beizer, 1990, S. 21, Spillner und Linz, 2012, S. 43, Perry, 2006, S. 70).

Komponententests Bei einem Komponententest wird eine einzelne Programmkomponente dahingehend getestet, ob sie ihre jeweiligen Aufgaben gemäß den Anforderungen erfüllt. Die Art der Komponente ist nicht näher festgelegt. Beispiele sind etwa einzelne Routinen, Module oder Subsysteme. Je nach Autor werden die Begriffe Komponententest und Unittest synonym verwendet (siehe z.B. Perry, 2006, S. 70), Unittest als eine Art Komponententest betrachtet (siehe z.B. Spillner und Linz, 2012, S.44) oder graduell unterschieden (siehe z.B. Beizer, 1990, S. 21). In dieser Arbeit wird der Begriff Unittest verwendet, wobei Tests einzelner Prozeduren im Vordergrund stehen (siehe auch Abschnitt 3.3).

Integrationstests Bei einem Integrationstest werden mehrere (bereits getestete) Komponenten zu größeren Teilsystemen verbunden und so das Zusammenspiel der Komponenten miteinander überprüft (vgl. Spillner und Linz, 2012, S. 52).

Systemtests Bei einem Systemtest wird ein vollständig integriertes System dahingehend überprüft, ob es die spezifizierten Anforderungen erfüllt.

Als weitere Teststufe in dieser Reihe nennen viele Autoren noch den *Akzeptanztest*, auch *Abnahmetest* genannt, bei denen unter Beteiligung von BenutzerInnen der Software diese auf ihre Tauglichkeit für den produktiven Einsatz hin überprüft wird. Diese Stufe bildet gegenüber den Systemtests keine neue Granularitätsstufe, unterscheidet sich aber durch ihre Ziele und Art und Weise der Durchführung. Sie ergibt vor allem dann Sinn, wenn a) es sich bei der zu testenden Software um eine interaktive Endbenutzeranwendung handelte und b) es eine klare Trennung zwischen EntwicklerInnen und BenutzerInnen gibt. Beides ist bei Klimamodellen nicht der Fall, weshalb diese Art von Test hier auch nicht weiter betrachtet werden soll.

Auch die Unterscheidung zwischen Integrationstest und Systemtest ist in diesem Kontext wenig zielführend (siehe auch Clune und Rood, 2011). Es ließe sich zwar durchaus argumentieren, dass gewisse umfangreiche Experimente, die für ein Modell zu Testzwecken standardmäßig durchgeführt werden, als Systemtest betrachtet werden können, während einfachere Konfigurationen wiederum Integrationstests darstellen, jedoch wäre diese Unterscheidung eine graduelle und eine Diskussion der Frage, wo der Integrationstest aufhört und der Systemtest anfängt, für diese Arbeit, die sich auf Unittests konzentriert, unerheblich.

Anstelle von Integrationstest und Systemtest soll in dieser Arbeit der Begriff des *Ende-zu-Ende-Tests (end-to-end test)* Verwendung finden, welcher vor allem als Abgrenzung zum Unittest verstanden werden soll und auch von Easterbrook und Johns (2009) im Zusammenhang mit Klimamodellen verwendet wird. Easterbrook und Johns liefern selbst keine Definition, es gelte daher die Folgende bezogen auf Klimamodelle:

Definition 3.1: Ende-zu-Ende-Test

Test bestehend aus der Ausführung eines Klimamodells in einer beliebigen Konfiguration, bei dem alle Phasen einer Simulation von der Initialisierung, über die Zeitschleife bis zum Aufräumen durchlaufen werden.

Im Folgenden wird auch die von der englischen Bezeichnung abgeleitete Abkürzung *E2E-Test* verwendet. Zu den Phasen einer Simulation siehe auch Abschnitt 2.1.7. Die Zeitschleife kann bei einem E2E-Test aus beliebig vielen Zeitschritten bestehen, muss jedoch mindestens einen umfassen. Die Phase des Postprocessings kann, aber muss nicht Teil eines E2E-Tests sein. Der E2E-Test ist die vorherrschende Art des Testens

in der Klimamodellierung (siehe auch Kapitel 4). In der Softwaretechnik-Literatur ist dieser Begriff bisher nicht allzu sehr etabliert, vereinzelt wird er für eine Form hochrangiger Intergrationstests verwendet (siehe z.B. Tsai u. a., 2001).

3.3 Unittest

Im Fokus dieser Arbeit stehen Tests einzelner Prozeduren, sog. Unittests. Die Verwendung dieses Begriffs ist sehr uneinheitlich. Sucht man im Internet nach „Unittest“ findet man diverse Newsgroup-, Foren- und Blogeinträge mit verschiedensten Meinungen dazu, was ein Unittest sei und was nicht. Diese Arbeit orientiert sich maßgeblich an zwei Definitionen: Zum einen an der von mir und Julian Kunkel bereits in Hovy und Kunkel, 2016 verwendeten Definition für Unittests im HPC-Kontext, welche auf den folgenden Eigenschaften beruht:

Definition 3.2: Unittest (A)

A1. *„The code under test (CUT) is a defined subset of the application code, usually a certain module or routine. It is tested isolated from the parts of the application that call the CUT, but don't have to be separated from the modules which are used by the CUT itself.“*

A2. *„Testing does not focus on scientific validation but on finding defects in the implementation and/or preventing code regression.“*

A3. *„Execution is significantly faster than running the whole application. For most HPC applications that means that a test shouldn't contain the main loop, for example, in numerical simulations with time integration, a test should cover only one time step. As a consequence, unit tests are not a means to test the stability of numerical algorithms but they can test the stability of one integration step. Similar techniques may be used to perform tests across time steps, but we won't call it a unit test.“*

A4. *„Execution is automated, that is, the check whether or not a test is passed is done by the test program itself and not by a person.“*

Zum anderen an der Definition von Martin Fowler (2014)[6], die er wiederum aus anderen

[6]Bei der zitierten Quelle handelt es sich zwar lediglich um einen Blogeintrag, der Autor ist jedoch einer der weltweit bekanntesten Experten für moderne Softwareentwicklung und Autor einflussreicher Bücher, wie etwa „Refactoring: Improving the Design of Existing Code“ (Fowler, 1999) oder „Domain-Specific Languages“ (Fowler, 2010). Bei dem genannten Blog handelt es sich um eine Mischung aus Blog und Wiki („Bliki“), in der der Autor eine Vielzahl softwaretechnischer

existierenden Definitionen extrahiert hat und nach der ein Unittest allgemein folgende Eigenschaften aufweist:

Definition 3.3: Unittest (B)

B1. *„Firstly there is a notion that unit tests are low-level, focusing on a small part of the software system.“*

B2. *„Secondly unit tests are usually written these days by the programmers themselves using their regular tools - the only difference being the use of some sort of unit testing framework.“*

B3. *„Thirdly unit tests are expected to be significantly faster than other kinds of tests.“*

Wie man sieht, widersprechen sich diese beiden Definitionen nicht, sondern setzen unterschiedliche Schwerpunkte und ergänzen sich dabei. Die Punkte A1 und B1 entsprechen sich weitestgehend. Die Konkretisierung von A1 auf einzelne Module oder Prozeduren findet sich bei Fowler im selben Artikel in ähnlicher Weise etwas später, jedoch betont er dabei auch die Abhängigkeit vom jeweiligen Kontext:

> *„So there's [sic] some common elements, but there are also differences [in den Definitionen von Unittests]. One difference is what people consider to be a unit. Object-oriented design tends to treat a class as the unit, procedural or functional approaches might consider a single function as a unit. But really it's a situational thing - the team decides what makes sense to be a unit for the purposes of their understanding of the system and its testing.“* (Fowler, 2014)

Fowler geht darüber hinaus in seinem Artikel auf die Unterscheidung zwischen *Solitary* und *Sociable Unit Tests* nach Jay Fields (2014) ein. Solitary Unit Tests verwenden sog. *Testdoubles*, um den CUT unabhängig von seinen eigenen Abhängigkeiten testen zu können. Unter Testdoubles versteht man vereinfachte Ersatzkomponenten, die im Rahmen von Tests anstelle von echten Komponenten, also beispielsweise für andere Prozeduren, die vom CUT aufgerufen werden, eingesetzt werden (vgl. Meszaros, 2006, S. 133ff, Fowler, 2007). Dadurch werden die Ergebnisse von Solitary Unit Tests nicht von Programmierfehlern im darunter liegenden Code beeinflusst. Unittests, die keine Solitary Unit Tests sind, also indirekt auch Code außerhalb des CUT aufrufen, bezeichnet Fields als Sociable Unit Tests. In Abbildung 3.1 werden die beiden Testarten für Tests einer einzelnen Prozedur im Vergleich zum Anwendungscode dargestellt.

Begriffe definiert.

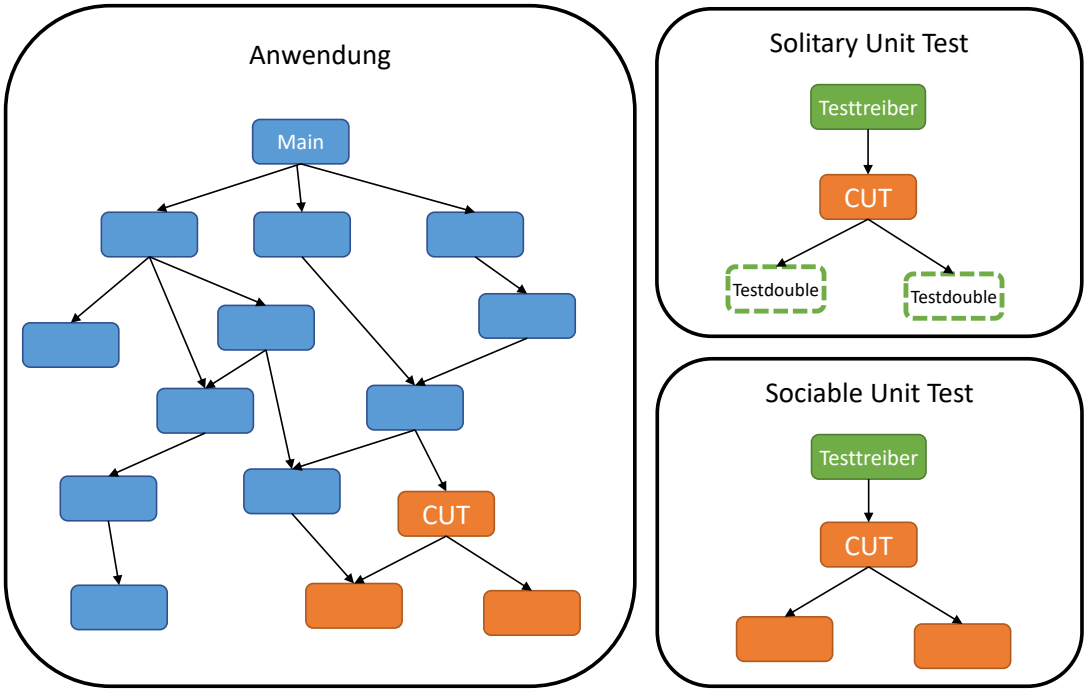


Abbildung 3.1: CUT im Kontext der Originalanwendung sowie innerhalb eines Solitary und eines Sociable Unit Tests

Nicht immer ist die Erstellung von Solitary Unit Tests sinnvoll. Sie ist vor allem dann nützlich, wenn eine ersetzte Komponente die Testausführung unnötig verlangsamen würde, beispielsweise durch den Zugriff auf externe Ressourcen, oder wenn sie sich nicht deterministisch verhält (vgl. Fowler, 2014). Oftmals sind jedoch die richtige Wahl und die richtige Verwendung von Unterprozeduren wesentliche Aspekte der Implementation einer Prozedur, welche gerade auch durch einen Test überprüft werden sollen. In solchen Fällen müssen diese im Test natürlich unverändert bleiben. In A1 wird daher ausdrücklich betont, dass auch Tests, bei denen der CUT nicht von den eigenen Abhängigkeiten getrennt wird, also Sociable Unit Tests, ebenfalls von der Definition eingeschlossen sind. Auch Fowler schließt diese Art von Tests ausdrücklich ein, lässt dabei jedoch nicht unerwähnt, dass es in diesem Punkt durchaus abweichende Meinungen gibt:

> *„Indeed using sociable unit tests was one of the reasons we were criticized for our use of the term „unit testing“. I think that the term „unit testing“ is appropriate because these tests are tests of the behavior of a single unit. We write the tests assuming everything other than that unit is working correctly.“* (Fowler, 2014)

Da A2 nur im Kontext wissenschaftlicher Software wie etwa Klimamodelle Sinn ergibt, findet sich bei Fowler, der im Wesentlichen die Entwicklung von interaktiven, objektorientierten Anwendungen beschreibt, keine Entsprechung.

Der Punkt A3 aus Hovy und Kunkel, 2016 findet seine Entsprechung in Punkt B3 aus Fowler, 2014, wobei er diesen um einige klimamodellrelevante Aspekte erweitert. Beide Definitionen legen nicht fest, was unter signifikant zu verstehen ist. Diese Frage wird in dieser Arbeit an späterer Stelle diskutiert (siehe Abschnitt 6.2).

Abbildung 3.1 verdeutlicht einen wesentlichen technischen Unterschied zwischen E2E- und Unittests: Während bei E2E-Tests immer das Hauptprogramm der eigentlichen Anwendung ausgeführt wird, werden für Unittests spezielle Programme, sog. *Testtreiber*, benötigt, in denen der CUT isoliert und automatisiert ausgeführt wird. Gemäß A4 soll nicht nur die Ausführung des CUT automatisiert geschehen (da Klimamodelle i.d.R. keine Benutzerinteraktion enthalten, ergibt sich dies normalerweise von allein), sondern auch auf die Überprüfung der Ergebnisse und die Entscheidung, ob ein Test erfolgreich war oder nicht. Als Entsprechung kann bei Fowler der Verweis auf die verwendeten *Unittestframeworks* in B2 verstanden werden, wobei dieser zusätzlich die Mittel der Automatisierung konkretisiert. Unittestframeworks stellen die Infrastruktur zur automatischen Ausführung von Unittests bereit, u.a. für die Erstellung von Testtreibern, und erleichtern das automatische Überprüfen von Testergebnissen. Sie existieren für die meisten Programmiersprachen, zu den bekanntesten gehört etwa *JUnit* für Java. Für Fortran existieren mehrerer derartiger Frameworks (vgl. Fortran Wiki, 2018), wie z.B. *pFUnit* (Rilee und Clune, 2014). Die Verwendung eines solchen Frameworks wird in dieser Arbeit nicht verlangt.

Darüber hinaus enthält die Definition von Fowler in B2 einen wesentlichen, weiteren Punkt, welchen die Definition aus Hovy und Kunkel, 2016 nicht abdeckt. Danach werden Unittests von den EntwicklerInnen selbst unter Verwendung ihrer üblichen Werkzeuge geschrieben und nicht von dedizierten TesterInnen mit ihren eigenen spezialisierten Werkzeugen. Hieraus lässt sich ein anwendungsorientierter Ansatz ableiten. Die sich hieraus ergebenden Konsequenzen werden ausführlich in Kapitel 6 betrachtet.

3.4 Das Testorakelproblem

Das Testen *wissenschaftlicher Software* gilt allgemein als schwierig (siehe z.B. Carver u. a., 2007; Kelly und Sanders, 2008; Segal, 2009). Mit wissenschaftlicher Software (*scientific software*) ist Software gemeint, die zum Zwecke des wissenschaftlichen Erkenntnisgewinns erstellt wird, wie etwa Klimamodelle. Ein häufig genanntes Problem im Zusammenhang mit dem Testen wissenschaftlicher Software ist das Fehlen

geeigneter *Testorakel* (vgl. Kanewala und Bieman, 2014). Ein Testorakel ist ein Mechanismus oder eine Informationsquelle, der bzw. die dazu dient, zu entscheiden, ob das Istverhalten eines zu testenden Programms bzw. Programmteils seinem Sollverhalten entspricht (siehe auch Definition 1.5). Testorakel basieren beispielsweise auf Spezifikationen oder auf Referenzimplementationen, ergeben sich aus dem Wissen oder Erfahrung der EntwicklerIn bzw. der TesterIn oder im Fall von numerischen Problemen auf analytischen Lösungen.

3.4.1 Vollständige und unvollständige Orakel

Ein *vollständiges Orakel* kann für für jede beliebige Eingabe eines zu testenden Programms entscheiden, ob die jeweilige Ausgabe des Programms korrekt ist (vgl. Staats, Whalen und Heimdahl, 2011). Da Tests jedoch i.d.R. aus Stichproben bestehen, ist ein vollständiges Orakel normalerweise nicht notwendig. Es reicht aus, wenn für die Ausgaben der Stichproben diese Entscheidung getroffen werden kann. Doch häufig ist auch dies nicht zuverlässig möglich. Für viele Probleme innerhalb wissenschaftlicher Anwendungen sind die korrekten Ergebnisse unbekannt. Der Zweck einer Software besteht in vielen Fällen gerade darin, Lösungen für ein Problem zu liefern, für das es keine andere Lösungsmöglichkeit gibt. Beispielsweise werden numerische Verfahren angewendet, um Gleichungssysteme zu lösen, für die es keine analytische Lösung gibt. In diesen Fällen müssen andere Testorakel verwendet werden, z.B. basierend auf Heuristiken (siehe auch Hoffman, 1998; Kanewala und Bieman, 2014; Barr u. a., 2015). Damit verbundene Ungenauigkeiten müssen dann in Kauf genommen werden.

3.4.2 Metamorphe Relationen

Eine alternative Form zu klassischen Testorakeln, d.h. dem Vergleich mit konkreten erwarteten Ergebnissen, stellt beispielsweise das sog. *metamorphe Testen (metamorphic testing, Chen, Cheung und Yiu, 1998)* dar. Dabei werden sog. *metamorphe Relationen (metamorphic Relations)* abgeleitet, die zwischen den Ergebnissen unterschiedlicher Testfälle gelten müssen. Beim Testen werden dann die Ergebnisse der unterschiedlichen Testfälle untereinander verglichen und überprüft, ob die Relationen eingehalten werden.

Beispiel 3.1: Metamorphe Relationen

a) Ein zu testendes Programm implementiert eine Funktion f, von der wir wissen, dass sie stetig wächst. Als einfachste metamorphe Relationen können wir daraus ableiten: $x' > x \Rightarrow f(x') > f(x)$.

b) Ein anderes Programm implementiert die Sinusfunktion (und wir haben keine Taschenrechner oder andere Testorakel parat). Wir kennen zwar die erwarteten Ergebnisse einiger spezieller Werte, wie etwa $sin(0) = 0$, $sin(\frac{1}{2}\pi) = 1$, $sin(\pi) = 0$ oder $sin(\frac{3}{2}\pi) = -1$, andere konkrete Ergebnisse jedoch nicht. Allerdings kennen wir u.a. die folgende metamorphe Relation: $x' = x + n \cdot \pi \Rightarrow sin(x') = sin(x) \cdot -1^n$ für alle $n \in \mathbb{N}$.

3.4.3 Implizites Testorakel

Die einfachste Form des Testorakels ist das sog. *implizite Orakel* (vgl. Barr u. a., 2015) oder auch *Nullorakel* (Shrestha und Rutherford, 2011) genannt. Dieses Orakel prüft, vereinfacht gesprochen, lediglich, ob ein Programm erfolgreich durchläuft und während der Ausführung keinen Fehler produziert, der z.B. zum Programmabbruch führt.

3.4.4 Menschliche und automatische Testorakel

Wird die Entscheidung, ob ein Test erfolgreich war oder nicht, durch eine EntwicklerIn oder TesterIn getroffen, spricht man von einem *menschlichen Testorakel* (siehe auch Oliveira, Kanewala und Nardi, 2014). Menschliche Orakel gelten allgemein als aufwendig und unzuverlässig (vgl. Shahamiri, Kadir und Mohd-Hashim, 2009), automatisierte Tests sollten daher auch über eine automatische Prüfung der Ergebnisse verfügen. Der gegenteilige Begriff *automatisches Orakel* wird in der Literatur vorwiegend für vollständige automatische Orakel verwendet (siehe z.B. Shahamiri, Kadir und Mohd-Hashim, 2009; Barr u. a., 2015). In dieser Arbeit werden damit jedoch sämtliche automatischen Prüfungen, die entscheiden, ob ein Test erfolgreich war oder nicht, bezeichnet.

3.5 Regressionstests

Jede Änderung an einer Software birgt das Risiko neuer Fehler. Dies betrifft nicht nur etwaige neue Funktionalitäten einer Software, auch in bereits bestehenden und getesteten Funktionalitäten können im Zuge von Änderungen Fehler entstehen. *Regressionstesten* hat zum Ziel derartige Fehler aufzudecken (siehe auch Spillner und Linz, 2012, S. 77f). Hierzu können bereits bestehende Tests erneut ausgeführt werden, sofern diese die mit der Änderung verbundenen Risiken ausreichend abdecken.

Häufig kann beim Regressionstesten eine Vorversion der geänderten Software als Testorakel dienen. Bedingung hierfür ist, dass das getestete Verhalten der Software

gleichbleiben soll. Diese Form des Orakels nennt Douglas Hoffman (1998) auch *Konsistenzorakel*.

In der Klimamodellierung haben kleine Änderungen häufig große Auswirkungen, die in vielen Fällen auch gewollt bzw. toleriert sind. Dennoch spielt das Regressionstesten hier eine große Rolle (siehe auch Kapitel 4), etwa bei nichtfunktionalen Änderungen, d.h. bei Änderungen, die die Modellergebnisse nicht beeinflussen sollen. Hierzu gehören beispielsweise Optimierungen, Refactorings, Änderungen an der Infrastruktur oder Portierungen. Hinzu kommt, dass Änderungen häufig nur bestimmte Konfigurationen betreffen sollen, während sich für andere Konfigurationen die Modellergebnisse nicht ändern sollen. Auch dies kann mit Hilfe von Regressionstests überprüft werden.

3.6 Continuous Integration

Automatisierte Regressionstest spielen auch eine wichtige Rolle beim *Continuous Integration (CI)*. Als Continuous Integration bezeichnet man die Praxis, dass einzelne EntwicklerInnen ihre eigenen Änderungen am Code so schnell wie möglich, idealerweise täglich, in den Hauptentwicklungszweig des Entwicklungsteams integrieren. Dieses Vorgehen wird unterstützt durch *Versionskontrollsysteme*, die es erlauben Änderungen leicht wieder rückgängig zu machen und zwischen verschiedenen Versionsständen und Entwicklungszweigen hin- und herzuspringen bzw. diese miteinander zu verschmelzen, sowie durch sog. *CI-Server*, die aus neuen Versionen automatisiert einen *Build* erstellen, d.h. alle Schritte durchführen, die notwendig sind um aus dem Quellcode der Software ein ausführbares Programm zu erstellen und dieses ggf. in Betrieb zu nehmen. Dazu gehört auch das automatische Ausführen von Regressionstests. Auch in Umgebungen, in denen kein Continuous Integration praktiziert wird, werden CI-Server häufig dafür eingesetzt, um in regelmäßigen Abständen, etwa täglich bzw. in der Nacht (*nightly*), einen Build des Hauptentwicklungszweigs zu erstellen und Regressionstests auszuführen (vgl. Fowler, 2006).

3.7 Testevaluation

In der Softwaretestliteratur werden zahlreiche Testverfahren und -werkzeuge vorgeschlagen (siehe z.B. Garousi und Mäntylä, 2016). Um ein Testverfahren zu evaluieren, muss u.a. überprüft werden, ob mit ihm nützliche Tests erzeugt werden können. Zudem ist es für EntwicklerInnen und TesterInnen notwendig, die Nützlichkeit einer Testsammlung bewerten zu können.

Nützliche Tests sind solche, die *effektiv* Fehlerursachen im zu testenden Programm wirksam werden lassen, d.h. zu wahrnehmbaren Fehlern bei der Ausführung des Programms führen. Die Effektivität ergibt sich aus dem Verhältnis der Anzahl der aufgedeckten Fehlerursachen zur Anzahl aller Fehlerursachen im getesteten Programm. Um dieses zu bestimmen, muss die Anzahl aller Fehlerursachen bekannt sein. Dies in realen Anwendungen in der Regel nicht der Fall.

Testverfahren können jedoch evaluiert werden, indem stellvertretend Programme getestet werden, von denen die Menge der Fehlerursachen weitestgehend bekannt ist. Hierbei existieren zwei alternative Formen:

1. Programme, in die künstliche Programmierfehler eingefügt wurden, sog. *Mutationen* (siehe auch DeMillo, Lipton und Sayward, 1978; Jia und Harman, 2011)
2. Reale Programme, in die bereits korrigierte Programmierfehler wieder eingefügt wurden (siehe z.B. Frankl und Iakounenko, 1998; Just, Jalali und Ernst, 2014)

Mutationen können entweder manuell erzeugt werden (siehe z.B. Hutchins u. a., 1994) oder automatisch mit Hilfe von Softwarewerkzeugen generiert werden (siehe z.B. King und Offutt, 1991; Coles u. a., 2016). Sowohl die manuelle Zusammenstellung historischer Programmierfehler aus realen Programmen sowie die manuelle Erzeugung von Mutationen sind mit einem hohen Aufwand verbunden. Werkzeuge für die automatische Erzeugung von Mutationen stehen zudem nicht für jede Programmiersprache zur Verfügung. Für modernes Fortran existiert beispielsweise (nach meinem Kenntnisstand) kein derartiges Werkzeug, auch wenn viele der ersten dieser Werkzeuge auf FORTRAN 77 ausgerichtet waren (vgl. Jia und Harman, 2011).

Eine Alternative zur Messung der Fehlererkennungsfähigkeit eines Testverfahrens oder eine Testsammlung stellt die Messung der *Codeabdeckung* dar. Bezogen auf eine Zielanwendung wird dabei der Anteil des beim Testen ausgeführten Codes in Relation zu dessen Gesamtmenge ermittelt. Dahinter steht die Annahme, dass je größer der Codeanteil des Programms ist, der beim Testen ausgeführt wird, desto mehr Programmierfehler können wirksam werden. Dieser Zusammenhang ist umstritten, so gibt sowohl Studien, die diese Annahme bestätigen als auch solche, die ihr widersprechen (vgl. Schwartz und Hetzel, 2016). Dennoch dienen derartige Abdeckungsmetriken häufig zur Bewertung von Testsammlung und Testverfahren (siehe z.B. Fraser und Arcuri, 2011; Gopinath, Jensen und Groce, 2014; Fu und Su, 2017).

Typische Abdeckungsmetriken, die in der Regel mit entsprechenden Softwarewerkzeugen ermittelt werden, sind (siehe auch Myers, Sandler und Badgett, 2011, S.41ff, Ammann und Offutt, 2016, S. 73ff):

- *Prozedurabdeckung* (*procedure* oder *function coverage*):

$$\frac{\text{Beim Testen ausgeführte Prozeduren}}{\text{Gesamtmenge der Prozeduren des zu testenden Programm(teil)s}}$$

- *Zeilenabdeckung* (*line* oder *statement coverage*):

$$\frac{\text{Beim Testen ausgeführte Codezeilen}}{\text{Gesamtmenge der ausführbaren Codezeilen des zu testenden Programm(teil)s}}$$

- *Zweigabdeckung* (*branch coverage*):

$$\frac{\text{Beim Testen ausgeführte Zweige von Fallunterscheidungen}}{\text{Gesamtmenge der Zweige des zu testenden Programm(teil)s}}$$

Werden derartige Metriken erhoben, ist es normalerweise das Ziel, eine möglichst hohe Abdeckung zu erreichen. Ein solches Testziel wird auch *Abdeckungskriterium (coverage criteria)* genannt (vgl. Ammann und Offutt, 2016, S. 73ff). Dabei gibt es stärkere und schwächere Abdeckungskriterien. So impliziert beispielsweise eine Zweigabdeckung von 100 % auch eine Zeilenabdeckung von 100 %, während diese wiederum eine Prozedurabdeckung von 100 % zur Folge hat. In die andere Richtung lassen sich diese Zusammenhänge nicht herstellen. Die Zweigabdeckung gilt daher als das stärkste, die Prozedurabdeckung als das schwächste Kriterium.

3.8 Zusammenfassung

Das Gebiet des Softwaretestens ist ein wichtiger Bestandteil des Wissenskontexts dieser Arbeit gemäß des Design-Science-Modells (siehe auch Wieringa, 2014, S. 7 f). In diesem Kapitel wurden die für die Arbeiten relevante Begriffe aus diesem Gebiet eingeführt. Mit Tests sind in dieser Arbeit dynamische Tests gemeint, d.h. Tests, die auf der Ausführung der zu testenden Software bestehen. Ziel des Testens ist das Aufdecken von Fehlern. Mit Fehler sind Abweichungen des Ist- vom Sollverhalten der Software gemeint.

Ende-zu-Ende-Tests bestehen aus der Ausführung des Hauptprogramms eines Klimamodells und Durchlaufen alle Phasen einer Simulation von der Initialisierung, über die Zeitschleife bis zur Aufräumphase. Dagegen stehen Unittests, die kleinere Codeabschnitte, etwa einzelne Routinen, testen. Diese werden isoliert mit Hilfe eines sog. Testtreibers ausgeführt. Testorakel werden benötigt, um zu entscheiden, ob ein Test erfolgreich war oder nicht. Regressionstests werden nach Änderungen einer Software ausgeführt, um Fehler aufzudecken, die durch die Änderungen entstehen. In vielen Fällen kann bei diesen Tests die Vorversion der zu testenden Software als Testorakel dienen.

Kapitel 4

Praxis des Testens in der Klimamodellierung

In Kapitel 2 wurde ausführlich der Kontext der Softwareentwicklung in der Klimamodellierung beschrieben. In einer solchen Umgebung ist Softwareentwicklung nur möglich mit einem Mindestmaß an Koordination und systematischem Testen. Da die EntwicklerInnen von Klimamodellen es jedoch vorziehen, über ihre wissenschaftlichen Erkenntnisse in ihren jeweiligen Fachdisziplinen zu publizieren anstatt über ihre eigenen Softwarenentwicklungspraktiken, existieren in der Literatur keine umfangreichen Beschreibungen davon, wie Testen in der Klimamodellierung praktiziert wird. Im Rahmen dieser Arbeit habe ich daher mit Hilfe einer eigenen qualitativen Studie die Testpraktiken in verschiedenen Klimaforschungsinstituten in Deutschland, Japan und den USA untersucht.

Ausgangspunkt waren dabei die Forschungsfragen:

1. Was charakterisiert die Praxis des Softwaretestens in der Klimamodellierung?
2. Welche Defizite lassen sich dabei aus softwaretechnischer Sicht identifizieren?

Die Untersuchung alltäglicher Entwicklungspraxis hilft dabei, die besonderen Eigenschaften wissenschaftlicher Softwareentwicklung zu verstehen. Sie liefert die Grundlage für Verbesserungen und anwendungsorientierte Methoden, zugeschnitten auf die Bedürfnisse der EntwicklerInnen. Die Ergebnisse der Studie bilden somit auch die Motivation für den konstruktiven Teil dieser Arbeit. Die Beschreibungen in diesem Kapitel gehen jedoch darüber hinaus und liefern ein breites Bild der Testpraxis in der Klimamodellierung. Diese können somit auch weiteren Arbeiten als Grundlage dienen. Die Inhalte dieses Kapitel wurden zu großen Teilen bereits in Hovy u. a., einger. 2019 beschrieben und stellen zum Teil direkte Übersetzungen des englischen Originals dar.

Vor der eigentlichen Studie enthält dieses Kapitels zunächst eine Beschreibung der wichtigsten Methoden zur wissenschaftlichen Evaluation von Klimamodellen (Abschnitt 4.1). Danach werden die im Rahmen der Studie untersuchten Fallstudien eingeführt (Abschnitt 4.2) und anschließend das Vorgehen erläutert (Abschnitt 4.3). Anschließend werden die Ergebnisse der Studie beschrieben (Abschnitte 4.4–4.10), wobei das zentrale Thema dieser Arbeit Unittests in Abschnitt 4.9 diskutiert wird. Abschnitt 4.10 beschäftigt sich wiederum mit den Auswirkungen der herrschenden Praxis auf die Softwarequalität. Zum Schluss werden in Abschnitt 4.11 Einschränkungen dieser Studie diskutiert, Abschnitt 4.12 wirft einen Blick auf vergleichbare Arbeiten in diesem Bereich.

4.1 Wissenschaftliche Modellevaluation

In dieser Arbeit wird zwischen Softwaretests und der wissenschaftlichen Evaluation von Klimamodellen unterschieden. Während bei der Evaluation die wissenschaftliche Nützlichkeit der Modellergebnisse untersucht wird, ist das Ziel des Softwaretestens das Auffinden von Fehlern bei Ausführung des Modells. Ein Fehler kann jedes unerwünschte Verhalten sein, wie etwa Programmabbrüche oder verfälschte Ergebnisse. Der Fokus dieser Arbeit, ebenso wie der in diesem Kapitel beschriebenen Studie, ist das Softwaretesten. Da hierbei jedoch auch Verfahren zur Modellevaluation eine Rolle spielen, gibt dieser Abschnitt eine Einführung in die wichtigsten Methoden.

Eine strikte Trennung zwischen Evaluieren und Testen lässt sich nicht immer herstellen. Ein typischer Arbeitsablauf einer EntwicklerIn sieht folgendermaßen aus:

- Änderungen am Code vornehmen
- Code kompilieren ⇒ *Kompilierung erfolgreich? Irgendwelche Warnungen?*
- Simulation ausführen ⇒ *Stürzt ab oder läuft durch?*
- Ergebnisse anschauen (Diagramme) ⇒ *Sieht gut aus? Wenn nicht, warum nicht? Programmierfehler? Instabiles numerisches Verfahren? Falsche Parameter?*

So gesehen, ist jede Modellausführung eine Art von Test, die sowohl aus Programmierer-, wie auch aus Wissenschaftlersicht betrachtet wird. Dennoch ist möglich, spezifische Methoden dem einen oder dem anderen Ziel zuzuordnen.

Modelle sind nicht nur Mittel, sondern auch Gegenstand der Forschung. Entsprechend existiert auch viel Literatur zu dessen Evaluation, sowohl zu Methoden als auch zu Evaluationsergebnissen einzelner oder mehrerer Modelle. Vicky Pope und Terry Davies (2002) beschreiben verschiedene Evaluationsmethoden, die am britischen *MetOffice*

in der Modellentwicklung zum Einsatz kommen. Die Evaluation von Modellen und Ergebnissen, die für die Erstellung des 5. IPCC-Reports herangezogen wurden, wird von Gregory Flato u. a. (2013) ausführlich dargestellt. Eine Übersicht von speziellen Methoden für die Evaluation von dynamischen Kernen von Atmosphärenmodellen hat David L. Williamson (2007) zusammengestellt.

Charakterisierend für die Modellevaluation ist, dass sie sich nicht automatisieren lässt. Die Bewertung der Ergebnisse wird grundsätzlich von den WissenschaftlerInnen selbst vorgenommen. Ein wichtiger Schritt dabei ist das Postprocessing, in dem die Modellergebnisse aufbereitet werden, um sie der jeweiligen Fragestellung entsprechend visualisieren zu können. Die so angefertigten Diagramme sind alltägliche Werkzeuge der EntwicklerInnen, sowohl um selbst einen Überblick über die Ergebnisse einer Simulation zu bekommen als auch zur Kommunikation innerhalb des Entwicklerteams oder auch auf Tagungen und in wissenschaftlichen Veröffentlichungen.

Neben der wissenschaftlichen Betrachtung spielen auch Leistungstests eine wichtige Rolle bei der Modellevaluation (siehe z.B. Tomita, Goto und Satoh, 2008; Dennis u. a., 2012; Balaji, Maisonnave u. a., 2017). Nur ein Modell, dass in akzeptabler Zeit die gewünschten Simulationen durchführen kann, ist nützlich.

4.1.1 Vergleich mit Beobachtungsdaten

Der Vergleich mit Beobachtungsdaten ist die wichtigste Form der Modellvalidierung, da letztlich nur diese Aufschluss darüber gibt, wie gut ein Modell das Klima der Erde tatsächlich simuliert. Die größte Schwierigkeit hierbei ist es, geeignete Vergleichsdaten zu finden. Naturgemäß stehen diese nur für vergangene Zeiträume zur Verfügung. Anders als bei Wettervorhersagen, wo nach wenigen Tagen die Güte der ursprünglichen Prognosen festgestellt werden kann, beziehen sich Klimaprognosen auf Zeiträume von mehreren Jahrzehnten. Nach Ablauf dieser Zeit sind die einst verwendeten Modelle jedoch längst überholt und nicht mehr Gegenstand oder Mittel der aktuellen Forschung. Für vergangene Zeiträume besteht jedoch das Problem, dass nicht für alle Zeiträume und alle Gebiete der Erde ausreichende Messdaten vorliegen, zudem enthalten auch Messdaten Ungenauigkeiten und Fehler. Verwendet werden Daten von Messstationen und Satelliten sowie *Reanalysedaten* (siehe z.B. Schmidt u. a., 2006). Unter Reanalyse versteht man die Homogenisierung von heterogenem und lückenhaftem Datenmaterial. Dabei kommen ebenfalls Modelle (i.d.R. Wettermodelle) zum Einsatz, mit deren Hilfe durch Datenassimilation aus zeitlich und räumlich unregelmäßig verteilten Messwerten gleichmäßig verteilte Daten berechnet werden (siehe auch Edwards, 2010). Reanalysedaten, die sich zur Evaluation von Klimamodellen eignen, werden z.B. vom *European Centre for Medium-Range Weather Forecasts (ECWMF)* (Dee u. a., 2011) oder vom amerikanischen NCAR (Kalnay u. a.,

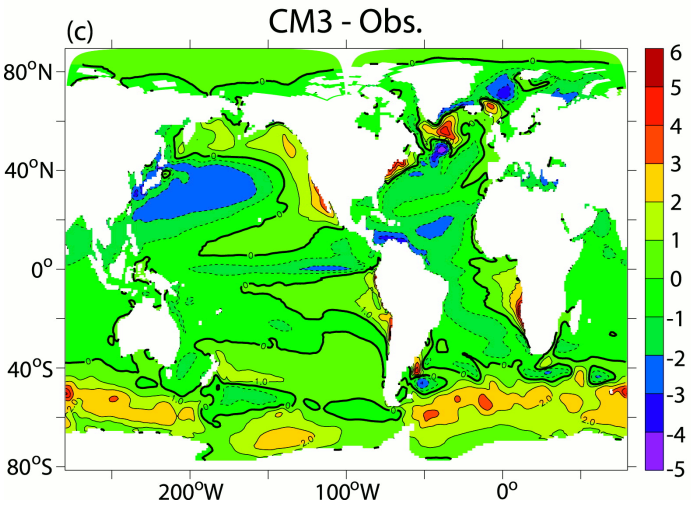


Abbildung 4.1: Darstellung von Abweichungen der Meeresoberflächentemperatur zwischen Modellergebnissen und Beobachtungsdaten (Donner u. a., 2011)

1996) herausgegeben. Neben den Beobachtungs- und Reanalysedaten der Wetterdienste, kommen auch Daten aus wissenschaftlichen Studien als Vergleichsgrößen in Frage. Für eine umfassende Evaluation müssen in der Regel mehrere Datenquellen verwendet werden, um Vergleichswerte für verschiedene klimatologische Variablen zu erhalten.

Auch in den *Model Intercomparison Projects* (siehe Abschnitt 2.2.3) werden die Modellergebnisse nicht nur untereinander, sondern auch mit Beobachtungsdaten verglichen. Die hier verwendeten Daten bilden oft einen Standard, der auch in der täglichen Arbeit in den Instituten häufig herangezogen wird.

Um Ergebnisse zu erhalten, die sich sinnvoll mit Beobachtungsdaten vergleichen lassen, sind i.d.R. längere, umfangreichere Modellläufe nötig. Daraus werden Mittelwerte über sinnvolle Zeiträume gebildet und die Abweichungen zu den Vergleichsdaten berechnet. Üblich sind auch Visualisierungen der räumlichen Verteilung dieser Abweichungen, sog. *Fehlerdiagramme (bias plots)*. Abbildung 4.1 zeigt ein Beispiel eines solchen Fehlerdiagramms.

Ursprünglich aus der numerischen Wettervorhersage kommt die Berechnung von *Performancemetriken*[7], auch *Skill Scores* genannt, die mit wenigen Werten die Güte von Modellergebnissen beschreiben. Dazu werden statistische Methoden verwendet, um die mittlere Abweichung der Ergebnisse zu bestimmen. Da die Prognosen von Wettervorhersagemodellen innerhalb kurzer Zeit mit dem tatsächlich eingetretenen Wetter verglichen werden können, gehört die kontinuierliche Berechnung der Vorhersagegüte

[7]Performance wird hier nicht im Sinne von Rechenleistung, sondern als Güte der Ergebnisse verstanden.

hier zum Alltag. Seit einigen Jahren kommen solche Metriken auch in der Evaluation von Klimasimulationen zum Einsatz (siehe z.B. Gleckler, Taylor und Doutriaux, 2008; Pincus u. a., 2008; Reichler und Kim, 2008).

4.1.2 Vergleich mit anderen Modellen

Neben dem Vergleich mit Beobachtungsdaten spielen auch Vergleiche mit anderen Modellen eine wichtige Rolle bei der Evaluation. Ein zentrales Element sind hier die global organisierten MIPs. Weitere gebräuchliche Vergleichsmöglichkeiten sind u.a.:

- der Vergleich mit ausgereifteren Modellen (z.B. Vorgängermodelle derselben Forschungseinrichtung)
- der Vergleich mit Modellen, die mit anderen Gittern oder Auflösungen arbeiten
- der Vergleich mit Modellen, die ähnliche oder sehr unterschiedliche Ansätze für bestimmte Aspekte verwenden
- der laufende Vergleich mit Vorversionen desselben Modells
- der Vergleich von verschiedenen Konfigurationen desselben Modells

4.1.3 Idealisierte Experimente

Insbesondere, wenn ein neuer dynamischer Kern entwickelt oder ein bestehender verbessert wird, wird dieser mit idealisierten Experimenten validiert. Für diese ist das erwartete Verhalten so deterministisch, dass eine Abweichung leicht erkennbar ist, oder es besteht sogar eine analytische Lösung, mit der das Ergebnis des numerischen Modells verglichen werden kann. Im Folgenden werden einige verbreitete Verfahren vorgestellt.

Flachwassermodell

Einen ersten Schritt in der Entwicklung eines dynamischen Atmosphärenkerns bilden Modelle auf Basis der sog. *Flachwassergleichungen (shallow-water equations)*. Die Flachwassergleichungen beschreiben die horizontalen Bewegungen eines inkompressiblen Fluids unter der vereinfachenden Annahme der Barotropie, d.h. die Temperatur sei allein abhängig vom Druck und keine unabhängige Variable (Hantel, 2013, S. 217). Übertragen auf eine rotierende Kugel ergibt sich ein einfaches Modell zum Testen numerischer Methoden zur zweidimensionalen Beschreibung atmosphärischer

Strömungen, da sie einige der wesentlichen Schwierigkeiten bzgl. der Modellierung horizontaler Strömungen hervorheben (vgl. Williamson u. a., 1992).

Williamson u. a. (1992) beschreiben sieben Standardtests ansteigender Komplexität für Verfahren zur Lösung der Flachwassergleichungen. Für vier der sieben vorgeschlagenen Tests gibt es analytische Lösungen, für die anderen drei müssen hingegen Vergleichsmodelle als Referenz herangezogen. Diese Testsuite gehört zum Standard bei der Evaluation von dynamischen Kernen von Atmosphärenmodellen in deren frühen Entwicklungsphasen (siehe z.B. Lin und Rood, 1997; Tomita, Tsugawa u. a., 2001; Ripodas u. a., 2009).

Deterministische Tests des dynamischen Kerns

Der nächste Schritt nach dem Test des barotropen, zweidimensionalen Flachwassermodells sind deterministische Tests des baroklinen 3D-Modells der Atmosphäre. Seit den 2000er Jahren wurden hierfür mehrere Standardtests vorgeschlagen (vgl. Williamson, 2007). Ein etabliertes Beispiel hierfür ist der *Jablonowski-Williamson baroclinic wave test*:

Christiane Jablonowski und David L. Williamson (2006a,b) schlagen dabei einen zweistufige Teststrategie vor. Zunächst wird der dynamische Kern des Atmosphärenmodells mit einem stabilen Anfangszustand, welcher vollständig analytisch beschrieben ist, initialisiert, um anschließend 30 Tage zu simulieren. Der stabile Zustand sollte über die gesamte Laufzeit beibehalten werden, bzw. die schleichenden Abweichungen hiervon so klein wie möglich sein. Im zweiten Schritt wird der stabile Zustand durch einen zonal, d.h. breitengradparallelen Wind gestört, wodurch eine barokline Welle entstehen sollte[8]. Auch wenn es für dieses Szenario keine analytische Lösung gibt, sollte sich zumindest in den ersten 10 Tagen der Simulation ein weitestgehend deterministisches Verhalten zeigen. Als Referenzlösung bieten die Autoren die Ergebnisse von vier verschiedenen hochaufgelösten Modellen an. Um die Konvergenzeigenschaften des getesteten dynamischen Kerns zu untersuchen sollten beide Testschritte mit unterschiedlichen Auflösungen durchgeführt werden.

Held-Suarez-Test

Ein Standardtest für das Verhalten von dynamischen Kernen von Atmosphärenmodellen in Langzeitsimulationen, sprich in Klimasimulationen, stammt von Isaac M.

[8] Im Gegensatz zu *barotrop* bedeutet *baroklin*, dass Flächen gleichen Drucks und Flächen gleicher Temperatur nicht parallel verlaufen sondern sich überschneiden. Bei einer barokline Welle ist demnach die Welle im Druckfeld gegenüber der Welle im Temperaturfeld verschoben (vgl. Brunotte u. a., 2001, S. 120f).

Held und Max J. Suarez (1994). Mit Hilfe des Held-Suarez-Tests soll die Dynamik eines AGCMs von der Physik isoliert werden, indem die physikalische Parametrisierung durch eine idealisierte Physik in Form fest vorgegebener Antriebsfunktionen ersetzt wird (vgl. Jablonowski, 1998). Die vorgeschlagene Simulationsdauer beträgt 1.200 Tage. Es handelt sich um einen sog. statistischen Test, da der Verlauf der Simulation nicht deterministisch ist. Das Modellklima wird über den Durchschnitt der letzten 1.000 Tage ermittelt und kann nur im Vergleich zu den Testergebnissen anderer Modelle, verschiedener Modellkonfigurationen oder in Bezug auf die Konvergenz bei Erhöhung der Auflösung bewertet werden. Die ersten 200 Tage werden dabei nicht berücksichtigt, um den Einfluss modellspezifischer Initialisierungsphasen zu eliminieren.

Wasserplanet

Um gezielt das Zusammenspiel von dynamischem Kern und der Physik eines Atmosphärenmodells zu testen, ohne dass die Ergebnisse von zu komplexen Randbedingungen beeinflusst werden, werden Experimente mit Planeten ohne Landmassen, sog. Wasserplaneten (Aqua-Planet) durchgeführt. Hierbei wird mit einfachen Verteilungen der Meeresoberflächentemperatur gearbeitet und die entstehenden Strömungs- und Niederschlagsregime beobachtet, wie etwa die Ausprägung der Innertropischen Konvergenzzone. So kann beispielsweise das Verhalten verschiedener Physikimplementationen untersucht werden oder der Einfluss einzelner Parameter wie der Meeresoberflächentemperatur, der Erdrotation oder der Auflösung des Modells (vgl. Hess, Battisti und Rasch, 1993). Aqua-Planet-Modelle füllen somit die Lücke zwischen stark vereinfachten Evaluationen, bei denen entweder nur der dynamische Kern oder nur die Physik getestet werden, und einem vollständigen Atmosphärenmodell (vgl. Blackburn und Hoskins, 2013).

4.2 Fallstudien

Die im Folgenden beschriebene Studie wurde an vier Forschungsinstituten durchgeführt:

- dem *Max-Planck-Institut für Meteorologie (MPI-M)* in Hamburg
- dem *Geophysical Fluid Dynamics Laboratory (GFDL)* in Princeton, USA
- der *Japan Agency for Marine Earth Science and Technology (JAMSTEC)* in Yokohama, Japan
- dem *RIKEN Center for Computational Science (R-CCS)* in Kobe, Japan

In diesem Abschnitt werden die an diesen Instituten untersuchten Fallstudien bzw. die dazugehörigen Modelle vorgestellt.

4.2.1 ICON

Das Max-Planck-Institut für Meteorologie entwickelt zusammen mit dem *Deutschen Wetterdienst (DWD)* und anderen Instituten das Modellierungsframework *ICON (ICOsahedral Non-hydrostatic)*, welches sowohl ein Atmosphären- als auch ein Ozeanmodell enthält (siehe auch Zängl u. a., 2015). Beide Modelle verwenden ein Dreiecksgitter basierend auf einem Ikosaeder (siehe auch Abbildung 2.3b). Das ICON-Atmosphärenmodell wird sowohl für die Klimaforschung als auch für die Wettervorhersage eingesetzt. Seit 2015 ist es das operative Wettervorhersagemodell des DWD (vgl. DWD, 2015). ICON nimmt an verschiedenen Teilprojekten von CMIP6 teil, u.a. als Basis für das Erdsystemmodell *ICON-ESM*. Am MPI-M werden weitere Modelle entwickelt und gepflegt, wie etwa die älteren Atmosphärenmodelle *ECHAM5* und *ECHAM6*. Im Fokus dieser Studie steht jedoch das ICON-Entwicklungsteam.

Zu den weiteren an ICON beteiligten Instituten gehören u.a. das DKRZ, das CSCS sowie das *Institut für Meteorologie und Klimaforschung - Department Troposphärenforschung* des *Karlsruher Instituts für Technologie (KIT)*.

4.2.2 GFDL-Modelle

Am GFDL werden mehrere Modelle entwickelt und gepflegt. *AM4* ist das neueste Atmosphärenmodell und basiert auf einem Würfelgitter (Abbildung 2.3a) (siehe auch Zhao u. a., 2018a,b). Die aktuelle Generation des GFDL-Ozeanmodells heißt *MOM6 (Modular Ocean Model)*. Beide Modelle sind sowohl im aktuellen gekoppelten Modell *GFDL-CM4* sowie im Erdsystemmodell *GFDL-ESM4* enthalten. Mit diesen Modellen beteiligt sich das GFDL auch an CMIP6. Der Fokus der Studie liegt auf der Entwicklung von CM und ESM sowie von MOM6. Letzteres wird deshalb separat betrachtet, da dessen Entwicklungsteam einige besondere Testverfahren verwendet, die anderswo nicht anzutreffen waren.

4.2.3 MIROC

JAMSTEC pflegt zusammen mit mehreren anderen Instituten das Modell *MIROC (Model for Interdisciplinary Research on Climate)*, eines der dienstältesten und ausgereiftesten Modelle Japans (siehe auch Tatebe u. a., 2019). MIROC basiert auf einem

traditionellen Längen-/Breitengrad-Gitter. In Gestalt des gekoppelten Modells *MIROC6* sowie des Erdsystemmodells *MIROC-ES2* nimmt es auch an CMIP6 teil.

4.2.4 NICAM

In Kooperation mit dem R-CCS und dem *Atmosphere and Ocean Research Institute (AORI)* der Universität Tokio ist JAMSTEC außerdem an der Entwicklung von *NICAM (Nonhydrostatic Icosahedral Atmospheric Model)* beteiligt (siehe auch Satoh, Tomita u. a., 2014). NICAM ist ein hochauflösendes Atmosphärenmodell, das u.a. die Simulation von Wolken ermöglicht. Wie ICON verwendet auch NICAM ein ikosaederbasiertes Dreiecksgitter. NICAM nimmt an dem in CMIP6 eingebetteten MIP für hochauflösende Modelle teil (*HighResMIP*, siehe auch Haarsma u. a., 2016).

4.2.5 SCALE

R-CCS arbeitet außerdem an der Bibliothek *SCALE (Scalable Computing for Advanced Library and Environment)*, einem Open-Source-Framework für globale und regionale Modelle, das sowohl Infrastruktur- als auch Physikmodule enthält. Zur Zeit der Studie lag der Fokus auf der Arbeit an dem regionale Modell *SCALE-RM* (siehe auch Nishizawa u. a., 2015; Sato u. a., 2015). Ein globales Open-Source-Modell (*SCALE-GM*), basierend auf dem dynamischen Kern von NICAM war ebenfalls in Vorbereitung.

4.3 Vorgehen

Zu Beginn dieser Studie habe ich mit dem ICON-Team am MPI-M zusammengearbeitet, um deren Softwareentwicklungspraktiken zu untersuchen. Dort habe ich verschiedene Interviews geführt und an Entwicklertreffen teilgenommen. Zudem habe ich im Jahr 2014 eine Umfrage zum Thema Testen unter ICON-Entwicklerinnen und -entwicklern durchgeführt (auch unter Beteiligung des DWDs und anderer Partnerinstitute; siehe Anhang A).

Mit einem engeren Fokus auf Testpraktiken habe ich dann 2017 die anderen Institute besucht und dort semistrukturierte Interviews mit mehreren Entwicklerinnen und Entwicklern geführt. Um sich auf die Fragen und Diskussionen vorzubereiten, haben die Entwicklungsteams zuvor eine Liste von Fragen erhalten (siehe Anhang B). Die Ausgangsfrage der Studie war: Wie testen EntwicklerInnen von Klimamodellen ihre Software? Da Testen im Sinne dieser Arbeit aus dem Ausführen von Software sowie

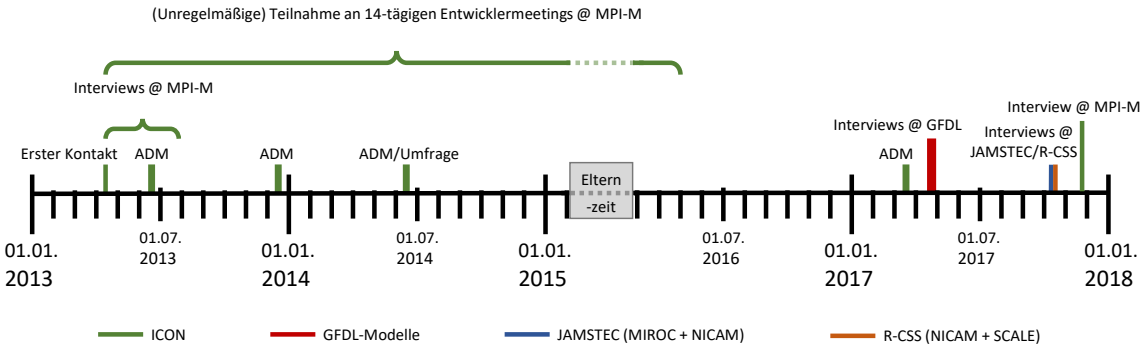


Abbildung 4.2: Zeitleiste der Studie zur Testpraxis in der Klimamodellierung. Auch zwischen den Ereignissen fand ein Austausch mit EntwicklerInnen statt. ADM = Teilnahme am ICON All Developers Meeting

der Beobachtung und Bewertung ihres Verhaltens besteht, ergaben sich als Leitfragen der Interviews:

- Was wird ausgeführt?
- Wie wird das Verhalten beobachtet?
- Wie wird das Verhalten bewertet?

Darüber hinaus habe ich nach dem allgemeinen Entwicklungsprozess gefragt und wie das Testen in diesen eingebettet ist, sowie danach welche Werkzeuge hierbei verwendet werden. Die Interviews wurden in der Regel als Gruppeninterviews mit mehreren EntwicklerInnen geführt. Dabei wurden nicht schematisch immer dieselben Fragen gestellt, sondern offene Diskussionen geführt, geleitet durch meine Fragen. Ziel war es ein vertieftes Verständnis der spezifischen Entwicklungs- und Testpraxis zu entwickeln. Nach den Besuchen beim GFDL, bei JAMSTEC und am R-CCS habe ich ein weiteres Interview mit einer ICON-Entwicklerin geführt, um meinen Kenntnisstand zu aktualisieren und Fragen zu klären, die während der Diskussionen mit den anderen Gruppen aufgekommen sind.

Abbildung 4.2 zeigt eine Zeitleiste der Arbeit an dieser Studie. Zusätzlich zu den Feldbeobachtungen, Umfragen und Interviews habe ich Dokumente studiert. Dazu gehört u.a. Modell- und Projektbeschreibungen, Dokumentationen, Quellcode, wissenschaftliche Publikationen, Vortragsfolien, Webseiten und Wikis. Nach Zusammenführung der Ergebnisse habe ich diese mit Vertretern der untersuchten Teams rückgekoppelt, um etwaige Missverständnisse aufzudecken und einzelne Verständnisfragen zu klären.

4.4 Teamstruktur und Entwicklungsprozesse

Um die Rolle des Testens besser zu verstehen, werden hier zunächst die Strukturen und allgemeine Entwicklungsprozesse der einzelnen Teams beschrieben. Tabelle 4.1 auf Seite 94 fasst die Ergebnisse zusätzlich zusammen. Anschließend werden in den darauffolgenden Abschnitten verschiedene Teststrategien beleuchtet.

4.4.1 ICON

Das ICON-Team teilt sich in sieben Untergruppen am MPI-M, DWD und den anderen Instituten. Diese sind jeweils für verschiedene Aspekte innerhalb des Modells verantwortlich, so gibt es etwa eine Atmosphärengruppe, eine Ozeangruppe, eine Infrastrukturgruppe etc. Es gibt ungefähr 30 Kernentwicklerinnen und -entwickler. Aufgrund von Fluktuation und wechselnden Projekten kann eine genaue Zahl jedoch nicht ermittelt werden. Seit etwa Ende 2016 verwenden die ICON-Gruppen *git* (git-scm.com) für die Versionskontrolle. Zuvor wurde *Subversion (SVN)* (subversion.apache.org) eingesetzt. Eine Analyse des SVN-Repositorys zeigt, dass es neben den KernentwicklerInnen noch zahlreiche weitere Beteiligte gibt. Bis zum Jahr 2016 steigerte sich die Zahl der aktiven Benutzerkonten auf über 70 pro Jahr, unter Berücksichtigung aller Entwicklungszweige (Abbildung 4.3). Wie die Umfrage im Jahr 2014 ergab, verfügt ein großer Teil der EntwicklerInnen über langjährige Erfahrung als ProgrammiererInnen (Abbildung 4.4).

Neben git stehen *Redmine* (redmine.org) als Ticketsystem und Wiki sowie *Buildbot* (buildbot.net) als Continuous-Integration-Server zu Verfügung. Letzter erstellt tägliche Builds mit verschiedenen Compilern auf allen wichtigen Rechnerplattformen und führt dazu unterschiedliche Tests aus. Jede Gruppe hat einen eigenen Entwicklungszweig innerhalb des git-Repositories und veröffentlicht eigene stabile ICON-

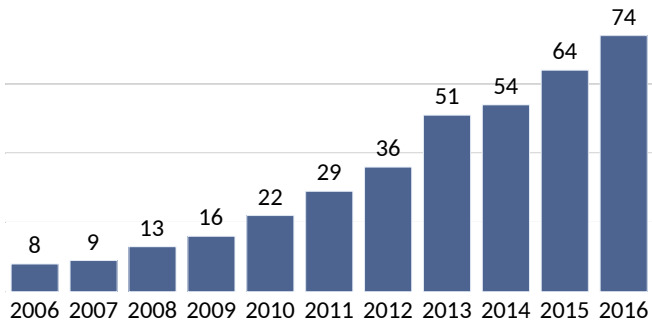


Abbildung 4.3: Anzahl der Benutzerkonten des ICON-SVN-Repositorys, mit denen mindestens ein Commit in einen beliebigen Entwicklungszweig getätigt wurde.

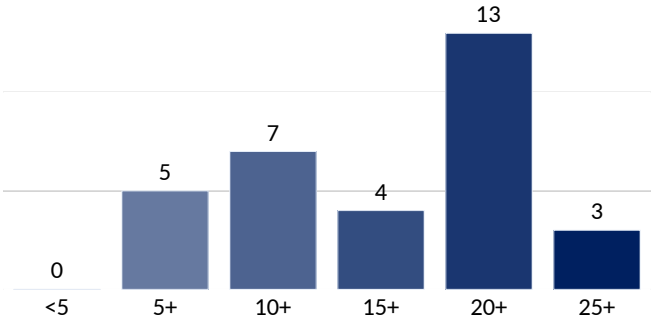


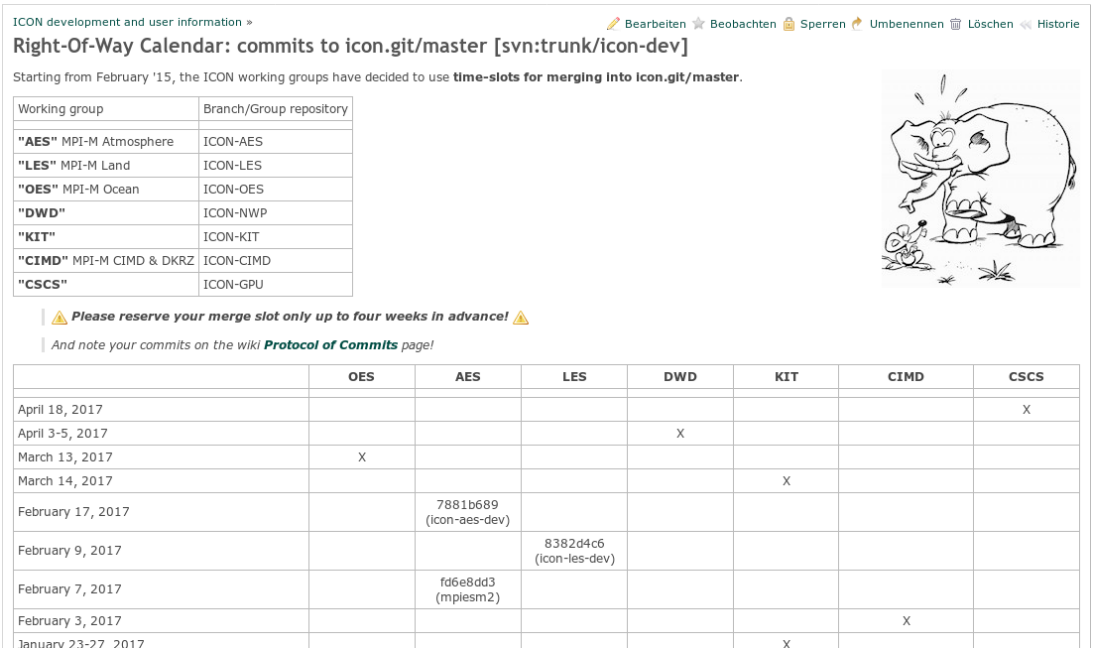
Abbildung 4.4: Programmiererfahrung der ICON-EntwicklerInnen in Jahren, Umfrage aus dem Jahr 2014 (siehe auch Anhang A)

Versionen (*Releases*). Darüber hinaus gibt es einen gemeinsamen Entwicklungszweig, der nach dem jeweils kommenden gemeinsamen Release benannt ist. Die Releases werden in den Wochen vor dem halbjährlich stattfinden, mehrtägigen Treffen aller ICON-EntwicklerInnen von MPI-M, DWD und den anderen Instituten (*All Developers Meeting*) vorbereitet und kurz nach diesen Treffen veröffentlicht.

Es gibt einen dokumentierten Ablauf, wie Änderungen in den gemeinsamen Entwicklungszweig eingefügt werden sollen. Jede Gruppe hat einen sog. *Gatekeeper*, der für das Einfügen der Codeänderungen seiner Gruppe verantwortlich ist. Um gleichzeitiges Einfügen zu vermeiden, gibt es im Wiki einen sog. *Right-Of-Way-Kalender*, in dem die Gatekeeper Zeit zum Einfügen von Code in den gemeinsamen Entwicklungszweig reservieren können (Abbildung 4.5). Es gibt eine gemeinsame Sammlung von Regressionstests, mit der jede Codeeinfügung überprüft werden muss. Es ist Aufgabe der Gatekeeper diese Testsammlung mit Hilfe des Buildbots auszuführen. Sechs Wochen vor dem gemeinsamen Release dürfen keine Änderungen mehr eingefügt werden (*Feature Freeze*) und es beginnt eine Phase des umfangreichen Testens durch alle Gruppen.

4.4.2 GFDL-CM4/ESM4

Das GFDL verfügt ebenfalls über mehrere Abteilungen und Entwicklungsteams, die ihre Arbeit und Codeänderungen koordinieren müssen. Aktuell gibt es etwa 35 KernentwicklerInnen in fünf Teams. Die meisten von ihnen arbeiten vor Ort in Princeton. Hinzu kommen Post-Docs und DoktorandInnen, sowie einige Personen, die früher einmal am GFDL waren, aber immer noch zum Code beitragen. Atmosphären- und Ozeanmodell sind nicht so eng miteinander verbunden wie bei ICON, wo sich die Modelle Gitter und Infrastruktur teilen. Hier werden sie in getrennten Coderepositories entwickelt und nur für das gekoppelte Modell und das Erdsystemmodell zusammengebracht. Die Modelle teilen sich zwar auch Infrastrukturcode, aber dieser ist einem



ICON development and user information »

Bearbeiten Beobachten Sperren Umbenennen Löschen Historie

Right-Of-Way Calendar: commits to icon.git/master [svn:trunk/icon-dev]

Starting from February '15, the ICON working groups have decided to use **time-slots for merging into icon.git/master**.

Working group	Branch/Group repository
"AES" MPI-M Atmosphere	ICON-AES
"LES" MPI-M Land	ICON-LES
"OES" MPI-M Ocean	ICON-OES
"DWD"	ICON-NWP
"KIT"	ICON-KIT
"CIMD" MPI-M CIMD & DKRZ	ICON-CIMD
"CSCS"	ICON-GPU

Please reserve your merge slot only up to four weeks in advance!

*And note your commits on the wiki **Protocol of Commits** page!*

	OES	AES	LES	DWD	KIT	CIMD	CSCS
April 18, 2017							X
April 3-5, 2017				X			
March 13, 2017	X						
March 14, 2017					X		
February 17, 2017		7881b689 (icon-aes-dev)					
February 9, 2017			8382d4c6 (icon-les-dev)				
February 7, 2017		fd6e8dd3 (mpiesm2)					
February 3, 2017						X	
January 23-27, 2017					X		

Abbildung 4.5: Right-Of-Way-Kalender im internen ICON-Wiki

Framework ausgelagert, dem *Flexible Modeling System (FMS)*, das ebenfalls separat entwickelt wird. GFDL verwendet ebenfalls git für die Versionskontrolle sowie GitLab (gitlab.com) für die Repositoryverwaltung, wobei die Funktionen, die GitLab bereit stellt, wie etwa ein Ticketsystem, ein Wiki oder Continuous Integration, nicht von allen Teams verwendet werden.

Verantwortlich für FMS und das Zusammenfügen von CM und ESM ist die GFDL-Abteilung *Model Systems Group (MSG)*. Diese Abteilung dient als Zentrale, die die Änderungen aller anderen Gruppen zusammenführt. Die Mitarbeiterinnen und Mitarbeiter von MSG verfügen ebenfalls über Ausbildungen in den fachlichen Disziplinen, haben aber eine höhere technische Affinität oder sogar Erfahrungen in der industriellen Softwareentwicklung. Sie sehen sich mehr als ProgrammiererInnen und weniger als WissenschaftlerInnen. Für jede der anderen (wissenschaftlichen Gruppen) gibt es in MSG einen designierte Verbindungsperson, die sog. *Model Liaison*. Die Model Liaison diskutiert aktuelle Modelländerungen mit ihrer zugewiesenen Gruppe, nimmt an deren Gruppentreffen teil und pflegt die Änderungen ins CM- und ESM-Repository ein.

Die Model Liaisons sind außerdem für das Testen ihrer Änderungen zuständig. Die GitLab-Umgebung startet nach jedem Commit automatische Builds und führt eine Basissammlung an Regressionstest aus. Darüber hinaus pflegt jede Model Liaison ihre eigene Testsuite. Die Auswahl der Testfälle hängt von den jeweiligen Änderungen ab und geschieht in Absprache mit der jeweiligen Wissenschaftlergruppe. Diese

benennen beispielsweise einzelne Experimente, die von besonderem Interesse sind. Wenn eine Model Liaison mit dem Einfügen und Testen fertig ist, wird die aktuelle Modellversion im Repository als Release-Kandidat markiert und weitergehend von den anderen Mitarbeiterinnen und Mitarbeitern von MSG getestet. Das Einfügen von Änderungen der wissenschaftlichen Gruppen ins CM- und ESM-Repository ist kein kontinuierlicher Prozess, sondern geschieht von Zeit zu Zeit auf Anfrage einer der Gruppen. Ein- oder zweimal im Jahr werden ausführlich getestete Hauptversionen erstellt. Diese werden am GFDL auch *City Release* genannt, da sie jeweils nach Städten (mit Initialen in alphabetischer Reihenfolge) benannt sind.

4.4.3 MOM6

Die meisten der wissenschaftlichen Entwicklergruppen am GFDL haben keine formellen Testprozesse. Die Entwicklerinnen und -entwickler sind verantwortlich dafür, ihre eigenen Änderungen nach eigenem Ermessen zu testen. Eine Ausnahme ist das Team des Open-Source-Ozeanmodells MOM6, welches über die ausführlichste Sammlung von Regressionstests verfügt. MOM6 ist öffentlich verfügbar über GitHub (GitHub, MOM6), aber wird intern mit Hilfe von GitLab gepflegt. Alle Änderungen werden jedoch an GitHub in Form von *Pull Requests* gesendet. Jeder Pull Requests wird von kleinen, schnellen Tests mit Hilfe des mit GitHub verbundenen CI-Dienst *Travis CI* (TravisCI, MOM6) automatisch überprüft. Pull Requests von externen EntwicklerInnen müssen zuerst die TravisCI-Tests bestehen, bevor sie von einer Kernentwicklerin oder einem Kernentwickler inspiziert werden. Diese müssen hierzu, genauso wie für ihre eigenen Pull Requests, eine erweiterte Regressionstestsammlung ausführen. Bei eigenen Pull Requests geschieht dies bevor dieser abgesendet wird. Die erweiterte Regressionstestsammlung wird in GitLab gepflegt und wird dort einmal täglich sowie nach Bedarf ausgeführt.

Weitere ausgefeiltere Tests werden wöchentlich von einem *Jenkins-CI-Server* (jenkins.io) ausgeführt, welcher von Kooperationspartner in Australien an der *National Computational Infrastructur (NCI)* aufgesetzt wurde. Diese enthalten unter anderem Simulationen in einer virtuellen Valgrind-Umgebung (valgrind.org), um Speicherfehler aufzudecken.

4.4.4 MIROC

Der Entwicklungsprozess von MIROC ist deutlich informeller als die der vorangehend beschriebenen Teams. Dies liegt möglichweise auch an der kleineren Teamgröße. Es gibt zwei Teams, eins für das gekoppelte Modell MIROC6 und eins für das Erdsystemmodell MIROC-ES2. Beide Teams bestehen aus Leuten von JAMSTEC und

verschiedenen anderen japanischen Forschungseinrichtungen. Die Entwicklung ist vor allem auf die Teilnahme an CMIP ausgerichtet. Das MIROC6-Team besteht aus ca. fünf KernentwicklerInnen, das MIROC-ES2-Team aus etwa 10. Auch hier sind die Zahlen schwankend. Die Teams haben ein gemeinsames Mercurial-Repository (siehe auch mercurial-scm.org) für die Versionskontrolle und ein Wiki für Dokumentationen. Alle EntwicklerInnen haben Zugriff zum gemeinsamen Hauptentwicklungszweig. Es gibt keine regelmäßigen Releases, aber stabile Versionen werden innerhalb des Repositories markiert und nummeriert. Nach finaler Evaluation der Güte der Ergebnisse für historische Klimaszenarien wird eine stabile Version veröffentlich und für die CMIP6-Experimente verwendet. Die CMIP-Experimente sind dementsprechend die Haupttestfälle für MIROC.

4.4.5 NICAM

Der NICAM-Entwicklungsprozess ist ebenfalls eher informell. Das Team, ungefähr zehn Kernentwicklerinnen und -entwickler an verschiedenen Instituten, verwendet git für die Versionskontrolle und verfolgt ein *git Flow* genanntes Verzweigungsmodell, mit Master-, Entwicklungs- und Featurezweig (siehe auch Driessen, 2010). Es gibt ebenfalls ein Redmine, aber dieses wird nicht so viel genutzt. Der Großteil der Kommunikation geschieht über eine Mailingliste. Jede EntwicklerIn hat Zugriff auf den gemeinsamen Entwicklungszweig. Nachdem sie größere Änderungen vorgenommen haben, teilen sie dies über die Mailingliste mit und bitten die anderen EntwicklerInnen die neue Version mit ihren typischen Konfigurationen zu testen. Fehler werden dann ebenfalls über Mailingliste diskutiert. Es gibt einen Hauptentwickler, der die Änderungen vom Entwicklungszweig in den Masterzweig überträgt und einmal im Jahr ein Release erstellt. Releasekandidaten werden zuvor mit einer erweiterten Testsammlung evaluiert.

4.4.6 SCALE

SCALE wird von einem kleinen Team von fünf Entwicklern am R-CCS entwickelt. Diese praktizieren ebenfalls ein git-Flow-artiges Entwicklungsmodell. Neben git und Redmine, welches aktiv als Ticketsystem und Wiki verwendet wird, gibt es außerdem einen Jenkins-CI-Server. Dieser erstellt tägliche Builds auf verschiedenen Rechnerplattformen, mit verschiedenen Compilern und verschiedenen Tests von Unittests, über idealisierte Experimente, bis zu realistischen Simulationen. Stabile Releaseversionen werden in unregelmäßigen Abständen auf der SCALE-Webseite (R-CCS, SCALE) veröffentlicht. Der Entwicklungsprozess ist dadurch formeller und werkzeugbasierter als der von NICAM, welches teilweise am selben Institut unter Mitarbeit von zum Teil denselben Entwicklern entwickelt wird. Die SCALE-Entwickler geben an, dass entscheidend für die Etablierung des Prozesses war, dass sie nach diesem

von Anfang an gearbeitet haben, und somit niemand überzeugt werden musste, sich umzustellen.

Modell	ICON	GFDL-CM4/ GFDL-ESM4	MOM6	MIROC6/ MIROC-ES2	NICAM	SCALE
Typ	CM/ESM	CM/ESM	Ozean	CM/ESM	Atmosphäre	Atmosphäre
Interviews an Institut	MPI-M	GFDL	GFDL	JAMSTEC	JAMSTEC, R-CCS	R-CCS
CMIP6-Teilnahme	Ja	Ja	Ja	Ja	Nur HighResMIP	Nein
Größe Kernteam	30	35	5	5 + 10	10	5
Untergruppen	7	5	1 + Open-Source-Community	2	1	1
Standort	Verteilt	Vorwiegend zentral	Verteilt	Verteilt	Verteilt	Zentral
Organisation des Entwicklungs-prozesses	Gatekeeper, Right-Of-Way-Kalender	Model Liaisons	Pull Requests	Ad Hoc	Ad Hoc, git Flow	Ad Hoc, git Flow
Versionskontrolle	git	git	git	Mercurial	git	git
Teamwerkzeuge	Redmine	GitLab	GitHub, GitLab	–	Redmine	Redmine
CI-Server	Buildbot	GitLab	TravisCI, GitLab, (Jenkins)	–	–	Jenkins

Tabelle 4.1: Modelle, Teams und Werkzeuge

4.5 Regressionstests

Der erste Test nach einer Änderung am Code ist immer: Kompiliert es? Alle untersuchten Modelle werden von WissenschaftlerInnen verschiedener Forschungseinrichtungen benutzt, unter der Verwendung verschiedener Compiler und verschiedener Hochleistungsrechner. Portabilität ist somit eine wichtige Anforderung. Da einzelne EntwicklerInnen normalerweise nicht Zugang zu allen relevanten Compilern und Computern haben, werden Portabilitätstests entweder zentral, d.h. entweder von einem CI-Server mit Zugang zu allen Plattformen, oder kollektiv, d.h. in Zusammenarbeit mehrerer EntwicklerInnen, durchgeführt. Der ICON-Buildbot testet beispielsweise 15 verschiedene Kombinationen von Plattformen und Compilern bzw. Compilereinstellungen.

4.5.1 Typische Testfälle

Der nächste Schritt besteht darin, einen Testfall auszuführen. Klimamodelle sind hochgradig konfigurierbar. Da nicht alle möglichen Kombinationen von Konfigurationsoptionen regelmäßig getestet werden können, haben alle Teams ihre Standardtestfälle, um die wichtigsten Konfigurationen, Auflösungen und Modellfeatures zu

überprüfen. Bei den meisten dieser Testfälle handelt es sich um E2E-Tests üblicher Experimente. Insbesondere die verschiedenen MIP-Konfigurationen spielen hier eine wichtige Rolle. Dabei ist es in der Regel ausreichend, kurze Zeiträume von wenigen Zeitschritten bis zu einem Monat zu simulieren. Klimamodelle reagieren sehr empfindlich auf die kleinsten Änderungen, so dass Programmierfehler sehr schnell sichtbar werden. Längere Simulationen oder Ensembles werden nicht regelmäßig im Rahmen von Regressionstest durchgeführt, sondern von Zeit zu Zeit von den EntwicklerInnen, die ExpertInnen für das jeweilige Experiment sind. Andere Regressionstest neben den MIP-Experimenten beinhalten etwa Experimente, die von aktuellem Interesse sind, beispielsweise für die Erstellung einer wissenschaftlichen Veröffentlichung, oder Experimente, die für bereits erschienene Veröffentlichungen erstellt wurden, oder idealisierte Standardtests.

4.5.2 NWP-Tests

Für Modelle, die sich auch zur numerischen Wettervorhersage eignen, wie ICON oder NICAM, wird auch die Vorhersagequalität regelmäßig überprüft. Für ICON, welches für die operative Vorhersage am DWD verwendet wird, geschieht dies in mehreren Schritten von Einzelvorhersagen bis hin zu parallelen, operativen Simulationen. Der NWP-Modus von NICAM wird wöchentlich von einem einzelnen Entwickler getestet.

4.5.3 Technische Tests

Neben der Funktionalität werden auch einige technische Aspekte im Rahmen der Regressionstests geprüft. Ein Beispiel ist der Checkpoint/Restart-Mechanismus, der in jedem Modell enthalten ist. Alle bis auf ein Team prüfen regelmäßig, dass das Unterbrechen einer Simulation und das anschließende Wiederanfahren von einem Checkpoint aus die Ergebnisse nicht beeinflusst.

Identische Ergebnisse über verschiedene Rechnerplattformen, Compiler oder Compilereinstellungen sind i.d.R. nicht zu erreichen. Kein einziges der interviewten Teams strebt dieses an. Ob die Anzahl der parallelen MPI-Prozesse und OpenMP-Threads die Ergebnisse beeinflussen darf, wird unterschiedlich gehandhabt. Während die ICON- und GFDL-Leute testen, dass dies nicht der Fall ist, werden bei NICAM und SCALE zwar auch unterschiedliche parallele Konfigurationen verglichen, nicht signifikante Abweichungen innerhalb eines festgelegten Bereichs jedoch zugelassen. Die MIROC-EntwicklerInnen überprüfen zwar auch, dass es keine signifikanten Abweichungen gibt, jedoch ist die Anzahl der Prozesse und Threads innerhalb einer Serie von Experimenten ohnehin konstant. In ICON gibt es sogar einen speziellen Testmodus *P-Test*

genannt, bei dem eine serielle Version des Modells simultan zur parallelen Ausführung läuft. Jedes Mal, wenn die Prozesse des parallelen Modus Daten, d.h. Arrayinhalte, miteinander synchronisieren, wird überprüft, ob die Inhalte die gleichen sind wie bei der seriellen Ausführung.

4.6 Automatisierung

Der Grad der Automatisierung variiert zwischen den Teams. Wie gesehen, verwenden einige Teams einen CI-Server, um das Modell automatisch zu kompilieren und zu testen. Daneben verfügt das GFDL über ein umfangreiches Framework, die *FMS Runtime Environment (FRE)*, um ein komplettes Experiment in einer XML-Datei zu konfigurieren. Dazu gehören u.a. die Schritte zur Beschaffung und Kompilierung des Quellcodes, das Kopieren von Daten, die Ausführung des Modells, die Überprüfung der Ergebnisse und das Postprocessing. FRE liest die XML-Dateien ein und generiert daraus die notwendigen Skripte für alle Einzelschritte. Es kann die Modellergebnisse automatisch mit Referenzdaten vergleichen und verschiedene MPI- und OpenMP-Einstellungen vergleichen sowie Checkpoint/Restart-Tests durchführen. Für das ICON-Atmosphärenmodell gibt es ebenfalls ein Skript, das derartige Tests für jedes beliebige Experiment durchführen kann. Zusätzlich verwenden einige EntwicklerInnen (aller Teams) ihre eigenen, persönlichen Testumgebungen, die unterschiedliche Grade der Automatisierung beinhalten.

4.7 Überprüfung der Modellergebnisse

Wenn ein Test abgeschlossen ist, müssen die Ergebnisse in den Ausgabedateien überprüft werden. In den Modellkonfigurationen wird festgelegt, welche Variablen in Dateien geschrieben werden und wie oft. Für die Überprüfung werden zwei Verfahren angewandt: automatischer bitweiser Vergleich mit Referenzdaten sowie die manuelle Überprüfung durch das Betrachten von im Postprocessing generierten Diagrammen. Zusätzlich zu den Ausgabedateien, die die Modellergebnisse enthalten, werden große Mengen an Debuginformationen in Protokolldateien geschrieben, die bei Bedarf angeschaut werden können. Einige Modelle enthalten ausgefeilte Debugmechanismen, die es erlauben, durch das Hinzufügen weniger Zeilen Code, jede Modellvariable zu jeder Zeit auszugeben.

4.7.1 Bitidentität

Die Verwendung von bitweisen Vergleichen variiert stark zwischen den untersuchten Teams. Das GFDL setzt sehr stark auf Bitidentitätsprüfungen, nicht nur für die oben beschriebenen technischen Tests, sondern auch um die Konsistenz der Ergebnisse zwischen aufeinanderfolgenden Modellversionen sicherzustellen. Beim GFDL sind Veränderungen der Ergebnisse nur zwischen den City Releases von CM und ESM erlaubt. Alle ergebnisverändernden Änderungen, die zwischen diesen Hauptversionen eingebracht werden, müssen über die Modellkonfiguration deaktivierbar sein. Dies gilt auch für Fehlerkorrekturen (*Bugfixes*) und führt beispielsweise zu Code wie dem Folgenden:

Beispiel 4.1: Deaktivierbarer Bugfix

```
1 IF (BUGFIX_XY_ENABLED) THEN
2   ! korrigierter Algorithmus
3   .
4 ELSE
5   ! fehlerhafter Algorithmus
6   .
7 END IF
```

Das Ziel dieses Vorgehens ist es, vorherige Experimente reproduzierbar zu halten. Dies wird mit einer erhöhten Codekomplexität bezahlt. In einige Fällen bleiben Fallunterscheidungen wie im Beispiel 4.1 über Jahre im Code, auch wenn sie gar nicht mehr benötigt werden. Die Möglichkeit Ergebnisse unter vergleichbaren technischen Randbedingungen zur reproduzieren und hierzu auch keine nichtdeterministischen parallelen Algorithmen zuzulassen, geht in einigen Fällen zudem zu Lasten der algorithmischen Effizienz. Einige GFDL-Entwickler sagen, sie hätten Angst, sie könnten durch Nichtdeterminismus auf Hardwareebene in zukünftigen Rechnersystemen die Bitreproduzierbarkeit als softwaretechnisches Werkzeug verlieren.

Andere Teams sind nicht so streng. Das ICON-Team prüft auch die bitweise Konsistenz der Ergebnisse zwischen aufeinanderfolgenden Versionen, erlaubt jedoch zu jeder Zeit begründete Abweichungen, z.B. durch geänderte Algorithmen, neue Parameter, neue Eingabedaten etc. Die MIROC- und NICAM-Teams führen systematisch keine derartigen Tests durch. MIROC-Entwickler haben jedoch berichtet, dass sie manchmal Ergebnisse auf Bitidentität hin vergleichen, etwa wenn sie an Infrastrukturcode arbeiten. Innerhalb der SCALE-Regressionstests werden nur die Unittests automatisch auf Bitidentität mit Referenzdaten verglichen.

4.7.2 Diagramme

Kein einziges Team verwendet automatische Prüfungen mit Toleranzen. Die einzige Alternative zum bitweisen Vergleich ist das manuelle Prüfen von Diagrammen. Alle Teams haben automatisierte Postprocessing-Routinen, um verschiedene Arten von Diagrammen für jede Ergebnisvariable zu erstellen. Um die Ergebnisse zu prüfen werden diese Diagramme häufig neben Referenzdiagrammen angezeigt. Eine andere Möglichkeit ist es Fehlerdiagramme, die Abweichungen zu den Referenzdaten zeigen, zu generieren.

Das NICAM-Team verwendet beispielsweise eine selbsterstellte Webanwendung, um die Variablen einer gewünschten Simulation zu durchsuchen und sie mit verschiedenen Referenzdatensätzen zu vergleichen. Abbildung 4.6 zeigt einen Screenshot dieser Anwendung.

Andere Teams verfügen über ähnliche Tools. Das SCALE-Team hat einen großen Bildschirm in seinem Büro hängen, auf dem im Wechsel verschiedene Diagramme des letzten Jenkins-Builds gezeigt werden (Abbildung 4.7). Beispielsweise nach der Mittagspause stehen die Teammitglieder zusammen vor dem Bildschirm und schauen, ob etwas schiefgegangen ist.

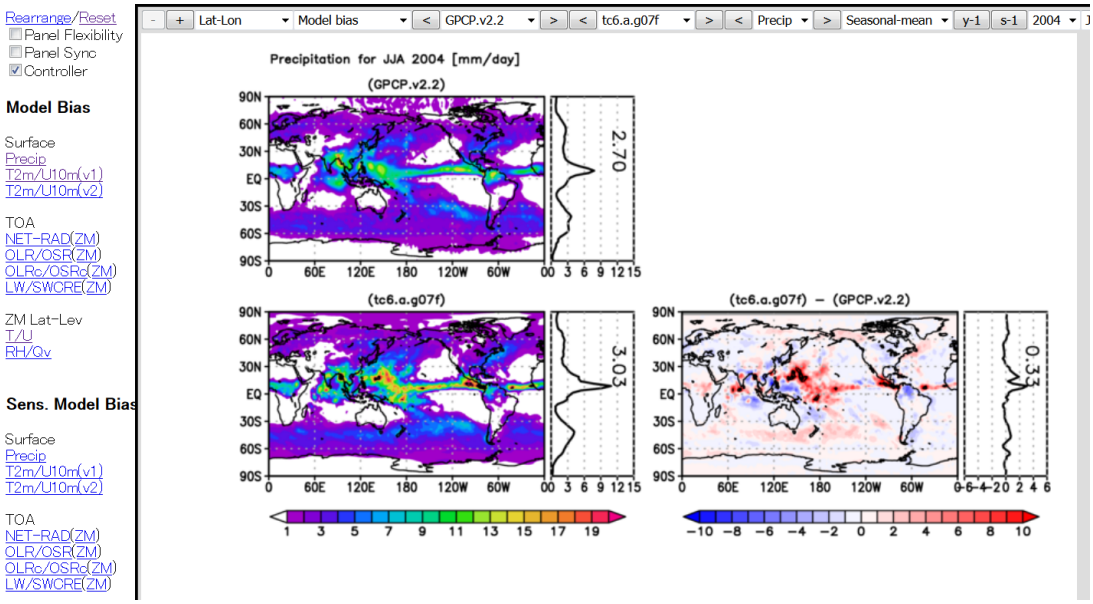


Abbildung 4.6: Ausschnitt aus der Webanwendung des NICAM-Teams zur Betrachtung von Diagrammen

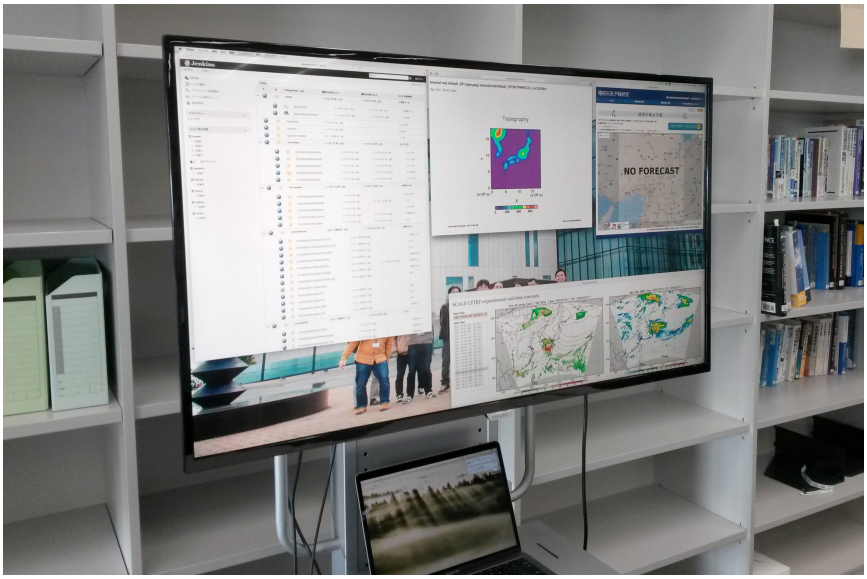


Abbildung 4.7: Bildschirm im SCALE-Büro. Angezeigt werden die Ergebnisse des letzten Jenkins-Build.

4.8 Besondere Testverfahren

Wie zuvor bereits erwähnt, hat das MOM6-Team die ausführlichste Testsammlung erstellt. Um Pull Request zu überprüfen, werden in der GitLab-Umgebung über einhundert Testfälle ausgeführt, hauptsächlich idealisierte Tests und Standardexperimente für Ozean- und gekoppelte Modelle. Zwei besondere Testverfahren sollen an dieser Stelle aufgrund ihrer Einzigartigkeit im Rahmen dieser Studie erwähnt werden.

4.8.1 Einheitentest

Zum einen wurde eine automatische Überprüfung der physikalischen Einheiten, wie etwa Meter, Joule, Pascal etc. in das Modell eingebaut. Dazu wird jeder Einheit eine Zweierpotenz, die zu Beginn des Tests zufällig gewählt wird, zugeordnet. Werte, die Variablen zugewiesen werden, werden bitweise verschoben, indem sie mit der ihrer Einheit zugehörigen Zweierpotenz multipliziert werden. Am Ende des Tests werden alle Werte durch Division zurückverschoben. Wenn man einen Test zweimal mit verschiedenen Zweierpotenzen ausführt, müssen die Ergebnisse übereinstimmen, anderenfalls liegt mit großer Wahrscheinlichkeit ein einheitenbezogener Programmierfehler vor. Man kann diese Bitverschiebung mit einer Skalenverschiebung, etwa einem Wechsel von Meter zu Zentimeter vergleichen, durch die Multiplikation mit Zweieranstelle von Zehnerpotenzen werden jedoch Rundungsfehler vermieden.

4.8.2 Domänentransformation

Der zweite Ansatz besteht darin, das Modell mit transformierten Domänen auszuführen. Dazu werden alle Daten, einschließlich Topografie, Richtungen, Randbedingungen etc. transformiert, beispielsweise indem sie um 90° gedreht werden (Abbildung 4.8). Wenn am Ende des Tests alle Daten zurücktransformiert werden, müssen die Ergebnisse identisch zum Originalmodus sein. Entstehende Abweichungen sind mit großer Wahrscheinlichkeit die Folge von Programmierfehlern bezogen auf Arrayindizes.

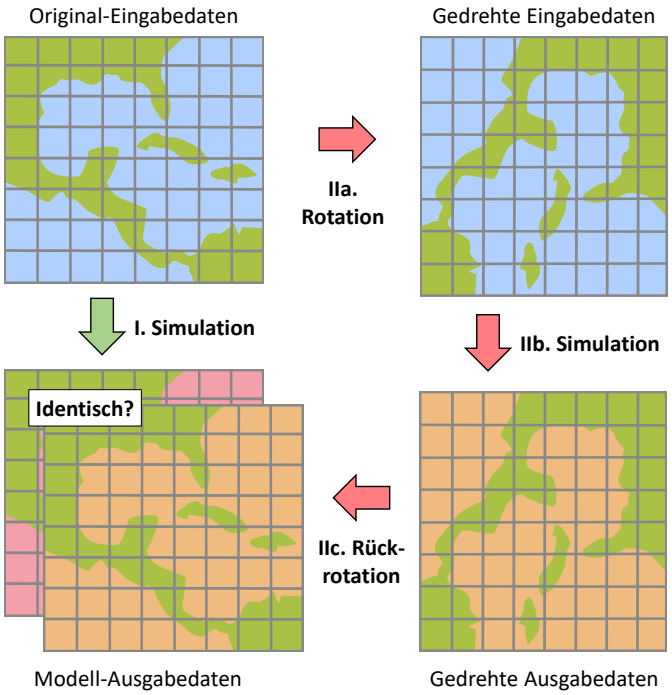


Abbildung 4.8: Veranschaulichung des MOM6-Rotationstests. Adaption von Hannah u. a. (2015)

4.9 Unittests

Unittests werden in der Entwicklung von Klimamodellen nicht häufig verwendet. Jedoch gaben alle Teams an, dass sie über ein paar Unittests für Infrastrukturbibliotheken oder -module verfügen. In den Regressionstestsammlungen von MOM6 und SCALE sind sogar einige Unittests für wissenschaftlichen Code vorhanden, jedoch nur sehr wenige. In der SCALE-Testsammlung gibt es zum Beispiel einen Unittest für die Schnittstelle des dynamischen Kerns des regionalen Modells. Der Test ruft die entsprechende Subroutine auf und übergibt als Argumente ausschließlich mit

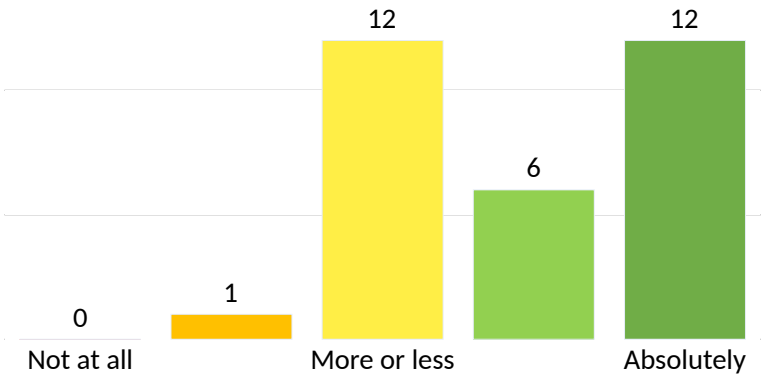


Abbildung 4.9: Antworten auf die Frage: „Do you think Unit Tests can be useful for ICON?“, Umfrage aus dem Jahr 2014 (siehe auch Anhang A)

Nullen oder anderen Konstanten gefüllte Arrays. Die Entwickler sagen, dass sogar solch einfache Tests bereits viele Fehler aufdecken.

In der Umfrage unter ICON-EntwicklerInnen im Jahr 2014 stellt nur eine von 31 befragten Personen die potenzielle Nützlichkeit von Unittests in Frage (Abbildung 4.9). Zudem äußerten Entwickler des CSCS, deren Aufgabe es ist, das ICON-Modell auf GPUs zu portieren, dass sie derartige Tests eigentlich dringend benötigten.

Wenn über Unittests gesprochen wird, meinen KlimamodellentwicklerInnen nicht zwangsläufig separate Testprogramme im Sinne dieser Arbeit. Es können auch sehr reduzierte Modellkonfigurationen gemeint sein, die es erlauben, nach einer grundlegenden Initialisierung eine bestimmte Prozedur aufzurufen. Das ICON-Modell verfügt beispielsweise über solch einen sog. *Testbed*-Modus und die MOM6-Unittests sind ebenfalls in das Modell integriert und werden ausgeführt, wenn bestimmte Konfigurationsoptionen gesetzt sind. Allerdings machen jeweils nur wenige EntwicklerInnen von diesen Möglichkeiten Gebrauch.

Es wurden verschiedene Gründe angegeben, weshalb Unittests nicht breiter verwendet werden. Einige sagen, dass ihr Modell so empfindlich auf Fehlerursachen reagiere, dass es zum Testen ausreicht, es in einer sinnvollen Konfiguration auszuführen. Andere, die Unittests offener gegenüber eingestellt sind, sagen, dass viele ihrer Kollegen sich der Vorteile nicht bewusst sind.

Ein wichtiger Grund für die Abwesenheit von Unittests scheint der Aufwand zu sein, der mit der Erstellung derartiger Tests verbunden ist. Viele Prozeduren in Klimamodellen benötigen eine riesige Menge an Eingabedaten, die korrekt diskretisiert und auf das Modellgitter abgebildet werden müssen. Dies geschieht normalerweise automatisch in der Modellinitialisierungsphase. Einen konsistenten und nützlichen Eingabedatensatz für eine einzelne Prozedur manuell aufzusetzen ist dagegen schwierig und würde sehr viel Zeit kosten.

Beispiel 4.2: Subroutine aus ICON

```
1  SUBROUTINE diffuse_hori_velocity(p_nh_prog, p_nh_diag, p_nh_metrics,     &
2                                   p_patch, p_int, prm_diag, ddt_u, ddt_v, &
3                                   d_11_ie, d_12_ie, d_13_ie, dt)
4
5    TYPE(t_nh_prog),    INTENT(in)         :: p_nh_prog
6    TYPE(t_nh_diag),    INTENT(in)         :: p_nh_diag
7    TYPE(t_nh_metrics),INTENT(in)          :: p_nh_metrics
8    TYPE(t_patch), TARGET, INTENT(inout)   :: p_patch
9    TYPE(t_int_state), INTENT(in),TARGET   :: p_int
10   TYPE(t_nwp_phy_diag),INTENT(inout)     :: prm_diag
11   REAL(wp),    TARGET, INTENT(inout)     :: ddt_u(:,:,:)
12   REAL(wp),    TARGET, INTENT(inout)     :: ddt_v(:,:,:)
13   REAL(wp), DIMENSION(:,:,:), INTENT(in) :: D_11_ie, D_12_ie, D_13_ie
14   REAL(wp),            INTENT(in)        :: dt
```

Hier dargestellt ist der Kopf einer typischen Prozedur aus dem ICON-Modell. Die Subroutine verfügt über zwölf Dummyargumente, von denen es sich bei einem um einen skalaren REAL-Wert handelt (`dt`) und bei fünf um dreidimensionale REAL-Arrays (`ddt_u`, `ddt_v`, `D_11_ie`, `D_12_ie`, `D_13_ie`). Die restlichen sechs Argumente haben wiederum einen benutzerdefinierten Verbunddatentyp. Zusammen verfügen diese Verbunddatentypen über ca. 700 Basiskomponenten, von denen rund 170 tatsächlich von der Subroutine verwendet werden[a]. Für einen Unittests dieser Subroutine müssten somit rund 175 Variablen, ein Großteil davon Arrays, mit Daten belegt werden, globale Variablen nicht mitgerechnet. Hierzu muss jedoch erst einmal bekannt sein, welche der rund 700 Basiskomponenten tatsächlich benötigt werden, was i.d.R. einer genauen Analyse der Subroutine sowie aller von ihr verwendeten Prozeduren bedarf.

[a] Die Anzahl aller Basiskomponenten und die der verwendeten wurden mit dem Softwarewerkzeug *FortranCallGraph*, welches später in dieser Arbeit vorgestellt wird, ermittelt.

Auch architektonische Gründe wurden von den EntwicklerInnen angegeben, z.B. dass es zu viele Abhängigkeiten oder zu viele globale Variable gäbe. Während Klimamodelle auf der einen Seite zwar stark modularisiert sind, gibt es auf der anderen Seite häufig auch eine starke Kopplung zwischen einzelnen Modulen, was es schwieriger macht, diese isoliert zu testen. Ein Entwickler vom GFDL-Landmodell berichtete, dass er manchmal Unittests erstellt, jedoch würde die Architektur des Modells dies auch besser unterstützen. Sie sei objektorientierter, mit Algorithmen, die auf einzelnen Kacheln des Gitters operieren, während Atmosphären- und Ozeanmodelle dahingehend optimiert sind, Arrays zu verarbeiten, die die Werte einzelner Variablen enthalten, jedoch für die gesamte Domäne (siehe Abschnitt 2.3.3). Wenn ein neues Modell von Grund auf neu gebaut würde, ließe sich unter Umständen eine besser testbare Architektur schaffen, jedoch basieren die meisten Modelle auf dem Code

von Vorgängermodellen und enthalten so sehr viel alten Code. Hinzu kommt, dass die WissenschaftlerInnen, die die Modelle entwickeln, keine ExpertInnen für Softwarearchitektur sind.

Das Testorakelproblem erschwert die Erstellung von Unittests zusätzlich. Auch für einzelne Prozeduren ist es häufig nicht möglich von vornherein die erwarteten Ausgaben anzugeben. Im Rahmen von Regressionstests können zwar auch hier Vorversionen als Testorakel verwendet werden, jedoch müssen diese dann wiederum auf andere Weise getestet werden, z.B. mit Hilfe eines E2E-Tests.

Anstelle von Unittests versuchen die EntwicklerInnen zum Testen mit minimalen Konfigurationen zu arbeiten, die nur die nötigen Modellkomponenten enthalten. Abhängig von der Art und Weise der jeweiligen Änderung ist jedoch nicht immer möglich, mit einer reduzierten Konfiguration zu testen. Beispielsweise beim Parameter-Tuning ist es manchmal notwendig, ein großes Experiment über eine längere Zeitskala durchzuführen und die Ergebnisse mit Beobachtungsdaten zu vergleichen, um festzustellen, ob die Abweichungen kleiner oder größer geworden sind.

Eine andere Form eines reduzierten Test sind sog. *Ein-Säulen-Modelle (single-column models)*, in denen einzelne Algorithmen nur auf einen einzelnen horizontalen Gitterpunkt oder -zelle angewendet werden, jedoch auf eine volle vertikale Säule. Solche Modelle werden verwendet, um physikalische Prozesse zu testen, die keine horizontalen Flüsse enthalten.

4.10 Softwarequalität

Wie diese Studie zeigt, testen einige Teams ihren Code systematischer und automatisierter als andere, es wird jedoch allgemein anerkannt, dass systematisches und automatisches Testen nützlich ist. Während von den Teams mit weniger strengen Teststrategien Aussagen zu hören waren wie: „We could/should do better“, haben diejenigen, die mehr in Praktiken wie automatischen Builds oder obligatorischer Testsammlungen investiert haben, ausgesagt, dass sich diese Investitionen gelohnt haben. Zwei Aspekte scheinen systematisches und automatisches Testen zu fördern:

1. organisatorischer Druck, z.B. durch große Teams oder Open-Source-Entwicklung
2. Einzelpersonen, die entsprechende Praktiken vorantreiben.

Hingegen sagten einige EntwicklerInnen aus, dass Zeitdruck sie davon abhält, systematischer zu testen. Dieser entsteht beispielsweise durch die CMIP-Termine oder andere Fristen.

Auch das Problem fehlender Testorakel wird von manchen als Grund angegeben, nicht systematischer zu testen. So äußerte ein Entwickler in einem Teammeeting sinngemäß: „Man kann nicht etwas testen, für das keine korrekte Lösung bekannt ist.“ Dabei zeigt gerade diese Studie, welche vielfältigen Möglichkeiten zum Testen dennoch bestehen.

Insgesamt ließ sich aus den Aussagen der interviewten EntwicklerInnen jedoch kein grundlegendes Qualitätsproblem ableiten. In der ICON-Umfrage aus dem Jahr 2014 gab zudem der Großteil der Teilnehmenden an, dass Fehler im Hauptentwicklungszweig nur manchmal bis nie vorkommen (siehe Anhang A). Dabei ist zu bemerken, dass die Umfrage durchgeführt wurde, bevor das Gatekeepersystem in der ICON-Entwicklung etabliert wurde. Zu dieser Zeit hatte noch jede EntwicklerIn Zugriff auf den Hauptentwicklungszweig. Aus Sicht der meisten EntwicklerInnen erscheint die herrschende Praxis jedoch geeignet, einen Großteil der Fehler im Modell aufzudecken. Wobei jedoch gilt: je systematischer das Testen, desto größer das Vertrauen in die eigene Praxis.

Dies lässt sich auch folgendermaßen begründen: Anders als in der industriellen Softwareentwicklung, wo es eine klare Trennung zwischen EntwicklerInnen und BenutzerInnen sowie zwischen Test- und Produktiveinsatz gibt, lässt sich diese Trennung in der Klimamodellierung nicht vornehmen. Die verwendeten Tests bestehen zum großen Teil aus den Experimenten, die auch zu wissenschaftlichen Zwecken durchgeführt werden. Hinzu kommt die große Empfindlichkeit der Modelle gegenüber kleinen Änderungen sowie gegenüber Fehlerursachen (Programmierfehler oder fehlerhafte Daten). Fehlerursachen werden so in den meisten Fällen entweder sehr schnell sichtbar oder eben gar nicht, wobei sich beides positiv auf die (gefühlte) Softwarequalität auswirkt.

Ein verstärkter Einsatz von Unittests zusätzlich zu den verbreiten E2E-Tests könnte das Vertrauen in die Softwarequalität weiter erhöhen und sich zusätzlich positiv auf die Produktivität der Entwicklung auswirken. Eine vertiefte Diskussion dieses Ansatzes folgt in Kapitel 5.

4.11 Einschränkungen

Die Studie basiert auf semistrukturierten Interviews, in denen die interviewten EntwicklerInnen ihre eigenen Praktiken beschrieben haben. Es ist nicht auszuschließen, dass in einigen Fällen die Interviewten den tatsächlichen Stand der Praxis mit dem angestrebten Stand durcheinander gebracht haben.

Es war nicht möglich, dieselbe Anzahl von Interviews mit allen Teams zu führen. Die Studie war umfangreicher für ICON und GFDL als für MIROC, NICAM und

SCALE. Eine ausgeglichene Studie hätte eventuell zu einer besseren Vergleichbarkeit der Beobachtungen geführt.

In alle Teams war die Teilnahme an den Interviews freiwillig. Es ist davon auszugehen, dass gerade diejenigen teilgenommen haben, die ein allgemeines Interesse an den diskutierten Themen haben. Darüber hinaus waren viele der Interviews Gruppeninterviews, in denen einige EntwicklerInnen häufiger zu Wort kamen als andere. Daher ist nicht auszuschließen, dass die Antworten der allgemein Softwaretechnik- und Testinteressierten EntwicklerInnen die Ergebnisse der Studie dominieren.

Es kann keine Allgemeingültigkeit der Beobachtungen dieser Studie beansprucht werden. Es gibt weltweit viele weitere Modellierungsgruppen, welche ihre eigenen Entwicklungs- und Testpraktiken pflegen. Es kann jedoch davon ausgegangen werden, dass einige der festgestellten Muster, wenn auch nicht in allen, so doch in einigen vergleichbaren Umgebungen anzutreffen sind (siehe auch Abschnitt 2.2.3).

4.12 Verwandte Arbeiten

Steve Easterbrook und Timothy C. Johns (2009) haben die Modellentwicklung am *Hadley Centre for Climate Prediction and Research* des britischen *MetOffice* untersucht. Einige Absätze in dem Artikel sind auch der Testpraxis gewidmet. Ihre Erkenntnisse ähneln sehr denen dieser Studie. Dies schließt unter anderem ein:

- die hohe Bedeutung des Regressionstestens
- die häufige Verwendung von E2E-Tests
- der Vergleich von Ergebnisdaten mit Referenzdaten, entweder automatisch durch bitweisen Vergleich oder manuell mit Hilfe von Diagrammen

Von Unittests ist bei Easterbrook und Johns nicht die Rede. Es ist davon auszugehen, dass diese daher auch nicht vorgefunden wurden.

Thomas L. Clune und Richard B. Rood (2011) berichten über ihre eigenen Erfahrung in der Modellentwicklung, u.a. am *Goddard Institute for Space Studies* der *NASA*. Auch diese Autoren bestätigen die Nichtverwendung von Unittests:

> *„The community invests heavily in system-level testing, which exercises most portions of the underlying code base with realistic inputs. These tests verify that multiple configurations of the model run stably for modest time intervals and for certain key invariants such as parallel reproducibility and checkpoint restart.*

> *On the other hand, the routine application of fine-grained, low-level tests (unit tests) is nearly nonexistent. Notable exceptions are limited to infrastructure software (such as that of the Earth System Modeling Framework).“*

Dementgegen berichten Drake, Jones und Carr (2005) von systematischem Unittesten in der Entwicklung des *Community Climate System Model (CCSM)* am NCAR.

Aussagen zur Softwarequalität von Klimamodellen finden sich in einer Arbeit von Jon Pipitone und Steve Easterbrook (2012). In dieser Studie haben die Autoren anhand von Einträgen in Bug-Tracking- und Versionskontrollsystemen die Fehlerdichte von drei GCMs berechnet und diese mit der Fehlerdichte von bekannten Open-Source-Softwareprojekten aus anderen Veröffentlichungen verglichen. Danach haben die GCMs eine deutlich geringere Fehlerdichte als die Vergleichsprojekte.

4.13 Zusammenfassung

Klimamodelle werden umfangreich getestet. In diesem Kapitel wurde eine bisher einzigartige Studie dieser Praxis an vier Forschungsinstituten beschrieben. Sie hebt sich von vergleichbaren Studien insbesondere durch den Fokus auf das Thema und eine damit verbundene detaillierte Beschreibung der entsprechenden Praktiken ab sowie durch die Untersuchung mehrerer Institute.

Die Studie basiert auf einer Langzeituntersuchung der Fallstudie ICON sowie auf semistrukturierten Interviews mit Entwicklerinnen und Entwicklern am GFDL, bei JAMSTEC und dem RIKEN CCS. Als Defizit aus softwaretechnischer Sicht konnte dabei identifiziert werden, dass Unittests in der Entwicklung von Klimamodellen nur sehr selten zum Einsatz kommen. Dies gilt insbesondere für die wissenschaftlich-fachlichen Teile des Modellcodes, obwohl die wenigen Unittests die hierfür existieren als nützlich angesehen werden. Verschiedene Gründe hierfür wurden diskutiert. Am schwersten scheint zu wiegen, dass der Aufwand für die Erstellung derartiger Tests als zu hoch eingeschätzt wird und dass die stattdessen verwendeten E2E-Tests als adäquat angesehen werden, den größten Teil der Fehler im Modell aufzudecken. Die Nachteile dieser Praxis sowie eine weitergehende Analyse der Gründe werden im nachfolgenden Kapitel 5 diskutiert.

Eine weitere wichtige Erkenntnis der Studie ist, dass das Regressionstesten eine zentrale Rolle im Entwicklungsprozess aller Teams einnimmt. Dies dient zum einen dazu, die Reproduzierbarkeit vergangener Ergebnisse zu gewährleisten und zum anderen, die verteilte Entwicklung zu koordinieren, indem die Änderungen einzelner EntwicklerInnen überprüft werden bevor sie von allen anderen übernommen werden. Für

das Regressionstesten mit E2E-Tests ist es häufig ausreichend nur kurze Zeiträume (wenige Zeitschritte bis zu einem Monat) zu simulieren.

Ergebnisse werden auf zwei verschiedene Art und Weisen überprüft: Zum einen manuell, in dem die EntwicklerInnen die Diagramme betrachten, die im Rahmen des Postprocessing erstellt werden, und ggf. mit Diagrammen von Referenzdaten vergleichen. Hierbei kommen auch spezielle Werkzeuge zum Einsatz, um diese Art der Überprüfung zu unterstützen. Sie wird von allen untersuchten Teams angewandt. Zum anderen automatisch durch den bitweisen Vergleich mit Referenzdaten. Diese stammen entweder aus Kontrollsimulationen mit vorherigen Versionen oder von Simulationen mit derselben Modellversion unter anderen technischen Randbedingungen, um sicherzustellen, dass diese keinen Einfluss auf die Modellergebnisse haben.

Kapitel 5

Unittests als Ergänzung zur bestehenden Testpraxis

Die Testpraxis in der Klimamodellierung basiert im Wesentlichen auf zwei Säulen: wissenschaftliche Evaluation und Regressionstests. Beide Teststrategien basieren zum größten Teil auf *Ende-zu-Ende-Tests (E2E-Tests)*, d.h. aus der Ausführung des Modells in einer bestimmten Konfiguration. Für die wissenschaftliche Evaluation ist dies in der Regel auch die einzige Möglichkeit. Um die wissenschaftliche Nützlichkeit einer Simulation beurteilen zu können, kommt man nicht umhin, diese über einen gewissen Zeitraum durchzuführen und deren Ergebnisse zu bewerten. Lediglich bei der Entwicklung einzelner, einfacherer physikalischer Prozesse, wie etwa der Strahlung oder in der Landmodellierung sind ggf. einfachere Verfahren, wie z.B. Ein-Säulen-Tests möglich.

Aber auch für Regressionstests, mit denen Fehler aufgedeckt werden sollen, die durch Änderungen am Modell entstehen, werden mit wenigen Ausnahmen E2E-Tests eingesetzt. Dies gilt auch, wenn lediglich geprüft werden soll, dass zwei Modellversionen oder -konfigurationen dieselben Ergebnisse liefern. In der Regel kann und sollte eine EntwicklerIn nur an einer Prozedur eines Modells gleichzeitig arbeiten und ihre Änderungen anschließend testen. Hierfür böten sich Unittests einzelner Prozeduren an. Doch solche werden, wie dargestellt, in der Klimamodellierung selten eingesetzt, d.h. selbst wenn nur Änderungen an einer einzelnen Prozedur getestet werden sollen, geschieht dies i.d.R. mit einem E2E-Test.

In diesem Kapitel werden die Nachteile (Abschnitt 5.1) und Gründe (Abschnitt 5.2) dieses Vorgehens diskutiert. Aus dieser Diskussion wird anschließend in Abschnitt 5.3 das Ziel dieser Arbeit abgeleitet.

5.1 Nachteile von E2E-Tests

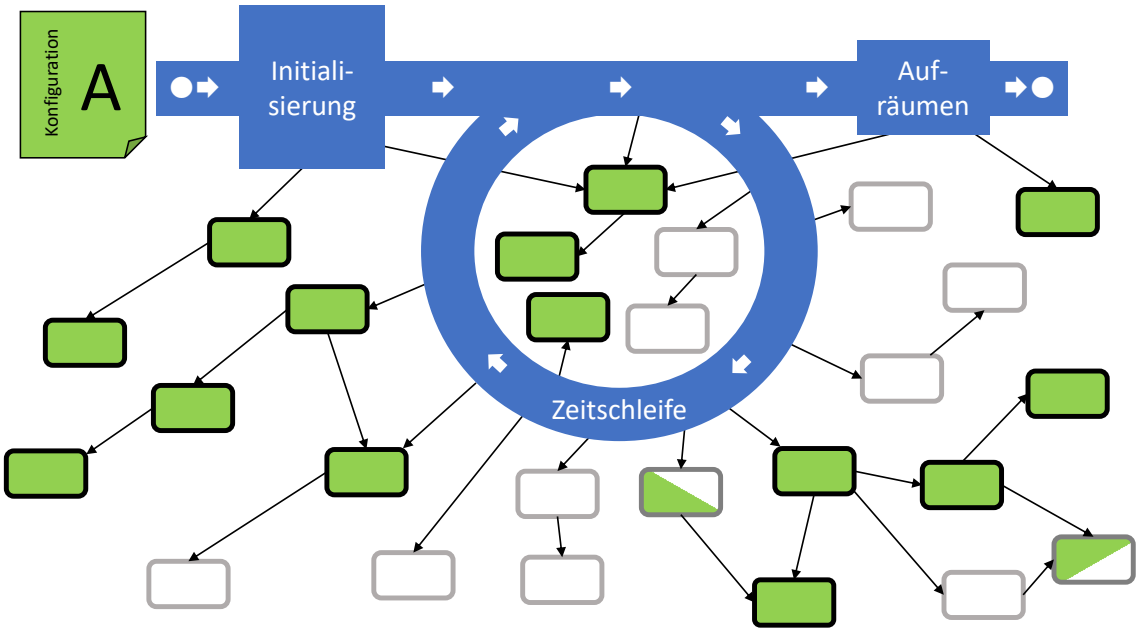
Für die Verwendung von E2E-Tests als alleinige Regressionstests gibt es naheliegende Gründe. Um etwa eine Prozedur zu testen, muss diese ausgeführt werden. Hierfür müssen einige Voraussetzungen gegeben sein. So benötigt man neben der Prozedur selbst zum einen die Module und Prozeduren, die von der zu testenden Prozedur selbst direkt und indirekt verwendet werden, und zum anderen ein Treiberprogramm, in dem die zu testende Prozedur aufgerufen wird sowie sinnvolle Eingabedaten. All dies ist vorhanden, wenn man eine bestehende Konfiguration des Modells verwendet, durch die die zu testenden Prozedur ausgeführt wird.

Hinzu kommt, dass Klimasimulationen ohne Benutzerinteraktionen und somit weitestgehend automatisiert ablaufen. Die EntwicklerIn muss lediglich eine Konfigurationsdatei erstellen oder von einem anderen Experiment übernehmen und mit diesem kann der Test dann wiederholt ausgeführt werden. Lediglich die Ergebnisse müssen ggf. begutachtet werden. Werden jedoch bitidentische Ergebnisse zu einer Referenzversion erwartet, kann der Vergleich der Daten ebenfalls automatisiert werden. So erhält man relativ leicht einen vollständig automatisierten Test inklusive automatischem Testorakel. Die Erstellung separater Testprogramme für einzelne Prozeduren ist dagegen mit zusätzlichem Aufwand verbunden.

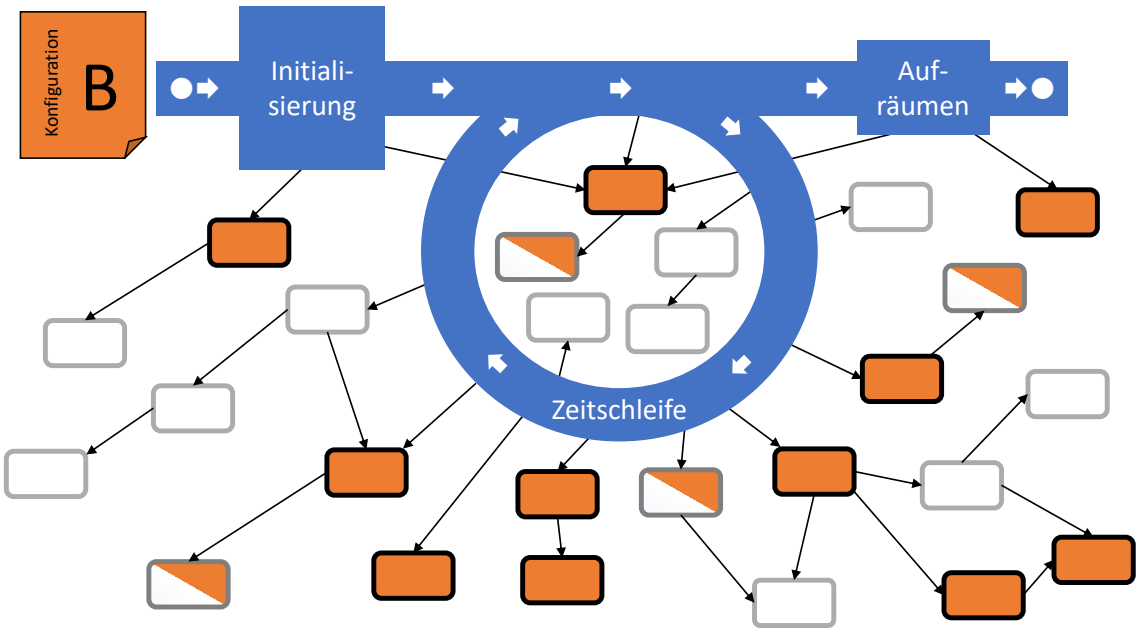
Dennoch ist die alleinige Verwendung von E2E-Tests und der Verzicht auf Unittests mit Nachteilen verbunden, insbesondere wenn es darum geht Änderungen an einer einzelnen Prozedur zu testen. Diese werden im Folgenden diskutiert.

5.1.1 Laufzeit

E2E-Tests dauern prinzipbedingt länger als Unittests einzelner Prozeduren. Abbildung 5.1 zeigt schematisch die Abläufe von zwei Simulationen, wie sie auch bei E2E-Tests stattfinden. Zusätzlich sind die Prozeduren, aus denen das Modell besteht, dargestellt. In allen Phasen einer Simulation werden je nach Modellkonfiguration verschiedene Prozeduren ausgeführt. Sollen Änderungen an einer Prozedur getestet werden, muss eine passende Modellkonfiguration gefunden werden, bei der die zu testende Prozedur ausgeführt wird. Zwar lassen sich von einem Klimamodell in vielen Fällen sehr reduzierte Konfigurationen erstellen, indem die simulierte Zeit auf einen oder wenige Zeitschritte begrenzt wird, die Auflösung reduziert und so viele unnötige physikalische Prozesse wie möglich deaktiviert werden, dennoch lässt es sich nicht vermeiden, dass neben der zu testenden Prozedur viele weitere Prozeduren ausgeführt werden. Dies verlängert die Laufzeit des Tests in unnötiger Weise. Insbesondere die Initialisierungsphase dauert immer eine gewisse Zeit und es lässt sich



(a) Konfiguration A



(b) Konfiguration B

Abbildung 5.1: Schematische Darstellung der ausgeführten Prozeduren während zweier Simulationen mit unterschiedlichen Konfigurationen.
Farbig ausgefüllte Kästen: Ausgeführte Prozeduren
Halb gefüllte Kästen: Teilweise ausgeführte Prozeduren
Leere Kästen: Nicht ausgeführte Prozeduren

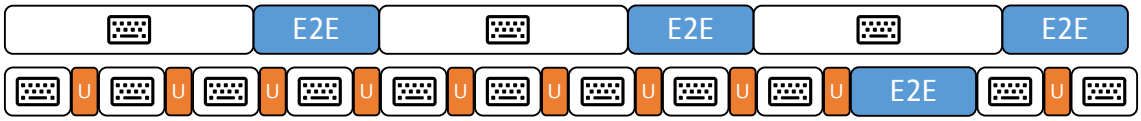


Abbildung 5.2: Programmierphasen und Feebackereignisse ohne (oben) und mit Unittests (unten); ⌨ = Programmierphase, E2E = E2E-Test, U = Unittest

nicht vermeiden, dass darin auch Datenstrukturen angelegt werden, die für die zu testende Prozedur selbst gar nicht benötigt werden.

Längere Laufzeiten bedeuten wiederum, dass Tests seltener ausgeführt werden und sich dadurch die Feedbackzyklen während des Programmierens verlängern. Dadurch werden Fehler später gefunden, was wiederum den Aufwand der Beseitigung ihrer Ursachen erhöhen kann. Abbildung 5.2 veranschaulicht diesen Umstand: Dargestellt sind die Phasen des Programmierens und Feedbackereignisse in Form von E2E- bzw. Unittests.

Hinzu kommt, dass das Verhalten einer Prozedur häufig von der Modellkonfiguration oder der Ausführungsumgebung (Rechnerplattform, vorhandene Bibliotheken etc.) abhängig ist. Je nach Konfiguration wird nur ein Teil des Codes einer Prozedur ausgeführt (siehe auch die halb gefüllten Kästen in Abbildung 5.1). Fallunterscheidungen wie die folgende sind keine Seltenheit:

Beispiel 5.1: Konfigurationsabhängige Fallunterscheidung

```
1 IF (FEATURE_XY_ENABLED) THEN
2   :
3 ELSE
4   :
5 END IF
```

Hieraus ergeben sich unterschiedliche Ausführungspfade, die separat getestet werden müssen. Längere Ausführungszeiten pro Test führen jedoch dazu, dass ggf. weniger Konfigurationsoptionen durchgetestet werden können.

Zudem lassen sich einige E2E-Tests nur auf einem Hochleistungsrechner ausführen. Die zur Verfügung stehende Rechenzeit ist dort jedoch meist begrenzt und jede Testausführung belastet das eigene Rechenzeitkonto. Längere Testlaufzeiten bedeuten somit nicht nur eine längere Wartezeit für den Entwickler, sondern auch eine höhere Belastung des Rechenzeitkontos.

Es ist schwierig typische Laufzeiten für E2E-Tests zu benennen, da diese stark vom jeweiligen Modell, dem gewählten Experiment bzw. der Konfiguration, der Rechner-

plattform und Compilereinstellung abhängen. Beispielhaft sei einer beliebig ausgewählter erfolgreicher Build des ICON-Buildbots[9] betrachtet: Darin enthalten sind 25 E2E-Tests mit Laufzeiten von 20 Sekunden bis zu 25 Minuten. Den Spitzenwert von rund 25 Minuten erreicht jedoch nur ein Test, der zweitlängste dauert 6:40 Minuten, während der zweitschnellste in 55 Sekunden abgearbeitet wurde. Die durchschnittliche Laufzeit beträgt 4:05 Minuten, der Median 2:49 Minuten. Der Build wurde auf der *Mistral*, dem Hochleistungsrechner des DKRZ ausgeführt (siehe auch DKRZ, 2016).

5.1.2 Übersetzungszeit

E2E-Tests benötigen gegenüber Unittests nicht nur mehr Zeit für die Ausführung, sondern auch für die Kompilierung. Klimamodelle bestehen im Wesentlichen aus einer Menge von Fortran-Modulen. Gibt es eine Änderung an einem dieser Fortran-Module, z.B. in dem Modul, welches die zu testende Prozedur beinhaltet, müssen bei der Neuübersetzung sämtliche Module, welche von dem geänderten Modul abhängen, erneut kompiliert werden. Am Ende des Übersetzungsprozesses steht also immer die Kompilierung des Hauptprogramms des Klimamodells. Wird nach Änderung der zu testenden Prozedur anstelle des Modells nur ein Unittest neu übersetzt, ist das einzige Modul, welches kompiliert werden muss, der Testtreiber. Abbildung 5.3 verdeutlicht diesen Unterschied im Übersetzungsaufwand.

Auch für den Übersetzungsaufwand lassen sich aus denselben Gründen wie bei der Laufzeit nur schwer typische Zahlen angeben. Als Beispiel sei auch hier der o.g. Build des ICON-Buildbots genannt. Die vollständige Kompilierung des Modells hat hier 14:29 Minuten gedauert.

5.1.3 Ergebnisanalyse

Klimamodelle sollen sich möglichst deterministisch verhalten, d.h. bei gleicher Konfiguration, gleichen Eingabedaten und gleichen technischen Randbedingungen sollten sie stets dieselben Ergebnisse liefern. Dies stellt die Reproduzierbarkeit von Experimenten sicher. Nichtdeterminismus erschwert zudem das Testen, da etwa ein bitweiser Vergleich mit Referenzdaten nicht möglich ist. Dennoch gibt es Modelle, die sich zumindest in einigen Konfigurationen nichtdeterministisch verhalten. Der Grund dafür sind häufig Optimierungen von parallelen Algorithmen. Quellen des Nichtdeterminismus sind dabei oft lokal begrenzt, d.h. nur einige Prozeduren verhalten sich nichtdeterministisch, während die Mehrzahl der Prozeduren deterministisch arbeitet.

[9] Build #3610, Builder: Mistral_gcc, Branch: icon-nwp/icon-nwp-dev, Commit: afec18d0

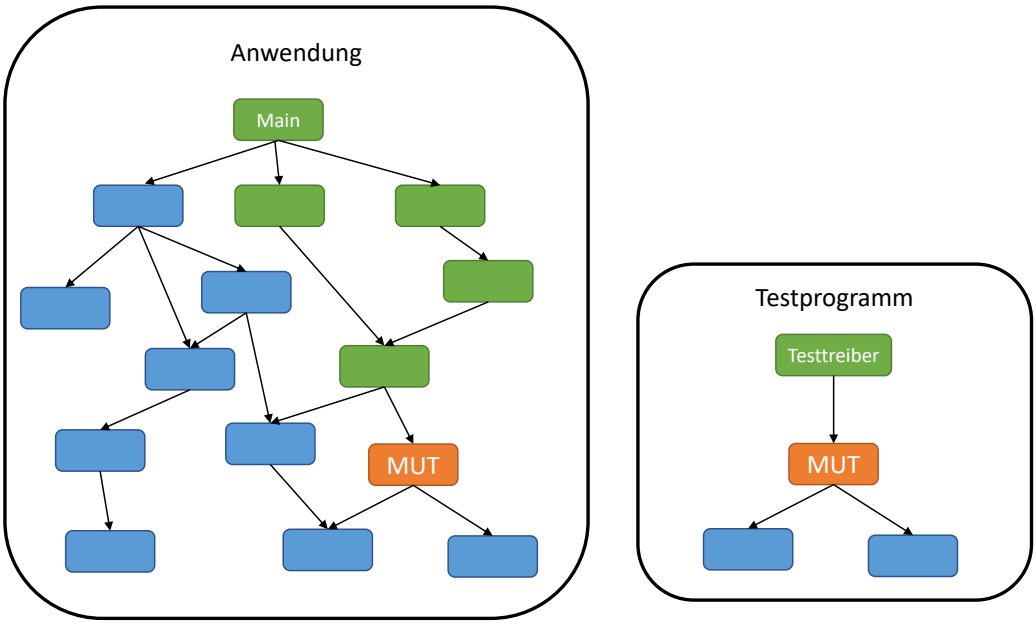


Abbildung 5.3: Neuübersetzung von Hauptanwendung und Unittest
MUT: zu testendes Modul (*module under test*)
Grün markiert: Module mit Abhängigkeiten zur MUT, welche bei Änderungen am MUT ebenfalls neu kompiliert werden müssen

Werden aber nun in einem E2E-Test neben einer zu testenden deterministischen Prozedur auch nichtdeterministische aufgerufen, erschwert dies unnötig die Analyse der Testergebnisse der deterministischen Prozedur.

Gleiches gilt auch für Prozeduren, deren Ergebnis unabhängig von der Anzahl der parallelen Prozesse oder anderer technischer Randbedingungen ist. Die Erhaltung dieser Unabhängigkeit innerhalb einer Prozedur lässt sich mit Hilfe von E2E-Tests, deren Endergebnisse hiervon nicht unabhängig sind, nur schwerlich testen.

5.1.4 Reproduzierbarkeit

Nichtdeterminismus erschwert nicht nur die Analyse von Testergebnissen, sondern verhindert auch die Reproduzierbarkeit einzelner Ausführungen. Auch wenn eine Prozedur selbst deterministisch arbeitet, innerhalb eines E2E-Tests jedoch mit zufallsbehafteten Eingabedaten aufgerufen wird, lässt sich eine konkrete Ausführung mit dem E2E-Test unter Umständen nicht wiederholen. Innerhalb von Unittests kann dagegen dafür Sorge getragen werden, dass eine zu testende Prozedur immer mit denselben Eingabedaten aufgerufen wird. Dies erleichtert die Bewertung des Tests.

5.1.5 Fehlerursachenlokalisierung

Isolierte Tests einzelner Prozeduren erleichtern die Lokalisierung von Fehlerursachen, da jeder Fehler, der bei so einem Test auftritt, sich nur in der zu testenden Prozedur oder in einer von ihr verwendeten Prozedur verursacht werden kann. Auch bei der Verwendung von E2E-Tests als Regressionstests lassen sich Fehlerquellen mit großer Wahrscheinlichkeit auf die Quelltextstellen eingrenzen, die seit dem letzten erfolgreichen Test geändert wurden. Die längeren Übersetzungs- und Ausführungszeiten von E2E-Tests können jedoch dazu führen, dass es mehr Änderungen und somit mehr potenzielle Fehlerquellen zwischen der Ausführung zweier Tests gibt. Zudem können kleinere Fehler im Rahmen eines E2E-Tests weitreichende Auswirkungen haben, so können z.B. fehlerhafte Daten in einer Variable zu fehlerhaften Daten in weiteren Variablen führen. In einigen Fällen lassen sich daher auch bei Kenntnis der jüngsten Änderungen Fehlerursachen nur schwer auffinden.

Erschwert wird dies zusätzlich, wenn Fehlern erst im Rahmen der Qualitätssicherung zu Tage treten, z.B. während eines automatisierten Builds eines CI-Systems. Hierbei werden häufig sehr viele Änderungen mehrerer EntwicklerInnen gemeinsam getestet, wodurch auch die Menge möglicher Fehlerursachen vergrößert wird. Hier können Unittests ebenfalls zur einfacheren Lokalisierung von Fehlerursachen beitragen, indem sie diese auf die Prozeduren eingrenzen, von denen die zugehörigen Unittests fehlgeschlagen ist.

5.1.6 Pfadabdeckung

Die Eingabedaten einer zu testenden Prozedur lassen sich bei E2E-Tests nicht direkt kontrollieren. Es können somit nur die Zustände getestet werden, die im Rahmen des jeweiligen E2E-Tests auftreten. In vielen Fällen lässt sich so keine hinreichende Pfadabdeckung erreichen. Auch Fehler und Randfälle lassen sich so deutlich schwieriger reproduzieren.

5.1.7 Monolithische Architekturen

Aufgrund der Arbeitsteilung innerhalb der Entwicklungsteams bestehen Klimamodelle aus einer Vielzahl spezialisierter Module. Das bedeutet jedoch nicht, dass das Geheimnisprinzip, nachdem einzelne Designentscheidungen möglichst innerhalb eines Moduls gekapselt sein sollten (Parnas, 1972), auch immer eingehalten ist. So finden sich häufig Annahmen über die Funktionsweise eines Moduls in einem anderen wieder, auch die Verwendung globaler Variablen ist nicht unüblich. Dies erschwert den Austausch und die Wiederverwendung einzelner Module.

Da sich unabhängige Module einfacher unabhängig voneinander testen lassen, geht man davon aus, dass die Verwendung von Unittests den Aufbau einer modularen Architektur unterstützt (vgl. Klammer und Kern, 2015), während monolithische Tests gleichfalls zu monolithischeren Architekturen führen können. Auch sind Unittests ein wichtiges Mittel, wenn es darum geht, bestehende Softwarearchitekturen umzugestalten und zu verbessern (vgl. Fowler, 1999, S. 89.ff; Feathers, 2004, S.9ff).

5.1.8 Notwendiges Fach- und Architekturwissen

Um das Verhalten einer einzelnen Routine im Kontext eines gesamten Klimamodells verstehen und beurteilen zu können sind ggf. sowohl ein hohes Fachwissen über die physikalischen Wechselwirkungen innerhalb des Modells als auch ein hohes Architekturwissen, über das Zusammenspiel der einzelnen Module notwendig. Dies ist vor allem dann problematisch, wenn neue oder externe EntwicklerInnen mit Änderungen an einer Routine betraut werden. Hier wäre es von Vorteil, wenn man diesen zunächst kleinere, fokussierte Tests der betreffenden Routine mit vorhersehbarem Verhalten an die Hand geben könnte.

Durch neue technische Herausforderung, wie die immer größere Zahl paralleler Rechnerknoten in modernen Hochleistungsrechner oder vermehrt heterogene Rechnerarchitekturen, sind immer häufiger fachfremde IT-ExpertInnen in die Entwicklung von Klimamodellen eingebunden (siehe auch Lawrence, Rezny u. a., 2018). Deren Aufgaben bestehen oft darin, nichtfunktionalen Änderungen wie Optimierungen, Portierungen oder Refactorings umzusetzen. Gerade für solche Arbeiten wären fokussierte Regressionstests, welche sich auch mit weniger Fach- und Architekturwissen durchführen lassen, nützlich.

5.2 Gründe für die Abwesenheit von Unittests

Die Nützlichkeit von Unittests wird auch von EntwicklerInnen von Klimamodellen grundsätzlich anerkannt. So werden Unittests teilweise bei der Entwicklung von Infrastrukturbibliotheken eingesetzt. Auch die sehr einfachen fachlichen Unittests des SCALE-Teams werden als sehr hilfreich beschrieben. Dennoch ist die Verwendung derartiger Tests sehr gering. Die Gründe hierfür wurden bereits in Abschnitt 4.9 diskutiert. Im Folgenden sollen diese noch einmal zusammengefasst und analysiert werden.

Im Wesentlichen lassen sich die Gründe in drei Kategorien einteilen:

- menschenbezogene Gründe
- das Testorakelproblem
- der Aufwand der Erstellung

Zu den menschenbezogenen Gründen gehören u.a. Hindernisse, die mit fehlender Ausbildung zu tun haben, wie etwa die Unkenntnis über die Vorteile von Unittests oder die fehlende Erfahrung in der Erstellung von Unittests. Zum anderen fallen hierunter psychologische Gründe, wie etwa der Hang an gewohnten Vorgehensweisen festzuhalten.

Das Testorakelproblem wurde bereits ausführlich in dieser Arbeit diskutiert. Wenn die Ergebnisse, die eine Prozedur liefern soll, unbekannt sind, lassen sich auch in einem Unittest keine entsprechenden Prüfungen einbauen. Fehler lassen sich jedoch auch mit anderen Methoden finden. So könnten beispielsweise, auch wenn die genaue Lösung einer Gleichung unbekannt ist, zumindest bekannte Eigenschaften getestet werden, etwa ob das Ergebnis über oder unter einem bestimmten Wert liegt, ob die Lösung konvergiert etc. oder auch dass bestimmte Variablen die Lösung nicht beeinflussen. Beim Regressionstesten existiert i.d.R. ohnehin ein Testorakel, nämlich das Verhalten der Vorversion der zu testenden Prozedur, sofern dieses bereits erfolgreich getestet wurde, z.B. mit Hilfe eines E2E-Tests. Ähnliches gilt für das Debugging, hier besteht das Testorakel in der Prüfung, ob ein bekannter Fehler noch auftritt oder eben nicht.

Am schwersten scheint dagegen der Aufwand zu wiegen, der mit der Erstellung von Unittests verbunden ist. Der Produktivitätsgewinn, der sich aus der Verwendung von Unittests ergibt, hängt in besonderem Maße von dem Aufwand ab, der mit der Testerstellung verbunden ist. Aufgrund der fachlichen und softwaretechnischen Strukturen von Klimamodellen ist dieser in vielen Fällen jedoch so hoch, dass sich eine rein manuelle Erstellung eines solchen Tests kaum lohnt. Ein einzelner Unittest besteht mindestens aus den folgenden Komponenten:

- Eingabedaten für die zu testende Prozedur
- Aufruf der zu testenden Prozedur
- Testtreiber: Programm, das beide Teile miteinander verbindet: das Erzeugen der Eingabedaten sowie den Aufruf der Prozedur

Dies kann Teil des Testprogramms sein oder nachträglich ausgeführt werden und es kann sehr einfach oder beliebig komplex ausfallen. Das implizite Orakel besteht lediglich in der Feststellung, ob die zu testende Prozedur terminiert. Dies lässt sich

einfach durch die Ausführung des Testtreibers feststellen, ohne dass zusätzlicher Code notwendig ist.

Der Aufruf der zu testenden Prozedur besteht in der Regel aus einer einzelnen Zeile Code. Der größte Aufwand besteht dagegen meist in der Erzeugung von geeigneten Eingabedaten. In Klimamodellen werden sehr große Datenmengen verarbeitet. Dies bedeutet zum einen, dass sowohl die Anzahl an Variablen, die von einer Prozedur zur nächsten weitergereicht werden, sehr groß ist, als auch dass die einzelnen Variablen, welche in den meisten Fällen Arrays sind, sehr groß sind. Um einen vollständigen Satz an Eingabedaten für eine Prozedur zu erstellen, müssen also ggf. sehr viele Datenpunkte mit Werten belegt werden. Erschwerend kommt dabei hinzu, dass das Layout dieser Arrays nicht immer vorhersehbar ist (siehe auch Abschnitt 2.3.3). Die entsprechenden Datenstrukturen werden innerhalb der Initialisierungsphase eines Modells angelegt. Eine manuelle Erzeugung ist dagegen kaum möglich. Ein weiteres Problem besteht darin, überhaupt zu wissen, welche Eingabedaten eine Prozedur benötigt. Dies ist nicht immer an deren Schnittstelle ersichtlich, beispielsweise wenn globale Variablen verwendet werden, was in Klimamodellen ausgiebig getan wird.

Die genannten Gründe können nicht isoliert voneinander betrachtet werden. So ist die Erzeugung von automatischen Testorakeln ebenfalls mit Aufwand verbunden. Der Aufwand selbst hat nicht nur Auswirkungen auf die rationale Abwägung, ob sich die Erstellung von Unittests lohnt, sondern auch psychologische Auswirkungen. Auf der anderen Seite hat wiederum die Erfahrung der EntwicklerInnen mit Erstellung von Unittests Einfluss auf den Aufwand.

5.3 Ziel dieser Arbeit

Vorangehend wurden die Nachteile, die mit dem alleinigen Einsatz von E2E-Tests in der Klimamodellierung verbunden sind, diskutiert. Unittests könnten hier eine sinnvolle Ergänzung darstellen. Für deren Abwesenheit gibt es jedoch nachvollziehbare Gründe. Um unter den gegebenen Bedingungen dennoch verstärkt Unittests einsetzen zu können, bedarf es einer Form der Unterstützung. Ziel dieser Arbeit soll es daher sein, eine Methode zur Unterstützung der Erstellung von Unittests einzelner Prozeduren zu entwerfen. Diese soll in Form eines Softwarewerkzeugs umgesetzt und erprobt werden.

Im Mittelpunkt steht dabei die Verringerung des Aufwands für die Erstellung solcher Tests. Fragen der Ausbildung sowie psychologische und soziologische Aspekte können und sollen kein Gegenstand dieser softwaretechnisch ausgerichteten Arbeit sein. Jedoch wird von der Annahme ausgegangen, dass sich durch die Verringerung des Aufwands die Motivation zur Erstellung von Unittests automatisch erhöht, wenn

dadurch eine Produktivitätssteigerung in den Augen der EntwicklerInnen wahrscheinlicher wird. Zudem wird ein anwendungsorientierter Ansatz verfolgt, d.h. dass sich die Funktionalität des Softwarewerkzeugs an den Aufgaben des Anwendungsbereichs orientiert, die Handhabung des Werkzeugs benutzergerecht sein soll, und sich die im System festgelegten Abläufe und Schritte je nach Anwendungssituation an die tatsächlichen Anforderungen anpassen lassen (siehe auch Definition 1.1). Die anvisierten BenutzerInnen des Werkzeugs sind EntwicklerInnen von Klimamodellen. Eine benutzergerechte Handhabung schließt auch deren typischen Kenntnisse in Softwareentwicklung und -testen mit ein, sowie deren gewohnte Arbeitsweisen. Im nachfolgenden Kapitel 6 werden die sich daraus ergebenden Anforderungen diskutiert.

Neben der wissenschaftlichen Evaluation bilden Regressionstests die wichtigste Säule des Testens in der Klimamodellierung. Auch für Unittests wäre dies der wichtigste Einsatzzweck. Daher wird zunächst von der Annahme ausgegangen, dass bestehende Softwareversionen als Testorakel dienen können. Auch wenn diese Annahme nicht in allen denkbaren Einsatzszenarien zutreffend ist, dürfte sie doch in einer Vielzahl der Fälle zu brauchbaren Tests führen. Diese Vereinfachung dürfte zudem weniger testerfahrenen EntwicklerInnen entgegenkommen, da das dahinterstehende Prinzip des Vorher-Nachher-Vergleichs leicht verständlich ist und vielfach auch im Rahmen von E2E-Tests zur Anwendung kommt. Doch auch unter dieser Annahme ist der mit Erstellung verbundene Aufwand von entscheidender Bedeutung, wenn Unittests eine Alternative zu bereits bestehenden E2E-Tests darstellen sollen.

Um eine Verringerung des Aufwands zu erreichen, müssen einzelne Schritte im Prozess der Testerstellung automatisiert werden. Wie dargelegt wurde, entfällt der größte Aufwand bei der Erstellung von Unittests auf die Erzeugung geeigneter Eingabedaten. Hier ließe sich durch Automatisierung die meiste Arbeit einsparen. Auch bei der Erstellung automatischer Testorakel, welche über das implizite Orakel hinausgehen, könnte durch Automatisierung der Arbeitsaufwand verringert werden. Welche Prozessschritte sich für eine Automatisierung eignen und wie diese anwendungsorientiert gestaltet werden kann, wird Gegenstand der folgenden Kapitel sein.

5.4 Zusammenfassung

In diesem Kapitel wurden die Nachteile, die mit der alleinigen Verwendung von E2E-Tests und dem Verzicht auf Unittests beim Regressionstesten von Klimamodellen verbunden sind, diskutiert. Hierzu gehören u.a. eine längere Lauf- und Übersetzungszeit von E2E-Tests im Vergleich zu Unittests, eine erschwerte Ergebnisanalyse und Lokalisierung von Fehlerursachen, eine erhöhte Schwierigkeit eine hohe Pfadabdeckung zu erreichen, eine Förderung monolithischer Softwarearchitekturen sowie ein höheres benötigtes Fach- und Architekturwissen.

Darüber hinaus wurden die bereits im vorherigen Kapitel untersuchten Gründe für die Abwesenheit von Unittests analysiert und darauf aufbauend das weitere Ziel dieser Arbeit formuliert. Dieses besteht darin, eine Unterstützung für die Erstellung von Unittests zu entwerfen und in Form eines Softwarewerkzeugs umzusetzen und zu erproben. Hierzu müssen einzelne Schritte des Erstellungsprozesses automatisiert werden. Bei der Gestaltung des Werkzeugs und der zugrundeliegenden Methode sollen anwendungsorientierte Aspekte berücksichtigt werden.

Kapitel 6

Anforderungen an eine Softwarelösung

Ziel dieser Arbeit ist es, eine anwendungsorientierte Softwarelösung zu entwickeln, um die Erstellung von Unittests für einzelne Prozeduren in Klimamodellen zu unterstützen. Durch Automatisierung einzelner Schritte soll dabei der Aufwand für die Erstellung solcher Tests reduziert werden. In diesem Kapitel werden Anforderungen an das zu entwickelnde Softwarewerkzeug formuliert.

Eine wesentliche Anforderung an eine anwendungsorientierte Lösung besteht darin, dass sie nicht nur an fiktiven Beispielprogrammen evaluiert wird, sondern mit echten Klimamodellen funktioniert. Der Anwendungsbereich Klimamodellierung wurde in den Kapiteln 2 und 4 ausführlich beschrieben. Wichtige Erkenntnisse dabei sind:

- Die EntwicklerInnen von Klimamodellen sind oft erfahrene ProgrammiererInnen ohne formelle Ausbildung in Informatik oder Softwareentwicklung.
- Klimamodelle sind langlebige Softwaresysteme, die evolutionär entwickelt werden.
- Klimamodelle sind im Wesentlichen in Fortran geschrieben.
- Klimamodelle sind hochgradig parallelisiert.
- Entwicklungs- und Produktionsumgebung sind von Rechenzentren betriebene Hochleistungsrechner.
- Bisher werden Klimamodelle im Wesentlichen mit Hilfe von Ende-zu-Ende-Tests getestet.

Aus diesen Erkenntnissen lassen sich weitere Anforderungen ableiten, die in diesem Kapitel beschrieben werden.

6.1 Nützliche Tests

Wichtigste Anforderung an die zu erstellenden Unittests ist, dass diese nützlich sind, d.h. sie müssen in der Lage sein, effektiv Fehler aufzudecken und bei Erfolg das Vertrauen in die Software zu stärken. Wie in Abschnitt 3.7 diskutiert, ist eine direkte Messung dieser Eigenschaft jedoch nur eingeschränkt möglich. Alternativ kann eine möglichst hohe Codeabdeckung angestrebt werden.

6.2 Geschwindigkeit

Unittests sollen vor allem dazu dienen, häufig während der Arbeit an einem Stück Programmcode ausgeführt zu werden, möglichst nach jeder kleineren Änderung, um unerwünschte Auswirkungen einer Änderung, so schnell es geht zu entdecken. Um das zu ermöglichen, müssen Unittests möglichst schnell ablaufen. Darüber, wie schnell schnell genug ist, gibt es unterschiedliche Meinungen (vgl. Fowler, 2014). Beispielsweise gilt laut Michael Feathers (2004, S. 13f) eine Laufzeit von einer Zehntelsekunde für einen einzelnen Unittest als langsam:

> *„Yes, I'm serious. At the time that I'm writing this, 1/10th of a second is an eon for a unit test. Let's do the math. If you have a project with 3,000 classes and there are about 10 tests apiece, that is 30,000 tests. How long will it take to run all of the tests for that project if they take 1/10th of a second apiece? Close to an hour. That is a long time to wait for feedback. You don't have 3,000 classes? Cut it in half. That is still a half an hour. On the other hand, what if the tests take 1/100th of a second apiece? Now we are talking about 5 to 10 minutes. When they take that long, I make sure that I use a subset to work with, but I don't mind running them all every couple of hours.“*

In den in dieser Arbeit betrachteten Systemen wären derartige Zeitskalen ohne umfangreiche manuelle Refactorings sicher unrealistisch. In der Regel dauert schon das Neuübersetzen des Quellcodes nach einer kleinen Änderung viel länger. Die Größenordnung der Anzahl von Codeeinheiten, aus denen Klimamodelle bestehen, ist grundsätzlich vergleichbar mit dem Beispiel von Feathers. Während Feathers von 3.000 Klassen spricht, besteht beispielsweise das ICON-Modell aus rund 880 Fortran-Modulen und enthält ca. 6.200 Prozeduren[10]. Ziel soll es jedoch nicht sein, auf einen Schlag Tests für jede einzelne Prozedur zu erzeugen. Vielmehr soll eine Lösung zum schnellen Erstellen von einzelnen Tests nach Bedarf gefunden werden, z.B. für Fälle,

[10] in Version 2.4.0 von Dezember 2018

in denen eine bestehende Prozedur optimiert oder refactored werden soll, oder zum Reproduzieren eines Fehlers.

Als Referenz soll daher die sonst übliche Alternative der E2E-Tests dienen. Ein Unittest sollte *signifikant schneller* sein als ein alternativer E2E-Test, in dem die zu testende Prozedur ebenfalls aufgerufen wird. Dabei stellt sich die Frage, was „signifikant“ in diesem Zusammenhang bedeutet. Mangels anderer Vorgaben aus der Literatur soll hier wie bei Feathers (1/100 gegenüber 1/10) der Faktor 10 das Maß sein, d.h. „signifikant schneller“ soll zehnmal so schnell bedeuten.

6.3 Benutzergerechte Handhabung

Anwendungsorientierte Software zeichnet eine benutzergerechte Handhabung und die Anpassbarkeit an die jeweilige Anwendungssituation aus. Diese Arbeit wird keine detaillierten Analysen zur Optimierung der *Benutzerfreundlichkeit (usability)* der angestrebten Lösung vornehmen, jedoch soll sich das grundsätzliche Benutzungsmodell an diesem Anspruch orientieren.

Tests können keine Korrektheit nachweisen, sollen jedoch das Vertrauen in die korrekte Funktion des zu testenden Codes erhöhen. Hierfür muss zunächst dem Test selbst vertraut werden. Grundlage dafür ist, dass seine Funktionsweise für die testende Personen selbst nachvollziehbar ist. Für Tests in der Klimamodellierung im Allgemeinen und für Unittests im Besonderen gilt, dass die testenden Personen die EntwicklerInnen der Klimamodelle selbst sind. Diese sind zwar oft erfahrene (Fortran-)Programmierer, verfügen jedoch i.d.R. über keine formale Ausbildung in Informatik oder Softwareentwicklung. Eine Lösung zur Unterstützung der Erstellung von Tests sollte dies berücksichtigen. Die Anwendung des Verfahrens sollte daher keine Fähigkeiten oder Kenntnisse voraussetzen, die über das hinausgehen, was in dieser Zielgruppe üblicherweise vorhanden ist. Berücksichtigt werden muss zudem, dass diese nicht homogen ist. So dürften langjährig festangestellte ProgrammiererInnen über mehr Erfahrung und technische Kenntnisse verfügen als DoktorandInnen, die ihr Forschungsprojekt gerade begonnen haben.

Eine vollautomatische Lösung, die auf Knopfdruck Tests erzeugt, ausführt und am Ende lediglich ausgibt, ob die Tests bestanden wurden oder nicht, würde u.U. zu nicht nachvollziehbarem Verhalten führen. Besser geeignet ist daher eine halbautomatische Lösung, bei der die aufwendigsten Arbeitsschritte durch Automatisierung beschleunigt werden, die EntwicklerInnen diese aber selbst steuern und die Ergebnisse manuell weiterverarbeiten können. Dies setzt voraus, dass zum einen die technischen

Mittel bereitstehen müssen, um die automatisierten Arbeitsschritte steuern und ihre Ergebnisse bearbeiten zu können, als auch dass Letztere für die EntwicklerInnen hinreichend les- und verstehbar sind.

Darüber hinaus sollte sich ein anwendungsorientiertes Softwarewerkzeug an die gewohnte Arbeitsweise der EntwicklerInnen anpassen und sich in deren vorhandenen Werkzeugkasten einfügen. Dieser besteht im Wesentlichen aus Linux/Unix-Kommandozeile, Texteditor, Compiler und einem Versionskontrollsystem, i.d.R. git. In einigen Entwicklerteams gehören auch CI-Server dazu. Daneben stehen die gängigen Linux/Unix-Werkzeuge zur Verfügung sowie Interpreter für Skriptsprachen, wie etwa Python, Ruby oder Perl.

Um eine EntwicklerIn überhaupt dazu zu bringen, einen Unittest für eine Prozedur zu erstellen, ist vor allem von Bedeutung den initialen Aufwand so gering wie möglich zu halten. Das bedeutet, dass möglichst schnell ein ausführbares Testprogramm vorliegen muss, welches dann Schritt für Schritt verfeinert und erweitert werden kann. Ein schnelles Feedback durch ein ausführbares Programm erhöht die Motivation zum Testen.

Eine allgemeine Lösung soll nicht nur mit einem konkreten Klimamodell, sondern mit möglichst vielen funktionieren. Dabei sollte sich die Software an das jeweilige Zielsystem anpassen lassen und nicht andersherum das Zielsystem an die Software. Die hierfür notwendigen Konfigurationen sollten jedoch nicht so aufwendig sein, dass sie eine Einsatzhürde für die EntwicklerInnen darstellen. Vielmehr sollte ein Softwarewerkzeug so gestaltet sein, dass notwendige Anpassungen an das jeweilige Zielsystem nur einmalig, ggf. mit Expertenunterstützung durchzuführen sind, um dann den EntwicklerInnen als einsatzfähige Konfiguration zur Verfügung zu stehen. Dies sollte jedoch die Anpassbarkeit durch die EntwicklerInnen an die jeweilige Einsatzsituation nicht einschränken.

6.4 Evolutionäre Entwicklung

Klimamodelle werden nicht auf der grünen Wiese geplant und entwickelt. Meist basieren „neue“ Modelle auf Vorgängermodellen und übernehmen große Teile des Codes von diesen. Von Zeit zu Zeit werden neue dynamische Kerne entwickelt, welche einen evolutionären Prozess vom zweidimensionalen Flachwassermodell bis hin zum hochentwickelten 3D-Modell durchlaufen, oder neue Physikmodule, welche sich in die vorhandene Modellarchitektur einfügen müssen. In den allermeisten Fällen ist davon auszugehen, dass eine EntwicklerIn es mit *Legacy Code* (Feathers, 2004) zu tun hat, d.h. mit Codefragmenten, für die noch keine Unittests existieren, und mit einer Softwarearchitektur, die nicht unbedingt unittestfreundlich aufgebaut ist. Gerade in so

einer Situation sollte die in dieser Arbeit entwickelte Softwarelösung Unterstützung bieten.

6.5 Fortran

Klimamodelle sind hauptsächlich in Fortran geschrieben. Die in dieser Arbeit entwickelte Lösung zielt daher auf Fortran-Systeme. Darüber hinaus verwenden viele Modelle separate Bibliotheken, die in C oder C++ geschrieben sind. Das Testen dieser Teile ist nicht Gegenstand dieser Arbeit, jedoch muss deren Existenz berücksichtigt werden. Die Verwendung von C/C++-Modulen aus Fortran-Code heraus sollte den Einsatz der vorgeschlagenen Lösung nicht verhindern.

Fortran-Prozeduren stellen in dieser Arbeit die Einheiten dar, die mit Hilfe von Unittests isoliert getestet werden sollen. Für die jeweils zu testende Prozedur eines Unittests wird im Folgenden die englische Abkürzung *PUT (procedure under test)* verwendet. Isoliert bedeutet, dass die PUT aus der umschließenden Anwendung herausgelöst und innerhalb eines Testtreibers direkt aufgerufen wird – mit Hilfe geeigneter Testdaten. Eine softwaregestützte Erstellung von Testdoubles für die von der PUT verwendeten Prozeduren wird nicht angestrebt. Somit stehen sog. Sociable Unit Tests (siehe Abschnitt 3.3) im Vordergrund.

6.6 Datenstrukturen

Im Mittelpunkt sollen Unittests für die fachlichen Prozeduren von Klimamodellen stehen, d.h. für den Teil des Codes, der für die klimatologischen Berechnungen zuständig ist. Für die Module der technischen Infrastruktur lassen sich Unittests in der Regel deutlich einfacher mit klassischen, manuellen Methoden erstellen, was zum Teil auch so gehandhabt wird.

Ein Softwarewerkzeug zur unterstützten Erstellung von Unittests für die fachlichen Prozeduren von Klimamodellen muss geeignet sein, Testdaten für deren typische Algorithmen und Datenstrukturen zu erzeugen. Klimamodelle realisieren im Wesentlichen numerische Verfahren mit Hilfe von Fließkomma-Berechnungen. Ein Großteil des fachlichen Codes besteht daher aus mathematischen Operationen mit Fließkommazahlen. Die Eingangsdaten einzelner Prozeduren bestehen dabei meist aus einer Vielzahl zum Teil sehr großer Arrays von Fließkommazahlen, die die Werte einzelner Variablen über das räumliche Gitter verteilt enthalten. Häufig werden auch mehrere (Array-)Variablen in benutzerdefinierten Verbunddatentypen zusammengefasst.

Innerhalb von Prozeduren wird oft mehrfach über diese Arrays iteriert und mathematische Operationen über ihre Elemente durchgeführt. Auch mathematische Fallunterscheidungen mit Hilfe von Vergleichsoperationen kommen vor, der größte Teil der Fallunterscheidungen bezieht sich jedoch auf Konfigurationsvariablen, in denen zum Beispiel festgelegt wird, ob bestimmte Berechnungen durchgeführt und welche Verfahren angewandt werden sollen. Häufig wird auch auf globale Variablen, d.h. öffentliche Variablen anderer Module, zugegriffen.

Eine Softwarelösung sollte also in der Lage sein, neben ganzzahligen und booleschen Testdaten auch Fließkommazahlen zu generieren. Dazu müssen Arrays und benutzerdefinierte Verbunddatentypen unterstützt werden. Tests für Prozeduren, die je nach Konfiguration in unterschiedlichen Variationen ausgeführt werden können, müssen unterstützt werden. Als Eingangsdaten müssen nicht nur ARGUMENTE, sondern auch globale Variablen berücksichtigt werden.

6.7 Parallelisierung

Alle großen Klimamodelle verwenden MPI zur Parallelisierung mittels Nachrichtenaustausch zwischen Prozessen und Rechnerknoten. Darüber hinaus kommt OpenMP für die Shared-Memory-Parallelisierung zum Einsatz. Es muss daher auch die Erstellung von Tests für Prozeduren unterstützt werden, die gewöhnlich parallel ausgeführt werden und MPI- und OpenMP-Konstrukte enthalten.

6.8 Hochleistungsrechner

Das zu entwickelnde Werkzeug muss unter den technischen Rahmenbedingungen funktionieren, die in dem Kontext, in dem sie eingesetzt werden soll, vorzufinden sind. Klimamodelle werden für die Ausführung auf Hochleistungsrechnern entwickelt. Auch ein Großteil der Entwicklungsarbeit, des Testens und des Debuggens findet in dieser Umgebung statt. Das bedeutet, dass die zu entwickelnde Lösung auf solchen Plattformen funktionieren muss. Dabei ist auch zu berücksichtigen, dass der EntwicklerInnen in einer solchen Umgebung i.d.R. über keine Administratorrechte verfügen. Das bedeutet, dass die Software auch mit einfachen Benutzerrechten funktionieren und sich ansonsten auf den üblicherweise vorhandenen Softwarestack stützen muss.

6.9 Weitere technische Anforderungen

Allgemeine Anforderungen an Fortran-Entwicklungswerkzeuge, welche im Anwendungsbereich Klimamodellierung eingesetzt werden sollen, werden von Mariano Méndez, Fernando G. Tinetti und Jeffrey L. Overbey (2014) formuliert. Die Autoren haben 16 Klimamodelle analysiert und verschiedene Metriken erfasst, wie die Anzahl der Codezeilen, die *zyklomatische Komplexität* (McCabe, 1976), die Anzahl der Präprozessordirektiven oder das Vorkommen obsoleter Fortran-Konstrukte. Darauf aufbauend sprechen sie drei Empfehlungen für Fortran-Entwicklungswerkzeuge (Fortran development tools) für Klimamodellcode aus, welche auch in dieser Arbeit berücksichtigt werden sollen:

1. Sie müssen mit Quelltextgrößen von mehreren hunderttausend Zeilen Code umgehen können.
2. Sie müssen sowohl alte als auch neue Sprachkonstrukte unterstützen.
3. Sie müssen Präprozessordirektiven verarbeiten können.

6.10 Zusammenfassung

In diesem Kapitel wurden Anforderungen an ein Softwarewerkzeug für die unterstützte Erstellung von Unittests für einzelne Prozeduren in Klimamodellen formuliert. Hierzu zählen u.a.:

- Die erstellten Unittests müssen für die EntwicklerInnen verständlich und nachvollziehbar sein.
- Eine halbautomatische Lösung ist einer vollautomatischen vorzuziehen.
- Das Werkzeug soll an die gewohnte Arbeitsweise der EntwicklerInnen anpassen und sich in deren vorhandenen Werkzeugkasten einfügen.
- Vor allem der initiale Aufwand der Testerstellung soll verringert werden.
- Das Werkzeug soll mit möglichst vielen Klimamodellen funktionieren.
- Legacycode muss unterstützt werden.
- Bei den zu testenden Einheiten handelt es sich um einzelne Fortran-Prozeduren.
- Das Vorhandensein von Modulen in anderen Sprachen, insbesondere C/C++ darf den Einsatz des Werkzeugs nicht verhindern.
- Zahlreiche große Arrays mit Fließkommadaten müssen unterstützt werden.

- Prozeduren, die MPI- und OpenMP-Konstrukte enthalten müssen unterstützt werden.
- Das Werkzeug muss auf Hochleistungsrechnern funktionieren.
- Das Werkzeug muss auch ohne Administratorrechte für den Rechner, auf dem es eingesetzt wird, funktionieren.
- Die erstellten Unittests sollen signifikant schneller als alternative E2E-Tests sein, d.h. mindestens zehnmal so schnell.
- Prozeduren aus großen Anwendungen von mehreren hunderttausend Zeilen müssen unterstützt werden.
- Alte und neue Fortran-Konstrukte müssen unterstützt werden.
- Das Vorhandensein von Präprozessordirektiven muss berücksichtigt werden.

Kapitel 7

Automatische Testdatengenerierung

Automatisierung spielt eine Schlüsselrolle, um den Aufwand für die Erstellung von Unittests zu verringern. Der größte Aufwand bei der Erstellung von Unittests entfällt auf die Erzeugung geeigneter Testdaten. Daher ließe sich hierbei durch Automatisierung die größte Aufwandsreduktion erreichen. In diesem Kapitel werden unterschiedliche Ansätze der automatischen Testdatengenerierung vorgestellt und hinsichtlich ihrer Eignung für eine anwendungsorientierte Lösung im Rahmen dieser Arbeit kritisch diskutiert, bevor im nachfolgenden Kapitel 8 der letztendlich ausgewählte *Capture-&-Replay*-Ansatz präsentiert wird.

Die automatische Generierung von Tests bzw. Testdaten ist seit den 1970er Jahren Gegenstand der Forschung. Dabei handelt es sich um ein recht unübersichtliches Forschungsfeld, in dem über die letzten Jahrzehnte verschiedene Ansätze entwickelt und verbessert wurden. Lehrbücher, die einen Gesamtüberblick über das Thema liefern, gibt es nach meinem Wissensstand nicht. Es existieren verschiedene Literatur- und Vergleichsstudien (siehe z.B. Ince, 1987; Burgess, 1993; Edvardsson, 1999; Rushby, 2008; Seung-Hee Han, 2008; Anand u. a., 2013; Galler und Aichernig, 2014; Shayma Mustafa Mohi-Aldeen, 2014; Mahadik und Thakore, 2016), jedoch verwendet jede ihre eigene Klassifizierung der Methoden und oft sind diese auf bestimmte Ansätze fokussiert, auch wenn sie allgemein lautende Titel tragen. Dennoch lassen sich verschiedene Hauptrichtungen bzw. -ansätze identifizieren. Zu den wichtigsten gehören die *symbolische Ausführung (symbolic execution)*, das *modellbasierte Testen (model-based testing)*, das *kombinatorische Testen (combinatorial testing)*, das *zufallsbasierte Testen (random testing)* sowie das *suchbasierte Testen (search-based testing)*[11]. Diese Einteilung verwenden auch Saswat Anand u. a. (2013).

[11] Die deutschen Bezeichnungen sind zum größten Teil selbst gewählt, da es zu den meisten Ansätzen keine deutschsprachige Literatur gibt.

7.1 Symbolische Ausführung

Bei der symbolischen Ausführung (symbolic execution, King, 1975) wird ein Programm mit Platzhaltervariablen anstelle von konkreten Werten als Eingabedaten ausgeführt. Als Ausgabedaten erhält man so Terme, welche die Eingangsvariablen als freie Variablen enthalten. Darüber hinaus lassen sich so für jeden Pfad, den das Programm nehmen kann, Bedingungen (*path constraints*) für die Eingangsvariablen, ebenfalls in Form von Termen, konstruieren. Mit Hilfe sog. *Constraint Solver* lassen sich diese auflösen und so konkrete Testdaten erzeugen.

Beispiel 7.1: Symbolische Ausführung

```
1  INTEGER FUNCTION example(a, b)
2
3    INTEGER, INTENT(in) :: a, b
4
5    IF (a < 0) THEN
6      example = 0
7    ELSE
8      IF (a > b) THEN
9        example = 1
10     ELSE
11       example = 2
12     END IF
13   END IF
14
15 END FUNCTION example
```

Diese Funktion kann drei verschiedene Pfade durchlaufen, welche jeweils mit unterschiedlichen Ergebnissen (0, 1 oder 2) enden, für die sich die folgenden Pfadbedingungen ergeben:

- Für den Pfad mit Ergebnis 0: $a < 0$
- Für den Pfad mit Ergebnis 1: $a \geqslant 0 \wedge a > b$
- Für den Pfad mit Ergebnis 2: $a \geqslant 0 \wedge a \leqslant b$

Da bei der symbolischen Ausführung der Testdatenerzeugung eine Analyse des zu testenden Programms vorausgeht, gehört dieser Ansatz zu den sog. *White-Box-Testverfahren*. Mit ihr sind drei grundlegende Probleme verbunden (vgl. Anand u. a., 2013):

Pfadexplosion Reale Programme können über extrem große Anzahlen von Pfaden verfügen, deren symbolische Ausführung in akzeptabler Zeit ggf. unmöglich ist.

Pfaddivergenz Reale Programme enthalten häufig Code in unterschiedlichen Programmiersprachen oder Teile, welche nur in Binärform vorliegen. In diesen Fällen müssen vom Benutzer ggf. Hilfestellungen, z.B. in Form von Modellen ge-

leistet werden. Dadurch kann es zu unpräzisen Pfadbedingungen kommen, was letztlich dazu führen kann, dass der Pfad, den ein Programm mit einem Satz generierter Eingabedaten nimmt, von dem eigentlich beabsichtigten Pfad abweicht.

Komplexe Bedingungen Da es kein allgemeingültiges Verfahren zu Lösung aller Bedingungen gibt, kann es vorkommen, dass bei der symbolischen Ausführung Pfadbedingungen erzeugt werden, die unlösbar sind, beispielsweise, wenn bestimmte mathematische Ausdrücke wie Multiplikation, Division oder Funktionen wie Sinus oder Cosinus enthalten sind.

Es gibt unzählige Forschungsarbeiten und Ansätze diesen Problemen zu begegnen, oder bestimmte Aspekte des Verfahrens zu optimieren. Häufig wird die symbolische Ausführung auch mit anderen Verfahren wie beispielsweise dem suchbasierten Testen kombiniert. Einen breiten industriellen Einsatz dieser Methode gibt es dennoch bisher nicht (vgl. Anand u. a., 2013). Zudem lässt die Zuverlässigkeit verfügbarer Systeme in diesem Bereich noch zu wünschen übrig (vgl. Kapus und Cadar, 2017).

Eine besondere Schwierigkeit für die symbolische Ausführung stellen zudem Fließkommazahlen dar, die in Klimamodellen oder ähnlichen Anwendung reichlich Verwendung finden. So lassen sich symbolische Berechnungen mit reellen oder rationalen Zahlen nicht eins zu eins auf Fließkommazahlen übertragen. Dies kann zu ungenauen und fehlerhaften Ergebnissen führen. So würden beispielsweise viele Constraint Solver Terme wie $x+(y+z)$ und $(x+y)+z$ als äquivalent behandeln, was im Bereich der Fließkommazahlen jedoch nicht zutrifft. Auch für diese Probleme existieren durchaus Lösungsansätze (siehe z.B. Michel, Rueher und Lebbah, 2001; Botella, Gotlieb und Claude, 2005; Bagnara u. a., 2013), insgesamt lässt sich für die symbolische Ausführung jedoch sagen, dass aufgrund der Komplexität des Ansatzes und seiner Erweiterungen eine Nachvollziehbarkeit für NichtexpertInnen kaum gegeben ist.

Hinzu kommt, dass mit Hilfe der symbolischen Ausführung zunächst nur Testeingabedaten erzeugt werden, aber keine Ausgangsdaten für die Verwendung als Orakel. Auch die Regressionstestannahme hilft hier nicht unbedingt weiter, denn die Beurteilung, ob eine zwar an sich als hinreichend „korrekt“ betrachtete Vorversion auch für einen beliebigen künstlich erzeugten Testdatensatz die richtigen Ergebnisse liefert, dürfte im Einzelfall sehr schwierig sein.

7.2 Modellbasiertes Testen

Modellbasiertes Testen (model-based testing) ist ein Obergriff für eine Vielzahl von Methoden, mit denen Tests bzw. Testdaten mit Hilfe von Modellen generiert werden, welche das gewünschte Verhalten einer zu testenden Software oder das Verhalten der

Umgebung, in der die Software eingesetzt wird, beschreiben (vgl. Utting, Pretschner und Legeard, 2012). In der Regel handelt es sich dabei um *Black-Box-Testverfahren*, bei denen lediglich das von außen beobachtbare Verhalten einer zu testenden Software betrachtet wird. Typische Grundlagen der Testmodelle sind somit beispielsweise Anforderungsdokumente oder Spezifikationen. Für die Modellbeschreibung werden diese in eine formale Sprache übertragen. Einer der ersten Ansätze modellbasierten Testens basierte auf *endlichen Automaten* (Chow, 1978), andere beispielsweise auf der UML (Shirole und Kumar, 2013) oder Petri-Netzen (Lill und Saglietti, 2012). Modellbasierte Tests können auf allen Ebenen des Testens eingesetzt werden, von Unittests bis zu komplexen Systemtests. Automatische Testorakel können mit diesem Verfahren ebenso erzeugt werden, sofern das Modell bzw. die Dokumente, aus denen es abgeleitet werden, entsprechende Informationen enthält.

Grundlegende Voraussetzung für diesen Ansatz ist, dass entsprechende Modelle bzw. die Dokumente, aus denen sie abgeleitet werden können, existieren. Im Kontext Klimamodellierung ist dies normalerweise nicht der Fall, auch ist es unwahrscheinlich, dass jemand bereit ist, Arbeitskraft in deren Erstellung zu investieren. Ließen sich mit Hilfe derartiger Modelle auf einfache Weise nützliche Unittests erstellen, dürfte dies die Motivation zwar erhöhen, jedoch ist es fraglich, in wie fern dies überhaupt möglich ist. Sehr detaillierte Modelle, die konkrete, manuell erstellte Testdaten enthalten, aus denen letztlich nur noch der entsprechende Testcode in Fortran erzeugt werden muss, dürften nur eine geringe Reduktion des Aufwands bedeuten und leiden ebenso wie die vollständig manuelle Testerstellung unter dem Problem, dass sich konsistente Testdaten nur schwerlich manuell erstellen lassen. Sehr grobe Modelle, aus denen konkrete Testdaten noch abgeleitet werden müssen, benötigen hierzu eines der anderen Verfahren zur automatischen Testdatenerzeugung und erben damit auch dessen spezifischen Probleme.

7.3 Kombinatorisches Testen

Eine spezielle Form des modellbasierten Testens stellt das kombinatorische Testen (combinatorial testing, Cohen u. a., 1996) dar. Dabei handelt es sich eine vergleichsweise einfach zu verstehende Form der automatischen Testgenerierung. Sie bietet sich dort an, wo Kombinationen einer überschaubaren Anzahl von Parametern mit jeweils überschaubarem Wertebereich (bzw. einer überschaubaren Anzahl an Äquivalenzklassen) getestet werden sollen, die Menge der möglichen Kombinationen jedoch so zahlreich ist, dass vollständiges Testen zu aufwendig wäre. Bei den Parametern, auch Faktoren genannt, handelt es sich meist um Eingabedaten, etwa von Formularen, oder um Konfigurationsvariablen, z.B. von Softwareproduktlinien. Aber auch

Kombinationen von Umgebungsvariablen, wie etwa Betriebssysteme, Bibliotheksversionen oder Compiler lassen sich so testen. Das Modell im Sinne des modellbasierten Testens besteht somit aus einer Auflistung der Parameter und den jeweils gültigen Werten. Strategien zur Reduktion und Auswahl von Testfällen basieren dabei auf der Annahme, dass nicht jeder Parameter zu jedem Fehler beiträgt, sondern dass Fehler meist auf der Interaktion einiger weniger Parameter basieren (vgl. Kuhn u. a., 2009).

Beispiel 7.2: Kombinatorisches Testen

Eine Webseite soll auf verschiedenen Betriebssystemen, mit verschiedenen Browsern und verschiedenen Versionen der Javascript-Bibliothek jQuery getestet werden. Für jeden Parameter gibt es zwei mögliche Optionen: Windows und Android als Betriebssystem, Firefox und Chrome als Browser sowie die jQuery-Versionen 2 und 3. Bildet man das kartesische Produkt über alle drei Eigenschaften, ergibt sich eine Gesamtzahl von Kombinationen von $2^3 = 8$:

Testfall	Betriebssystem	Browser	jQuery-Version	Testergebnis
1	Windows	Firefox	2	✖
2	Windows	Firefox	3	✔
3	Windows	Chrome	2	✔
4	Windows	Chrome	3	✔
5	Android	Firefox	2	✖
6	Android	Firefox	3	✔
7	Android	Chrome	2	✔
8	Android	Chrome	3	✔

Beim Durchtesten dieser acht Kombinationen stellt man nun fest, dass es die Kombination aus dem Browser Firefox und der jQuery-Version 2 ist, die Probleme bereitet, d.h. sowohl der Testfall 1 als auch der Testfall 5 schlagen fehl. Geht man nun von der Annahme aus, dass es in fast allen Fällen nur die Kombinationen aus zwei Parametern sind, die zu Fehlern führen, lässt sich die Anzahl der Testfälle auf vier reduzieren, wenn man diese so wählt, dass jedes Pärchen aus Parameteroptionen wenigsten einmal vorkommt.

Testfall	Betriebssystem	Browser	jQuery-Version	Testergebnis
1	Windows	Firefox	2	✖
4	Windows	Chrome	3	✔
6	Android	Firefox	3	✔
7	Android	Chrome	2	✔

Wie man sieht, wird auch so der Fehler bei der Kombination Firefox/jQuery 2 aufgedeckt.

Was in diesem Beispiel zu einer Reduktion von acht auf vier Testfälle geführt hat, kann beispielsweise bei fünf Parametern mit jeweils drei Optionen die Testfallmenge von $3^5 = 243$ auf elf Kombinationen reduzieren, sofern das Ziel paarweises Testen ist. Die Auswahl der Testfälle, so dass jedes Pärchen bzw. jedes gewünschtes n-Tupel abgedeckt ist, ist nicht ganz einfach. Hierbei kommen wiederum Softwarewerkzeuge ins Spiel, die diesen Prozess automatisieren, wobei es dazu verschiedene Strategien gibt, die Gegenstand der Forschung sind (vgl. Anand u. a., 2013). Beim kombinatorischen Testen handelt es sich also zunächst um ein Verfahren der automatischen Auswahl von Testfällen. Darüber, wie aus diesen Testfällen ablauffähige Testprogramme generiert werden, sagt das Verfahren nichts aus. Je nach Anwendungsbereich kann dies ggf. einfacher oder aufwendiger sein.

Bezogen auf das Testen von Klimamodellen würde sich das kombinatorische Testen insbesondere eignen, um zum einen Kombinationen der zahlreichen Konfigurationsoptionen als auch die verschiedenen Laufzeit- und Übersetzungsumgebungen zu testen. Für das Generieren von numerischen Testdaten stellt dieses Verfahren jedoch keine Lösung dar.

7.4 Zufallsbasiertes Testen

Die simpelste Form der automatischen Testgenerierung ist das zufallsbasierte Testen (random testing). Hierbei werden Testfälle durch die zufällige Auswahl von Eingabedaten aus dem gesamten Wertebereich gültiger Eingaben erzeugt. Laut Myers, Sandler und Badgett (2011, S. 41) handelt es sich hierbei um die ineffektivste Form der Testfallerzeugung:

> *„In general, the least effective methodology of all is random-input testing — the process of testing a program by selecting, at random, some subset of all possible input values. In terms of the likelihood of detecting the most errors, a randomly selected collection of test cases has little chance of being an optimal, or even close to optimal, subset.“*

Andere AutorInnen bescheinigen diesem Vorgehen wiederum durchaus brauchbare Ergebnisse zu liefern (siehe z.B. Duran und Ntafos, 1984; Ciupa u. a., 2011). Vorteile dieser Methode sind zum einen, dass sich so relativ einfach sehr viele Testfälle automatisiert erzeugen lassen, und zum anderen, dass diese frei von systematischen Fehlern bei der Testfallauswahl sind (vgl. Hamlet, 2002).

Eine Erweiterung des zufallsbasierten Ansatzes stellt das *adaptive zufallsbasierte Testen* (*adaptive random testing*, Chen, Leung und Mak, 2005) dar. Ziel des adaptiven zufallsbasierten Testens ist es, dass die zufällig ausgewählten Eingabedaten gleichmäßig über den Wertebereich verteilt sind. Ein mögliches Verfahren dies zu erreichen

besteht beispielsweise darin, im ersten Schritt zufällig einen Wert zu wählen; führt dieser im Test nicht zu einem Fehler, wird im nächsten Schritt aus einer größeren Menge zufällig gewählter Werte, dieser herausgegriffen, welcher im statischen Sinne am weitesten entfernt vom vorherigen Wert liegt.

Das größte Hindernis für den Einsatz von zufallsbasierten Testverfahren sind fehlende Testorakel. Bei willkürlich ausgewählten Eingabedaten sind die zu erwartenden Ausgabedaten zunächst unbekannt. Steht kein Referenzprogramm zu Verfügung, welches diese berechnen kann, muss dies händisch geschehen, was aber gerade, wenn es darum geht, sehr viele Testfälle zu erzeugen, in den meisten Fällen nicht praktikabel sein dürfte (vgl. Hamlet, 2002). Insbesondere in der Klimamodellierung, wo häufig keine analytischen Lösungen der zu berechnenden Probleme existieren, und wenn doch, diese mit sehr großem Aufwand verbunden sind, ist ein händische Berechnung i.d.R. unmöglich. Beim Regressionstesten könnte zwar eine Vorgängerversion zur Berechnung der zu erwartenden Ergebnisse verwendet werden, allerdings nur unter der Annahme, dass die Vorgängerversion für jede beliebige Eingabe das richtige Ergebnisse liefert. Davon kann in der Praxis normalerweise nicht ausgegangen werden.

Jedoch wenn einfachere Testorakel, beispielsweise die Überprüfung, ob die Ergebnisse in einem bestimmten Wertebereich liegen oder ob bestimmte Erhaltungssätze eingehalten werden, ausreichen, könnte das zufallsbasierte Testen Sinn machen. So könnte ein Algorithmus mit einer großen Zahl zufällig erzeugter Eingabewerte getestet werden, um anschließend eine zumindest statistische Aussage über seine Korrektheit zu treffen. Für Unittests, die der systematischen Suche von Programmierfehlern bei der Entwicklung dienen sollen, erscheint allerdings auch dies ungeeignet. Aufgrund der Anforderung nach kurzer Laufzeit, geht es nicht darum, möglichst viele, willkürlich ausgewählte Testfälle auszuführen, sondern eine kleine Zahl aussagekräftiger Testfälle. Hinzu kommt, dass Testdaten konsistent sein müssen, was beispielsweise Array- und Blockgrößen angeht. Dies zu erreichen stellt eine der größten Herausforderungen bei der Erzeugung von Testdaten dar (siehe auch Abschnitt 4.9) und lässt sich nicht mit Hilfe von Zufallsgeneratoren lösen.

7.5 Suchbasiertes Testen

Beim suchbasierten Testen (search-based testing) werden Testdaten nicht zufällig ausgewählt, sondern mit Hilfe von Optimierungsalgorithmen gesucht. Dabei werden mögliche Testdaten in der Regel „ausprobiert“, indem das zu testende Programm mit ihnen ausgeführt und anschließend mit Hilfe einer *Fitnessfunktion* (*fitness function*) bewertet wird. Die Fitnessfunktion repräsentiert dabei ein festgelegtes Testziel bzw. den Grad der Erreichung dieses Ziels. Ein Testziel könnte z.B. die Ausführung eines bestimmten Programmpfades sein. Die Fitnessfunktion würde in diesem Fall die

Ähnlichkeit des tatsächlich ausgeführten Pfades mit dem gewünschten Pfad berechnen. Für die Suche nach optimalen Testdaten werden einfache Metaheuristiken wie z.B. der Bergsteigeralgorithmus verwendet, aber auch evolutionäre bzw. genetische Algorithmen (vgl. McMinn, 2011).

Da grundsätzlich jedes beliebige Testziel in eine Fitnessfunktion umgewandelt werden kann, handelt es sich beim suchbasierten Testen um einen sehr generischen Ansatz, welcher bereits in vielen Anwendungsbereichen erprobt wurde (vgl. Anand u. a., 2013). Als erster vergleichbarer Ansatz gilt die Arbeit von Webb Miller und David L. Spooner (1976), in welcher numerische Maximierungsmethoden als Alternative zur symbolischen Ausführung für die Generierung von Fließkomma-Testdaten vorgeschlagen werden (vgl. McMinn, 2011). Das suchbasierte Testen könnte somit durchaus ein Kandidat sein, um numerische Fließkommatestdaten für Klimamodelle zu erzeugen. Ob sich auf dieser Basis eine anwendungsorientierte Softwarelösung zur Erstellung von Unittests erstellen ließe, erscheint jedoch fraglich. Für diesen Einsatzzweck kämen im Wesentlichen zwei Arten von Testzielen in Frage: fachliche und strukturelle.

Fachliche Testziele könnten z.B. so aussehen, dass nach Eingabedaten gesucht wird, die ein ganz bestimmtes Verhalten des zu testenden Programms/Algorithmus auslösen. In diesem Fall müsste für jede individuelle Fragestellung eine geeignete Fitnessfunktion definiert und ein passender Suchalgorithmus gewählt und ggf. implementiert werden. Es ist denkbar, dass solch ein Vorgehen in einige Testszenarien nützlich sein könnte, für die schnelle Generierung von Unittests als Alternative zu den gewohnten Ende-zu-Ende-Tests erscheint es jedoch zu aufwändig.

Strukturelle Testziele beziehen sich z.B. auf die Ausführung bestimmter Programmpfade oder eine gewünschte Codeabdeckung. Hier wären ggf. generische Lösungen möglich, die im Einzelfall mit weniger Aufwand verbunden wären. Ob sich mit einer allgemeinen Lösung jedoch fachlich sinnvolle Testdaten erzeugen ließen, die von der EntwicklerIn als nützlich akzeptiert würden, erscheint fraglich.

Auch das suchbasierte Testen leidet darunter, dass es zwar eine Methode zum automatischen Erzeugen von Testeingabedaten darstellt, aber kein Verfahren zum automatischen Erzeugen von Testorakeln beinhaltet. In vielen Fällen ist man daher auch hier auf menschliche Orakel, d.h. auf das manuelle Überprüfen der Ergebnisse angewiesen. Es existieren Forschungsansätze, den Aufwand hierfür zu reduzieren, indem die Verständlichkeit der Testfälle für Menschen bei der Generierung der Testdaten berücksichtigt wird (siehe z.B. McMinn, Stevenson und Harman, 2010; Fraser und Zeller, 2011). Da es jedoch beim suchbasierten Testen anders als beim zufallsbasierten nicht darum geht, möglichst viele Testfälle, sondern stattdessen möglichst wenige, besonders nützliche auszuwählen, könnten beim Regressionstesten automatische Orakel durchaus mit Hilfe von Vorversionen erstellt und nach einmaliger menschlicher Überprüfung verwendet werden.

7.6 Anwendungen und Evaluationen

In der Literatur finden sich Erprobungen der vorangehend aufgeführten Ansätze aus diversen fachlichen Domänen, von einer Etablierung im industriellen Kontext lässt sich jedoch nicht sprechen, so etwa eine Einschätzung von Anand u. a. (2013):

> *„Although automation techniques for test case generation have started to be gradually adopted by the IT industry in software testing practice, there still exists a big gap between real software application systems and the practical usability of the test case generation techniques proposed by the research community.“*

Berichte über den Einsatz einer der vorgestellten Ansätze in der Entwicklung von Klimamodellen sind nicht bekannt. Für jede der beschriebenen Methode existieren jedoch produktiv einsetzbare Softwarelösungen, sowohl kommerzielle als auch frei verfügbare. Viele davon sind speziell für die Generierung von Unittests ausgelegt und zielen häufig auf objektorientierte Programmiersprachen wie Java, C++ oder C# (vgl. Anand u. a., 2013; Mahadik und Thakore, 2016; Czerwonka, 2018). Bis in die frühen 1990er-Jahre entstanden zwar auch Testgeneratoren für Fortran-Programme, insbesondere auf Basis der symbolischen Ausführung (siehe z.B. Coward, 1988; Korel, 1990; Offutt, 1991). Diese sind inzwischen jedoch veraltet und in der Regel nicht mehr erhältlich.

Zudem existieren verschiedene Evaluationen aktueller Testgeneratoren. Typische Ansätze messen dabei die Code- bzw. Pfadabdeckung der generierten Tests (Fraser und Arcuri, 2014; Cseppentő und Micskei, 2017; Panichella und Molina, 2017), die Dauer der Testgenerierung (Cseppentő und Micskei, 2017; Panichella und Molina, 2017) oder die Fähigkeit der generierten Tests echte (Shamshiri u. a., 2015; Almasi u. a., 2017) oder künstliche (Wang und Offutt, 2009; Cseppentő und Micskei, 2017; Panichella und Molina, 2017) Programmierfehler aufzudecken. Galler und Aichernig (2014) sowie Cseppentő und Micskei (2015) haben zudem Testgeneratoren systematisch auf die Unterstützung verschiedener Sprachkonstrukte hin überprüft. Einige Arbeiten gehen auch auf anwendungsorientierte Aspekte ein, wie beispielsweise die Lesbarkeit (Fraser, Staats u. a., 2013; Rojas, Fraser und Arcuri, 2015) oder Wartbarkeit (Palomba, Di Nucci u. a., 2016; Palomba, Panichella u. a., 2016) des generierten Testcodes oder den Einfluss automatisch generierter Tests auf das Debugging (Ceccato u. a., 2015).

7.7 Zusammenfassung

In diesem Kapitel wurden fünf verschiedene Ansätze zur automatischen Testdatengenerierung vorgestellt und auf ihre Eignung für die Erzeugung von Unittests in der

Klimamodellierung hin untersucht. Vorgestellt wurden die symbolische Ausführung, das modellbasierte Testen, das kombinatorische Testen, das zufallsbasierte Testen und das suchbasierte Testen. Jeder dieser Ansätze ist mit spezifischen Nachteilen verbunden, die einem praktikablen, anwendungsorientierten Einsatz im Kontext Klimamodellierung im Wege stehen.

Die Untersuchung erfolgte nicht empirisch, sondern rein argumentativ. Ein erfolgreicher Einsatz kann somit für keinen der betrachteten Ansätze vollständig ausgeschlossen werden. Da für keinen eine einsatzfähige Lösung für Fortran-Anwendungen existiert, müsste diese für eine Erprobung zunächst erschaffen werden. Im Rahmen dieser Arbeit ist dies jedoch nur für einen Ansatz möglich. Unter Berücksichtigung der Ergebnisse der vorangegangenen Untersuchung erscheint dabei das im nachfolgenden Kapitel vorgestellte Capture & Replay am vielversprechendsten.

Kapitel 8

Capture & Replay

Meist wird das Problem der automatischen Generierung von Tests und Testdaten aus einer mathematisch-theoretischen oder technologischen Perspektive betrachtet, d.h. es werden Methoden vorgeschlagen, deren Ziel z.B. eine optimale Pfadabdeckung, Fehlererkennung oder Kostenreduktion ist. In dieser Arbeit wird von einer anwendungsorientierten Perspektive ausgegangen, d.h. Ziel ist eine gebrauchstaugliche Lösung für den Einsatz in der vorgesehenen Umgebung, nämlich der Entwicklung von Klimamodellen. Nichtsdestotrotz soll die zu entwickelnde Lösung dem eigentlichen Ziel, nützliche Unittests für die Suche nach Programmfehlern zu erzeugen, dienlich sein. Zudem soll sie den Aufwand für die Erstellung derartiger Test auch tatsächlich signifikant verringern.

Eine Alternative zu den im vorangehenden Kapitel vorgestellten Ansätzen zur automatischen Testgenerierung ist die Methode des *Capture & Replay (C&R)*. Dabei werden Ein- und Ausgabedaten einzelner Komponenten während der Ausführung der zu testenden Anwendung, beispielsweise im Rahmen größerer Systemtests, aufgezeichnet, um anschließend als Testdaten und -orakel für Regressionstest zu dienen. Da der konkrete Testfall von der EntwicklerIn/TesterIn selbst ausgewählt wird und i.d.R. nur die Datenaufzeichnung und Testerstellung automatisiert wird, handelt es sich um ein halbautomatisches Verfahren, welches in der Literatur unter dem Stichwort „automatische Testdatengenerierung“ häufig unerwähnt bleibt. Besonders beim Testen von grafischen Benutzeroberflächen ist es jedoch ein beliebter Ansatz zur Testgenerierung, aber auch für Tests auf Codeebene wurde C&R bereits erprobt.

In diesem Kapitel werden zunächst die Grundlagen des C&R-Ansatzes erläutert (Abschnitt 8.1). Anschließend wird sich mit der Frage auseinandergesetzt, wie C&R verwendet werden kann, um Unittests für Klimamodelle zu erstellen (Abschnitt 8.2). In Abschnitt 8.3 werden Designoptionen einer möglichen Softwarelösung diskutiert. Abschließend werden weitere Einsatzmöglichkeiten des C&R-Ansatzes (Abschnitt 8.4) sowie andere Arbeiten in diesem Bereich betrachtet (Abschnitt 8.5).

8.1 Grundlagen

Beim Capture & Replay werden kurze Programmsequenzen lang laufender oder interaktiver Programme reproduziert, indem das Programm (*Originalanwendung*) ausgeführt wird und dabei die Eingabedaten der zu reproduzierenden Sequenz (*Zielsequenz*) aufgezeichnet werden (Capture). Das Programm bzw. der betreffende Programmteil kann anschließend mit diesen Eingabedaten erneut ausgeführt werden (Replay). So können zum Beispiel Testdaten für zu testende Codeabschnitte, welche bereits in eine größere Anwendung integriert sind, erstellt werden. Die Zielsequenz ist in diesem Fall eine ausgewählte Ausführung des zu testenden Codes. Zum Zwecke des Regressionstesten lassen sich neben den Eingabedaten auch die Ausgabedaten der Zielsequenz aufzeichnen, um diese als Testorakel zu verwenden.

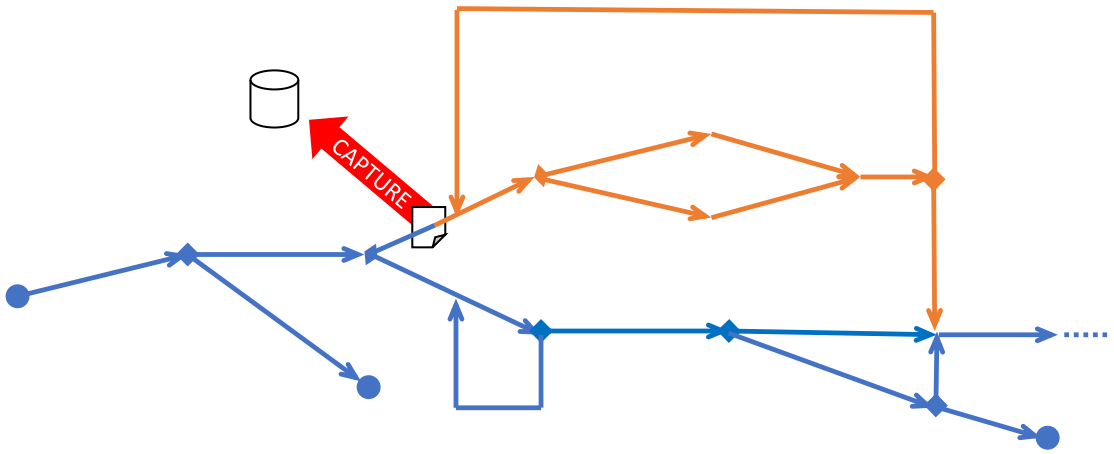
Abbildung 8.1 zeigt ein Beispiel dieses Vorgehens. Dargestellt ist der Programmfluss einer fiktiven Anwendung mit Verzweigungen, Schleifen etc. Die Zielsequenz, welche per C&R reproduziert werden soll, ist orange markiert, der Rest der Anwendung blau. Beim Capture wird die Originalanwendung einmal ausgeführt. Erreicht sie den Beginn der Zielsequenz wird ihr aktueller Zustand aufgezeichnet. Dabei ist es ausreichend, wenn der Teil des Zustands, d.h. die Datenfelder aufgezeichnet werden, welche Einfluss auf das Verhalten der Zielsequenz haben. Diese Teilmenge des Zustands stellt die Eingabedaten der zu reproduzierenden Sequenz dar. Beim Replay werden diese Eingabedaten geladen und die Zielsequenz erneut ausgeführt. Verhält sich die Zielsequenz deterministisch und wurden alle benötigte Eingabedaten berücksichtigt, sollte sie beim Replay das gleiche Verhalten zeigen wie beim Capture.

Um das Ausführen der Zielsequenz zu ermöglichen, muss es eine Form von Einstiegspunkt geben, der es erlaubt, sie unabhängig vom umschließenden Programm auszuführen. Handelt es bei der Zielsequenz um ein Unterprogramm (Prozedur) ist dies i.d.R. möglich, in dem man ein neues Hauptprogramm erstellt, welches nur die Eingabedaten einliest und die Prozedur aufruft (violett dargestellt). Im Rahmen von Tests würde man dieses als Testtreiber bezeichnen.

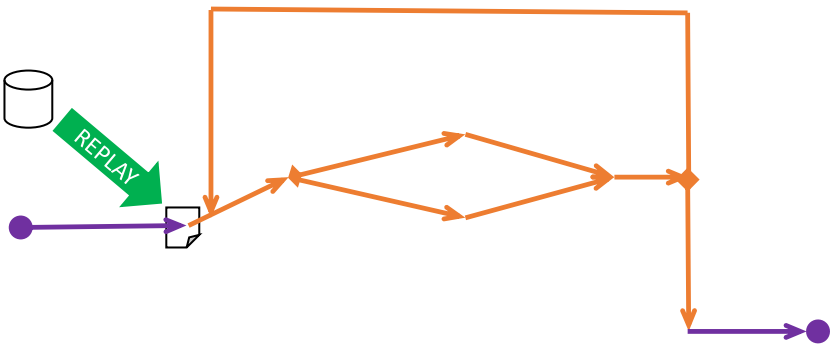
Bei interaktiven Programmen gehören neben dem Anfangszustand auch die Interaktionen während Ausführung der Zielsequenz zu den Eingabedaten (nicht dargestellt). Dabei kann es sich beispielsweise um Eingaben von BenutzerInnen als auch um Interaktionen mit der Laufzeitumgebung oder anderen Systemen handeln.

8.1.1 Capture & Replay für Tests grafischer Benutzeroberflächen

Insbesondere beim Testen graphischer Benutzeroberflächen *(graphical user interface, GUI)* kommt C&R zum Einsatz, um Benutzeraktionen aufzuzeichnen und wieder abzuspielen. Die ersten Softwarelösungen entstanden Anfang der 1990er Jahre, wie



(a) Capture



(b) Replay

Abbildung 8.1: Schematische Darstellung einer per Capture & Replay reproduzierten Zielsequenz (orange)

etwa der *Instrumented Interface Construction (IICON) Evaluator* (Hicinbothom und Zachary, 1993; Hicinbothom, 1996). Ein übliches Vorgehen beim Capture besteht darin, Benutzerereignisse (Events) wie etwa Mausbewegungen oder das Drücken von Tasten auf der Tastatur auf Ebene der GUI-Laufzeitumgebung abzufragen und in sog. Testskripts zu protokollieren. In der Regel gibt es daher für verschiedene GUI-Technologien spezialisierte C&R-Programme, welche auch für das Replay der protokollierten Benutzerereignisse zuständig sind. Sie werden daher auch Testroboter genannt (vgl. Spillner und Linz, 2012, S. 217).

Einige Testroboter bieten auch die Möglichkeit, Testskripte nachträglich zu modifizieren. So lassen sich leicht Variationen der gleichen Interaktionsabfolge erstellen. Zusätzlich lassen sich Testorakel in Form von Sollzuständen, die die GUI einnehmen soll, in die Testskripte aufnehmen. Diese können sowohl das Layout (z.B. Farbe, Größe, Position von GUI-Elementen) betreffen als auch funktionale Eigenschaften, wie z.B. textuelle Inhalte (vgl. Spillner und Linz, 2012, S. 217).

Ein Problem des Ansatzes GUI-Interaktionen auf Ebene der Laufzeitumgebung aufzuzeichnen, ist, dass er sehr sensitiv auf kleinste Änderungen der GUI reagiert. Werden beispielsweise Mausbewegungen in Form von x/y-Koordinaten aufgezeichnet und ändert sich die Position eines Buttons, wird dieser anschließend vom Testroboter ggf. nicht mehr „getroffen“. Eine Lösung besteht darin, Benutzerinteraktionen nicht auf GUI-, sondern auf Codeebene aufzuzeichnen (siehe z.B. Silverstein, 2003; Sjösten-Andersson und Pareto, 2006).

8.1.2 Modulares Regressionstesten

Außer zum Testen von grafischen Benutzeroberflächen, lässt sich C&R auch dazu verwenden, auf Codeebene Testdaten für einzelne Codeabschnitte eines größeren Programms zu extrahieren. Diese Codeabschnitte können beispielsweise einzelne Module sein oder Prozeduren, welche im Mittelpunkt dieser Arbeit stehen.

Das Prinzip des C&R zum Testen einzelner Module größerer Softwaresysteme wurde bereits von Bruce W. Weide (2001) beschrieben. Weide beschreibt ein Gedankenexperiment, um zu zeigen, dass die meisten realen Softwaresysteme keine *modulare Schlussfolgerbarkeit (modular reasoning property)* enthalten. Ein Softwaresystem verfügt laut Weide dann über modulare Schlussfolgerbarkeit, wenn sich das Verhalten jeder seiner Komponenten ausschließlich durch Betrachtung des jeweiligen Quellcodes dieser Komponente fundiert ableiten lässt. Weide argumentiert, dass sich diese Eigenschaft nur erreichen lasse, wenn die Komponenten mit Bedacht und Disziplin auf dieses Ziel hin entworfen und in modernen imperativen, objektorientierten Sprachen implementiert werden. Er behauptet zudem, dass die meisten realen Softwaresystem diese Eigenschaft nicht besitzen.

Um dies zu veranschaulichen führt er das Prinzip des *modularen Regressionstestens* ein. In einer „idealen Welt“ (d.h. unter der Annahme der modularen Schlussfolgerbarkeit) müssten nach Änderungen an einer Komponente nicht Regressionstest für das gesamte System durchgeführt werden, sondern nur für die geänderte (oder ausgetauschte) Komponente. Um dies zu ermöglichen schlägt Weide folgendes C&R-Verfahren vor:

- *„Log* [Capture] *every call to P while testing the software system the first time, recording the values of P's arguments and any global data P refers to, upon each call and upon each corresponding return.“*
- *„‚Play back‘* [Replay] *the calls to P into a simple test driver for P', comparing the results produced by P' to the recorded results produced by P for the same series of calls.“*

P steht hier stellvertretend für die zu ändernde Komponente und P' für eine neue Version der Komponente. Komponenten sind in Weides Beispiel Funktionen einer C++-Anwendung.

Wenn ein vollständiger „traditioneller“ Regressionstest keine weiteren Fehler aufdecken *könnte*, sei dieses Vorgehen tragfähig und könnte vollständige Regressionstests zuverlässig ersetzen. Wenn jedoch ein vollständiger Regressionstest weitere Fehler aufdecken könnte, dann würde dieser ohnehin benötigt und das Konzept des modularen Regressionstestens sei somit unnütz.

Weide führt aus, dass entscheidenden Einfluss auf die Tragfähigkeit des modularen Regressionstestens hat, ob das Verhalten einzelner Komponente von privaten, d.h. gekapselten, von außen nicht zugreifbaren, Daten abhängt. Als Beispiel nennt Weide u.a. eine Komponente U, dessen Verhalten von einer privaten Variable abhängt, die die Anzahl der Aufrufe von U zählt. Wenn nun P U nicht verwendete, P' aber schon, könnte der Wechsel von P zu P' Auswirkungen auf das Verhalten anderer Komponenten haben, die ebenfalls U verwenden. Modulares Regressionstesten von P' wäre somit nicht mehr ausreichend, um die Auswirkungen auf das Gesamtsystem zu testen.

Die Lösung wäre hier, so Weide, von einem *erweiterten Modul* auszugehen und nicht nur die Aufrufe von P zu erfassen, sondern auch alle Aufrufe von U und vergleichbarer Komponenten. Reale Softwaresysteme seien jedoch häufig nicht modular schlussfolgerbar, da die Wechselwirkungen zwischen einzelnen Komponenten zu zahlreich sind. Somit sei es auch schwerlich möglich, mit Hilfe von Reverse Engineering ein erweitertes Modul im o.g. Sinne einzugrenzen.

Mit ausgefeilten Programmanalysewerkzeugen sei es vielleicht möglich, das erweiterte Modul zu bestimmen. Gelänge dies, sei die These des schwierigen Reverse Engineering widerlegt:

> *„someone just needs to carry out the modular regression testing thought experiment in an empirical study of actual software systems in an attempt to show that the expanding module phenomenon is not severe for ‚real' software."*

Der in dieser Arbeit verfolgte Ansatz ist insofern vergleichbar mit dem Konzept des modularen Regressionstesten als dass hier ebenso Aufrufe einzelner Prozeduren zum Zwecke des Testens per C&R reproduziert werden. Dabei soll es sich jedoch nicht unbedingt um Aufrufserien handeln, sondern zunächst um einzelne, unabhängige Aufrufe. Wie später noch ausgeführt wird, wird auch die von Weide angesprochene „ausgefeilte" Programmanalyse zu diesem Ansatz gehören. Damit sollen sämtliche Variablen bestimmt werden, die während der Ausführung gelesen oder geschrieben werden. Dies ist vergleichbar mit der Bestimmung des „erweiterten Moduls". Um die Auswirkungen der Ausführung auf das gesamte erweiterte Modul zu bestimmen, müssen nicht zwangsläufig alle Aufrufe der enthaltenen Prozeduren von außerhalb aufgezeichnet werden, wie von Weide vorgeschlagen. Ein praktikablerer Ansatz wäre es beispielsweise die Zugreifbarkeit der privaten Variablen zum Zwecke des Testens zu erweitern. Somit würde die Auswirkung auf den Zustand des Systems anstelle des von außen beobachtbaren Verhaltens geprüft werden.

Ein entscheidender Unterschied zwischen Weides Konzept und dem dieser Arbeit ist die Zielsetzung. Man kann davon ausgehen, dass die modulare Schlussfolgerbarkeit für Klimamodelle nicht gegeben ist. Zwar sind diese in der Regel stark modularisiert, die Wechselwirkungen zwischen den Modulen jedoch groß. Nach Weide wäre der Ansatz des modularen Regressionstestens damit zum Scheitern verurteilt. Ziel dieser Arbeit ist es jedoch nicht, dass die per C&R erstellten Unittests die üblichen E2E-Regressionstests vollständig ersetzen, sondern dass sie diese ergänzen. Da es zudem nicht darum geht, Fehlerfreiheit nachzuweisen, sondern Fehler aufzudecken, haben Unittests auch dann ihren Wert, wenn sie dies in vielen Fällen schneller tun können als E2E-Tests, auch wenn dabei Fehler übersehen werden, die ein Test des Gesamtsystems wiederum aufdecken könnte. Dies würde die Ausführung der E2E-Tests zwar nicht überflüssig machen, jedoch könnte die Frequenz, mit der diese ausgeführt werden, gesenkt werden. Hinzu kommt die leichtere Lokalisierung der Ursachen von durch Unittests aufgedeckten Fehlern. Für eine allgemeine Diskussion der Vorteile von Unittests gegenüber der ausschließlichen Verwendung von E2E-Tests sei zudem auf Kapitel 5 verwiesen. Somit erscheint der Ansatz, Unittests per C&R zu erstellen, auch unter der Annahme nützlich zu sein, dass modulare Schlussfolgerbarkeit nicht gegeben ist.

8.1.3 Checkpoint/Restart

Ein dem C&R verwandter Ansatz, welcher im HPC und insbesondere auch in Klimamodellen häufig zum Einsatz kommt, ist das *Checkpoint/Restart*-Prinzip, auch *Rollback und Recovery* (Chandy und Ramamoorthy, 1972) genannt. Dabei wird zu definierten Zeitpunkten (Checkpoints) während der Ausführung einer Anwendung ihr Zustand so aufgezeichnet, dass sie ausgehend von diesem Zustand erneut gestartet werden kann (Restart). Damit soll verhindert werden, dass langlaufende Programme, wie etwa Klimasimulationen, erneut von vorne beginnen müssten, sollte es während der Ausführung zu einem Hard- oder Softwarefehler kommen.

Checkpoint/Restart-Mechanismen können auf Anwendungs- oder Betriebssystemebene realisiert werden. Daneben gibt es auch Zwischenformen, in Form von Bibliotheken, die in eine Anwendung eingebunden werden können. Auch Realisierung auf Hardwareebene existieren (vgl. Egwutuoha u. a., 2013). Checkpoint/Restart unterhalb der Anwendungsebene sind in der Regel flexibler bzgl. des Setzen von Checkpoints. Implementierungen auf Anwendungsebene lassen meist nur Checkpoints zu bestimmten, von der Anwendung definierten Zeitpunkten zu. Dafür können bei diesem Ansatz viel gezielter die Daten zur Aufzeichnung ausgewählt werden, die nötig sind, um die Anwendung wieder in den Zustand am Checkpoint zu bringen. Dadurch ist die aufzuzeichnende Datenmenge in der Regel kleiner als bei einem kompletten Speicherabzug, welcher bei den anderen Ansätzen nötig ist. Zudem ist dieses Vorgehen portabler, d.h. Checkpoints können auf anderen Rechnern oder mit anderen Compilern oder Compileroptionen übersetzten Anwendungen wiederverwendet werden.

Klimamodellen enthalten in der Regel Checkpoint/Restart-Implementierung auf Anwendungsebene. Dies hat historische Gründe, da aufgrund der langen Laufzeit von Klimasimulationen schon immer die Notwendigkeit für derartige Mechanismen bestand, schon bevor andere Ansätze überhaupt verfügbar waren. Üblicherweise greifen in Klimamodellen die Checkpoint/Restart-Mechanismen zu Beginn (oder Ende) eines Zeitschritts. In der Modellkonfiguration wird festgelegt, mit welchem Abstand, d.h. nach wie vielen Zeitschritten eine Aufzeichnung stattfinden soll.

Mit Hilfe dieses Verfahrens lassen sich nicht nur Simulationen im Fall eines Fehlers leichter fortsetzen, auch kann es im Rahmen von Tests dazu verwendet werden, bestimmte Zustände zu reproduzieren. Für Unittests sind die in Klimamodelle eingebauten Checkpoint/Restart-Mechanismen allerdings nicht geeignet, da die Aufzeichnungen eben nur am Anfang (oder Ende) eines Zeitschritts stattfinden und nicht gezielt vor Ausführung einer bestimmten Prozedur. Zudem werden dabei viele Daten aufgezeichnet, die für die Ausführung der zu testenden Prozedur ggf. gar nicht nötig sind. Andere Daten, die wiederum nur vor Ausführung der Prozedur, aber nicht zum Zeitpunkt der Aufzeichnung existieren, würden zudem fehlen. Für die Erstellung von Unittests mit Hilfe von C&R wird daher ein anderes Verfahren benötigt, mit dem sich

Aufzeichnungszeitpunkt und die Menge der aufzuzeichnenden Daten gezielt steuern lässt.

8.1.4 Zustands- und aktionsbasiertes Capture & Replay

C&R-Verfahren lassen sich in *zustands-* und *aktionsbasierte* Verfahren einteilen (siehe auch Elbaum u. a., 2009). Bei zustandsbasierten Verfahren wird beim Capture möglichst umfassend der Zustand der Anwendung in Form von Variablenbelegungen etc. aufgezeichnet, um diesen beim Replay wiederherzustellen, indem die aufgezeichneten Daten auf geeignete Weise injiziert werden. Das Checkpoint/Restart-Verfahren arbeitet beispielsweise nach diesem Muster.

Aktionsbasierte Verfahren zeichnen beim Capture ausgehend von einem Zustand Null (z.B. dem Programmstart) alle Interaktionen der Anwendung mit ihrer Umgebung (z.B. mit einer BenutzerIn) auf, bis der gewünschte Zielzustand erreicht ist. Beim Replay werden schließlich die aufgezeichneten Aktionen automatisiert wiederholt, um den Zielzustand wiederherzustellen. Die in Abschnitt 8.1.1 beschriebenen Testroboter für grafische Benutzeroberflächen verwenden i.d.R. dieses Verfahren.

Da Klimamodelle eigenständig ohne Interaktionen mit BenutzerInnen oder ihrer Umgebung laufen, kommt ein aktionsbasiertes C&R auf dieser Ebene nicht in Frage. Doch auch auf Codeebene lassen sich aktionsbasierte Verfahren einsetzen, etwa indem man die Interaktionen (z.B. Prozeduraufrufe) einer Zielkomponente mit anderen Komponenten des Systems aufzeichnet. Das in Abschnitt 8.1.2 beschriebene, von Weide (2001) vorgeschlagene Verfahren, basiert auf diesem Ansatz. Wie von Weide selbst ausgeführt, müssen dabei jedoch nicht nur die Interaktionen der eigentlichen Zielkomponente berücksichtigt werden, sondern alle Interaktionen des „erweiterten Moduls“, d.h. aller Komponenten, die Einfluss auf das Verhalten der Zielkomponente haben oder andersherum. Dies betrifft nicht nur die Interaktionen der Komponenten des erweiterten Moduls untereinander, sondern auch alle Interaktionen mit Drittkomponenten. Bei sehr vielen Abhängigkeiten, kann dies im schlimmsten Fall bedeuteten, dass letztendlich alle Interaktionen zwischen den Komponenten einer Anwendung vom Programmstart an aufgezeichnet und reproduziert werden müssen, also das gesamte Programm noch einmal ablaufen muss. Prozeduren von Klimamodellen greifen zum Teil auf sehr viele globale Variablen, auch von anderen Modulen zu. Die Wahrscheinlichkeit, dass ein aktionsbasiertes Vorgehen wie beschrieben ausartet, ist somit nicht gering. Stattdessen den Zustand der benötigten globalen Variablen und anderer Datenfelder zu einem gewünschten Zeitpunkt aufzuzeichnen, erscheint daher das passendere Vorgehen zu sein. Im Folgenden wird daher nur die zustandsbasierte Variante des Capture & Replay betrachtet.

8.2 Unittests auf Basis von Capture & Replay

Ähnlich den von Weide beschriebenen *modularen Regressionstests* (Abschnitt 8.1.2) könnte C&R auch dazu verwendet werden, Unittests einzelner Prozeduren für Regressionstests zu erstellen. Hierzu muss die zu testende Prozedur entweder bereits existieren oder zumindest die Aufrufstelle und die benötigten Eingabedaten feststehen. In den meisten Fällen, in denen Regressionstests benötigt werden, ist dies der Fall, schließlich dienen Regressionstests dazu, Änderungen an existierenden Codeabschnitten zu testen. Im Falle von Klimamodellen könnten existierende E2E-Tests verwendet werden, um Testdaten für Unittests zu generieren und per C&R bereitzustellen.

Im Gegensatz zu den meisten der in Kapitel 7 beschriebenen Verfahren zur automatischen Testdatengenerierung dürfte dieses Vorgehen leicht verständlich und nachvollziehbar sein. EntwicklerInnen von Klimamodellen verwenden bisher vorwiegend die genannten E2E-Tests auf Grundlage einiger typischer Experimente, um die Modelle zu testen. Leitet man nun Unittests mit Hilfe von C&R aus diesen E2E-Tests ab, entstehen keine willkürlichen oder künstlichen Testdaten, sondern solche, die die EntwicklerInnen ohnehin zum Testen verwenden, mit dem einzigen Unterschied, dass diese nun in kleineren, schnelleren und fokussierteren Tests verwendet werden können. Hinzu kommt die Ähnlichkeit dieses Vorgehens mit dem in der Klimamodellierung weit verbreiteten Checkpoint/Restart-Verfahren (siehe Abschnitt 8.1.3).

Im Folgenden wird zunächst das grundlegende Vorgehen beschrieben, mit dem Unittests mit Hilfe von C&R erstellt werden können (Abschnitt 8.2.1). Daraufhin wird diskutiert, wie sich zum einen auch Testorakel auf diese Weise erzeugen lassen (Abschnitt 8.2.2) und sich zum anderen weitere Testfälle von aufgezeichneten Daten ableiten lassen (Abschnitt 8.2.3). Anschließend wird betrachtet, was bei der Testdatenauswahl zu beachten ist (Abschnitt 8.2.4) und welche Ein- und Ausgabedaten berücksichtigt werden müssen, um eine Prozedur aus einem Klimamodell erfolgreich zu isolieren (Abschnitte 8.2.5 und 8.2.6).

8.2.1 Grundlegendes Vorgehen

Um per C&R einen Unittest für eine einzelne zu testende Prozedur (PUT) zu erstellen, muss diese in der Capturephase eingebettet in die Originalanwendung ausgeführt werden. Da Klimamodelle je nach Konfiguration verschiedene Ausführungspfade nehmen können, muss eine Konfiguration gewählt werden, in der die PUT in jedem Fall ausgeführt wird. Da jede Prozedur einer Anwendung in der Regel mehrfach ausgeführt wird, muss zudem festgelegt werden, bei welcher Ausführung das Capture stattfinden soll. Zu Beginn der gewählten Ausführung, noch bevor eine einzige Anweisung

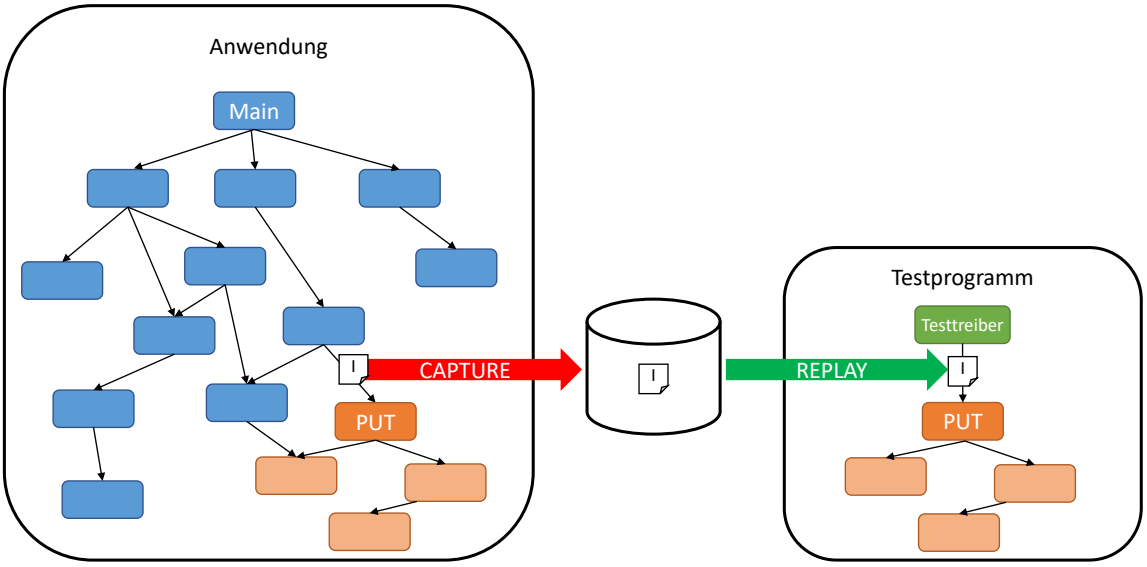


Abbildung 8.2: Grundprinzip des Capture-&-Replay-Verfahrens.
I = Eingabedaten (input)

der PUT ausgeführt wird, müssen die Eingabedaten der Prozedur aufgezeichnet, d.h. z.B. im Dateisystem abgelegt, werden. Zu den Eingabedaten gehören sämtliche Daten, die Einfluss auf das Verhalten der PUT haben. Welche Klassen von Daten dabei eine Rolle spielen, wird in Abschnitt 8.2.5 diskutiert. Vereinfachend gehen wir zunächst davon aus, dass die Originalanwendung und die PUT seriell ablaufen, die PUT deterministisch ist und sämtliche Eingabedaten daher zu Beginn der Ausführung zur Verfügung stehen. Ausnahmen von dieser Annahme werden ebenfalls zu einem späteren Zeitpunkt diskutiert. In der Replayphase können die Eingabedaten nun von einem Testtreiber geladen werden, um mit ihnen die PUT unabhängig von der Originalanwendung auszuführen.

Wie in Abbildung 8.2 zu sehen, kann auf diese Weise die PUT aus der Originalanwendung herausgelöst und unabhängig von ihr ausgeführt werden. Die Abbildung zeigt auf der linken Seite den Aufrufgraph der Prozeduren der Originalanwendung. Blau dargestellt sind die Prozeduren, die für einen Test der PUT nicht benötigt werden. Orange sind die PUT und die von ihr direkt und indirekt aufgerufenen Prozeduren, im Folgenden *Unterprozeduren* genannt. Für das Testprogramm werden ein Testtreiber benötigt, der das Laden der Daten übernimmt und die PUT aufruft, die PUT selbst und ihre Unterprozeduren sowie andere Abhängigkeiten wie globale Variablen oder Konstanten. Zusätzlich kann ein automatisches Testorakel, das prüft, ob ein Test erfolgreich war oder nicht, integriert werden.

8.2.2 Testorakel

Verschiedene Formen von Testorakeln bieten sich an, um mit Hilfe von C&R erstellte Tests auszuwerten. In einigen Fällen wird das implizite Orakel (das Testprogramm lässt sich kompilieren und ausführen) ausreichen. Die Ausgabedaten der PUT könnten darüber hinaus manuell durch die EntwicklerIn geprüft werden, ggf. mit Hilfe von Diagrammen. Zudem könnten in dem Testprogramm beliebige Prüfungen der Ausgabedaten implementiert werden.

Zudem lassen sich Referenz-Ausgabedaten auf die gleiche Art und Weise erzeugen wie die Eingabedaten (Abbildung 8.3). Hierzu müssen in der Capturephase am Ende der Ausführung der PUT ihre Ausgabedaten aufgezeichnet werden, genauso wie zu Beginn die Eingabedaten. Diese Referenz-Ausgabedaten können zunächst dazu verwendet werden, um zu überprüfen, ob die Isolierung der PUT erfolgreich war und sie sich im Test genauso verhält, wie in der Originalanwendung. Unter der Annahme, dass die PUT deterministisch ist, bedeuten Abweichungen zwischen den Ergebnissen, dass nicht alle relevanten Eingabedaten erfasst wurden. Darüber hinaus können diese Referenz-Ausgabedaten als Testorakel für Regressionstests dienen, solange die zu testenden Änderungen nichtfunktionaler Natur sind, d.h. keinen Einfluss auf die Ergebnisse der PUT haben sollen.

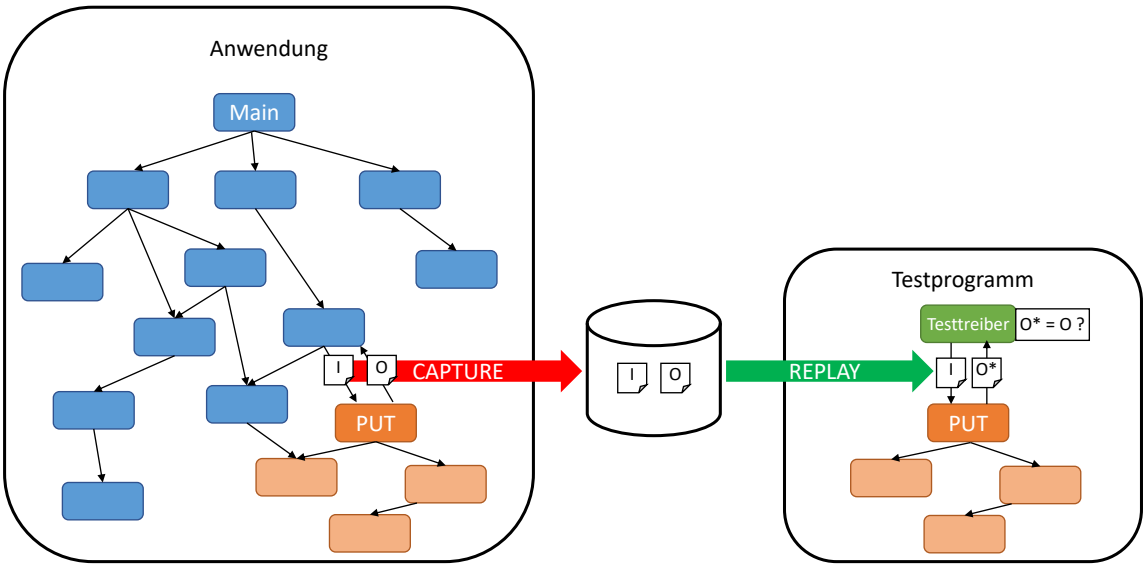


Abbildung 8.3: Capture-&-Replay mit Testorakel.
I = Eingabedaten (input), O = Ausgabedaten (output) in Originalanwendung, O* = Ausgabedaten im Test
Der rückwärtsgerichtete Pfeil von der PUT zur aufrufenden Prozedur stellt hier keine Aufrufbeziehung dar, sondern den Datenfluss, welcher sonst nur einseitig dargestellt ist.

8.2.3 Abgeleitete Testdaten

Per C&R lassen sich einzelne konsistente Testdatensätze erstellen, mit denen die PUT unabhängig von der Originalanwendung ausgeführt werden kann. Jeder so gewonnene Testdatensatz entspricht einem spezifischen Testfall. Diese Testfälle entsprechen Situationen, welche durch die Ausführung der Originalanwendung erzeugt werden. Bestimmte Grenz- oder Sonderfälle lassen sich so ggf. nur schwer reproduzieren. Allerdings können die per C&R gewonnen Testdatensätze von der EntwicklerIn manipuliert werden, um derartige Fälle gezielt zu erzeugen. Dies kann entweder durch direkte Bearbeitung der gespeicherten Daten (z.B. der Dateien) erfolgen oder programmatisch im Testprogramm nach dem Laden der Daten.

So können aus einem aufgezeichneten Datensatz mit relativ geringem Aufwand mehrere Testfälle erzeugt werden (Abbildung 8.4). Die Veränderungen der Daten müssen dabei sehr sorgfältig geschehen, um die Konsistenz der Datensätze nicht zu zerstören. So sollten beispielsweise Gitterstrukturen erhalten bleiben.

Referenzdaten für Testorakel könnten in diesem Fall auch manuell angepasst werden. Als Alternative hierzu bietet es sich an, das Testprogramm mit den abgeleiteten Eingabedaten zusammen mit der Referenzversion der PUT auszuführen, um so die abgeleiteten Referenz-Ausgabedaten zu erhalten (Abbildung 8.5).

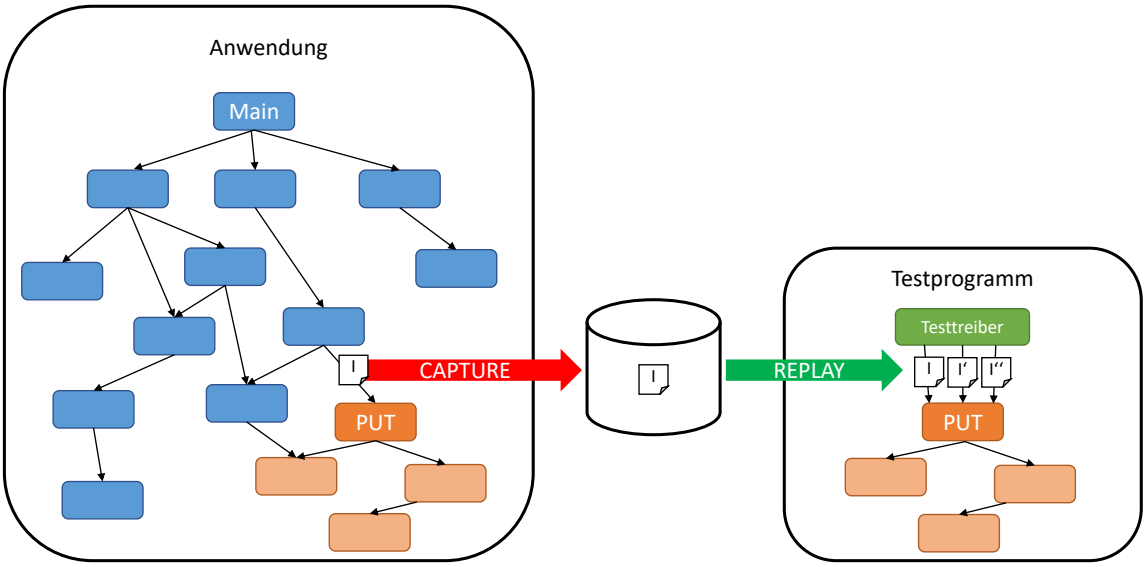


Abbildung 8.4: Capture-&-Replay mit abgeleiteten Testfällen.
I = Eingabedaten (Input), I', I'' = abgeleitete Eingabedaten

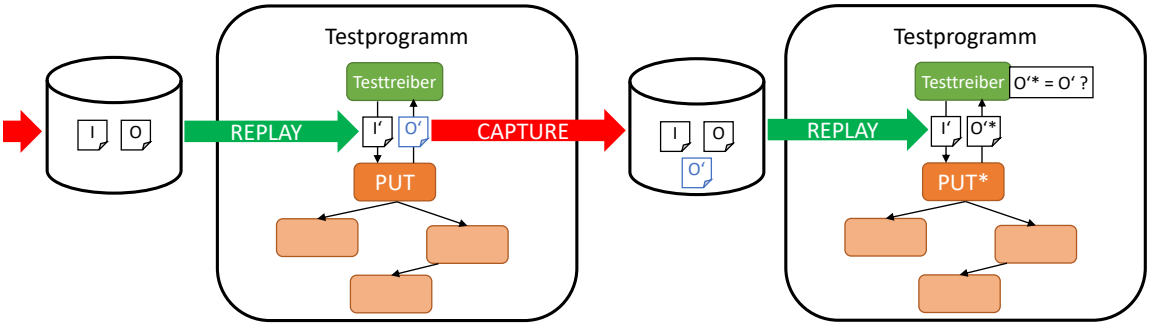


Abbildung 8.5: Capture-&-Replay mit abgeleiteten Referenzdaten
PUT* = geänderte PUT, I' = abgeleitete Eingabedaten, O' = abgeleitete Ausgabedaten, O* = abgeleitete Ausgabedaten der geänderte PUT

8.2.4 Testdatenauswahl

Im Gegensatz zu einigen der in Kapitel 7 genannten Verfahren, welche zum Teil basierend auf einem vorgegebenen Testziel (z.B. gute Codeabdeckung) völlig selbstständig Testfälle erzeugen, lässt sich mit Hilfe von C&R zunächst nur die Datenbeschaffung selbst automatisieren. Die Auswahl der Daten und damit der Testfälle verbleibt in der Verantwortung der EntwicklerIn. Dies deckt sich jedoch mit dem in Kapitel 5 formuliertem Ziel dieser Arbeit, vor allem den Aufwand für die Erstellung der Daten zu reduzieren.

Dennoch bleibt es die Aufgabe der EntwicklerIn, geeignete Testdatensätze auszuwählen. Im Fall von Klimamodellen lässt sich diese Entscheidung auf zwei Fragen herunterbrechen:

1. In welcher Konfiguration wird das Klimamodell in der Capturephase ausgeführt?
2. Zu welchem Zeitpunkt, d.h. vor welcher Ausführung der PUT, werden die Testdaten aufgezeichnet?

Wichtigstes Kriterium bei der Auswahl der Modellkonfiguration ist, dass durch diese die PUT ausgeführt wird. In den Konfigurationen von Klimamodellen lassen sich verschiedene physikalische Prozesse an- und ausschalten oder für unterschiedlichen numerischen Abbildungen eines physikalischen Prozesses eine auswählen. Gehört die PUT zu solch einer aktivier- oder deaktivierbaren Komponente, muss diese beim Capture aktiviert sein. Darüber hinaus lassen sich in den Modellkonfigurationen Gitterauflösung und Anzahl der parallelen Rechnerprozesse festlegen. Grundsätzlich sollten diese so grob bzw. so klein wie möglich gewählt werden. Dies spart nicht nur Ressourcen in Form von Speicherplatz für die aufgezeichneten Daten und verringert die Laufzeit des Unittests, sondern erhöht unter Umständen auch die Verständlichkeit

des Tests. Auch hier sollte jedoch auf die Aussagekraft des Tests geachtet werden. Sollten beispielsweise gewisse Problemfälle, welche zu testen sind, erst bei höherer Auflösung auftreten, ist dies bei der Konfiguration zu berücksichtigen.

Die Auswahl des Aufzeichnungszeitpunkts erscheint ungleich schwerer, da in vielen Fällen kaum vorhersehbar ist, welche Form die Eingabedaten zu welchem Zeitpunkt annehmen. Um eine lauffähige Version der PUT unabhängig von der Originalanwendung zu erstellen, dürfte in vielen Fällen jedoch eine pragmatische Auswahl möglich sein. Die einfachste Wahl bestünde beispielsweise in der ersten Ausführung der PUT. Eine andere Möglichkeit wäre es, eine bestimmte Anzahl i von Ausführungen abzuwarten, zum Beispiel bis eine Spin-Up-Phase beendet ist. Prozeduren, die einen physikalischen Prozess in einem Klimamodell implementieren, werden meist einmal pro Zeitschritt oder einmal alle x Zeitschritte ausgeführt. Die i-te Ausführung entspricht somit der Ausführung der PUT im $i * x$-ten Zeitschritt. Helfer- oder Infrastrukturprozeduren werden ggf. auch mehrmals in einem Zeitschritt ausgeführt, was die Berechnung von i erschwert. Wie in (Kapitel 4) gezeigt, sind selbst Unittests, welche lediglich Arrays mit Nullen an die PUT übergeben, in der Lage eine Vielzahl von Fehlern aufzudecken. Darüber hinaus wurde gezeigt, dass bei E2E-Tests häufig einer oder wenige Zeitschritte ausreichen, um Regressionstests durchzuführen, da sich Abweichungen in den Ergebnissen i.d.R. recht schnell bemerkbar machen. Daher ist anzunehmen, dass in vielen Fällen mit solch pragmatisch gewählten Datensätzen nützliche Testfälle erzeugt werden können, aus denen darüber hinaus weitere abgeleitet werden können.

8.2.5 Eingabedaten

Für die erfolgreiche Reproduktion einer PUT-Ausführung mittels C&R ist es notwendig, sämtliche Eingabedaten aufzuzeichnen und bereitzustellen, die Einfluss auf das Verhalten der PUT haben. Vereinfacht gesprochen entspricht dies der Menge der Daten, die außerhalb der PUT und ihrer Unterprozeduren erzeugt werden, jedoch innerhalb dieser gelesen werden. Grundsätzlich sollte die Menge der aufgezeichneten Eingabedaten immer so klein wie möglich gehalten, d.h. keine unnötigen Daten enthalten. Dies spart zum einen Ressourcen in Form von Speicherplatz und verringert die Ladezeit der Daten im Unittest. Zum anderen erhöht dies die Verständlichkeit des Tests, wenn dieser nur Daten lädt, die von der PUT benötigt werden. Allerdings ist hier auch Pragmatismus geboten, d.h. bei Unsicherheiten ob des Einflusses einzelner Daten sollte lieber etwas zu viel als zu wenig aufgezeichnet werden, um die korrekte Reproduktion der PUT außerhalb der Originalanwendung zu gewährleisten.

Es lassen sich verschiedene Formen von Eingabedaten unterscheiden. In Fortran-Systemen wie Klimamodellen sind dies:

1. Prozedur-Argumente
2. SAVE-Variablen
3. Globale Variablen
4. Parameter
5. Zustände externer Bibliotheken
6. Dateiinhalte
7. Von parallelen Prozessen empfangene Daten
8. Von Drittsystemen empfangene Daten

Während die Eingabedaten der Formen 1–4 zu Beginn einer Prozedur feststehen und aufgezeichnet werden können, können die Daten der Formen 5–8 von einer Prozedur zu einem beliebigen Zeitpunkt ihrer Ausführung gelesen bzw. empfangen werden. Im Folgenden werden alle Formen und ihre Bedeutung für das C&R-Verfahren diskutiert.

Prozedur-Argumente

Wie in anderen prozeduralen Sprachen können auch in Fortran Prozeduren beim Aufruf Daten in Form von Argumenten übergeben werden. In anderen Sprachen werden diese auch Parameter genannt (siehe auch Abschnitt 2.4.1). Argumente können als In-, Inout- und Out-Argumente definiert werden, also als nur gelesene, gelesen und geschriebene oder nur geschriebene. Allerdings dient diese Festlegung im Wesentlichen der Dokumentation der beabsichtigten Rolle der Argumente. So verhindern Fortran-Compiler i.d.R. nicht, dass Out-Argumente in einer Prozedur gelesen werden. Grundsätzlich kann man also davon ausgehen, dass sämtliche Argumente einer Prozedur zu ihren Eingabedaten gehören.

Einschränken ließe sich diese Menge, indem nur diese Argumente aufgezeichnet werden, die von der Prozedur tatsächlich gelesen werden. Dies gilt insbesondere auch für Argumente mit benutzerdefiniertem Verbunddatentyp bzw. deren Komponenten. Im Klimamodell ICON gibt es beispielsweise Verbunddatentypen mit über 200 Komponenten. In einzelnen Prozeduren, die Argumente dieser Typen übergeben bekommen, werden jeweils nur ein Bruchteil dieser Komponenten verwendet. Hier lässt sich die Menge der beim C&R aufzuzeichnenden Eingabedaten merklich verkleinern, indem diese bei Verbundtypen auf jene Komponenten beschränkt wird, die tatsächlich von der PUT verwendet werden.

SAVE-Variablen

SAVE-Variablen sind lokale Variablen, deren Werte aufrufübergreifend erhalten bleiben (siehe Abschnitt 2.4.1). Da diese die Ergebnisse einer Prozedur beeinflussen können, sind auch sie als Eingabedaten anzusehen. Als lokale Variablen sind sie jedoch nicht von außerhalb der Prozedur zugreifbar, was die Möglichkeiten die Werte im Rahmen des C&R gezielt zu setzen, einschränkt.

Globale Variablen

Globale Variablen sind prozedurübergreifend definierte Variablen, die auch ohne als Argument übergeben zu werden, von Prozeduren verwendet werden können. In Fortran gibt es zwei Formen globaler Variablen: Modulvariablen (siehe auch Abschnitt 2.4.2) und Variablen, welche in Common Blocks (siehe auch Abschnitt 2.4.11) definiert werden.

Modulvariablen können privat oder öffentlich sein, d.h. entweder innerhalb des eigenen Moduls verwendet werden oder auch in anderen Modulen. Beim C&R müssen sämtlichen Modulvariablen berücksichtigt werden, welche in der PUT oder einer ihrer Unterprozeduren verwendet werden. Dies können auch private Variablen der Module der Unterprozeduren sein, welche für die PUT selbst nicht zugreifbar sind. Wie bei Argumenten kann bei Modulvariablen, die einen Verbunddatentyp haben, das Capture auf die tatsächlich benötigten Komponenten beschränkt werden.

Für Variablen in Common Blocks gilt beim C&R im Wesentlichen das gleiche wie für Modulvariablen. Da im Kopf der Prozeduren jedoch explizit angeben werden muss, welche Variablen aus welchen Common Blocks verwendet werden (sollen), ist leichter zu identifizieren, welche Variablen zumindest potenziell verwendet werden, als dies bei Modulvariablen der Fall ist. Seit der Einführung von Modulen in Fortran 90 gilt die Verwendung von Common Blocks als schlechte Praxis und wird nur noch selten eingesetzt, dennoch gehören sie weiterhin zum Fortran-Standard und können benutzt werden. Darüber hinaus enthalten viele Klimamodelle alte Codeteile, die ursprünglich in Fortran77 geschrieben wurden und daher ggf. auch Common Blocks verwenden. Grundsätzlich müssen Common Blocks somit beim C&R berücksichtigt werden.

Parameter

Konstanten, also „Variablen“ mit unveränderlichem Wert, in Fortran Parameter genannt (siehe auch Abschnitt 2.4.7), können sowohl auf Modul- als auch auf Proze-

durebene definiert werden. Ihr Wert wird statisch im Quelltext festgelegt. Die Werte von Konstanten müssen beim C&R nicht aufgezeichnet werden, da sie im Test genauso statischer Teil des Programms sind wie in der Originalanwendung. Die Module, in denen von der PUT verwendeten Konstanten definiert sind, müssen beim Kompilieren des PUT-Moduls im Testkontext ebenso zur Verfügung stehen wie alle anderen Abhängigkeiten auch.

Bibliotheken

Klimamodelle verwenden für einige Aufgaben häufig externe Bibliotheken, insbesondere im Infrastrukturbereich. Diese können ebenso in Fortran oder in anderen Programmiersprachen, wie etwa C oder C++, implementiert sein. Beispielsweise wird zur Benutzung von MPI eine Implementation wie OpenMPI oder MPICH benötigt (siehe auch Abschnitt 2.5.2), auch die NetCDF-Bibliothek (siehe auch Abschnitt 2.5.5) wird von vielen Modellen benutzt. Werden innerhalb einer PUT Prozeduren einer Bibliothek aufgerufen oder Variablen verwendet, die von einer Bibliothek bereitgestellt werden, beeinflusst der innere Zustand der Bibliothek ggf. das Verhalten der PUT. Die Zustandsvariablen einer Bibliothek sind beim C&R daher grundsätzlich genauso zu behandeln wie die Modulvariablen anderer verwendeter Module. Problematisch in diesem Zusammenhang könnte jedoch zum einen sein, dass bei externen Bibliotheken, wenn diese nur im Binärformat vorliegen, nicht so leicht herauszufinden ist, welche Variablen es gibt und welche das Verhalten der PUT beeinflussen. Zum anderen ist der direkte Zugriff auf diese Variablen unter Umständen nicht möglich. In diesem Fall muss sowohl beim Capture ggf. über die öffentliche Schnittstelle der Bibliothek versucht werden, den Zustand abzufragen bzw. beim Replay über diesen Weg der Zustand wiederhergestellt werden.

Parallele Prozesse

Klimamodelle werden i.d.R. parallelisiert ausgeführt, verteilt auf mehrere Prozesse bzw. Rechnerknoten. Als Programmiermodelle kommen hierzu MPI und OpenMP zum Einsatz (siehe auch Abschnitte 2.5.2 und 2.5.3). Daten, die während der Ausführung einer Prozedur auf einem Prozess von anderen Prozessen empfangen werden, beeinflussen maßgeblich das Verhalten der Prozedur. Hierbei handelt es sich um Eingabedaten, die nicht bereits zu Beginn einer Prozedur feststehen, sondern erst während der Ausführung empfangen werden.

Die folgenden Ausführungen gehen zunächst von folgenden Vereinfachungen für MPI-Kommunikation aus:

1. Eine PUT kommuniziert nur mit sich selbst, d.h. korrespondierende Sende- und Empfangsoperationen befinden sich immer gemeinsam im Code der PUT und durch Fallunterscheidungen führen unterschiedliche Prozesse entweder das eine oder das andere aus.

   Beispiel 8.1: Fallunterscheidung nach Prozessnummer

   ```
   1 IF (MY_PROCESS == 0) THEN
   2   CALL MPI_Send(...)
   3 ELSE
   4   CALL MPI_Recv(...)
   5 END IF
   ```

   Mit anderen Worten: Es ist nicht der Fall, dass eine Empfangsoperation in einer anderen Prozedur aufgerufen wird als die korrespondierende Sendeoperation. Ebenso sind auch korrespondierende globale Kommunikationsoperationen wie etwa `MPI_Bcast` nicht über mehrere Prozeduren verteilt.

2. Die i-te Ausführung einer PUT auf einem Prozess kommuniziert nur mit der i-ten Ausführung der PUT auf einem anderen Prozess.

Sind diese Voraussetzungen gegeben, kann die MPI-Kommunikation zwischen den Prozessen beim C&R vernachlässigt werden. Wie in Abbildung 8.6 dargestellt, müssen dazu beim Capture sämtliche übrigen Eingabedaten der i-ten Ausführung der PUT für jeden parallelen Prozess aufgezeichnet und das Replay ebenfalls parallel mit derselben Anzahl Prozesse, in derselben MPI-Konfiguration, durchgeführt werden wie das Capture. Dann werden beim Replay dieselben Daten (symbolisiert durch die lilafarbenen Verbindungen) zwischen den Prozessen ausgetauscht wie beim Capture. Bei diesem Vorgehen multipliziert sich die Menge der aufzuzeichnenden Daten um die Anzahl der verwendeten Prozesse. Es sollte daher immer mit so wenig Prozessen wie möglich durchgeführt werden, was beim Regressionstesten aber ohnehin gilt, um Ressourcen zu sparen und keine unnötige Komplexität hinzuzufügen.

Weitere Voraussetzung für eine korrekte Reproduktion ist auch bei diesem Vorgehen, dass die PUT deterministisch ist, es also beispielsweise keine *Race Conditions* o.ä. gibt. Determinismus ist jedoch ohnehin Grundvoraussetzung für das C&R-Prinzip, zumindest wenn eine bitidentische Reproduktion angestrebt wird. In einigen Fällen kann es ggf. auch möglich sein, das Replay mit weniger Prozessen als das Capture durchzuführen. Jedoch ist es dann Aufgabe der EntwicklerIn abzuschätzen, ob dies Einfluss auf die Ergebnisse haben könnte.

Für die Parallelisierung mit Hilfe von OpenMP gilt zunächst folgende Vereinfachung: Die PUT darf OpenMP-parallelisierte Blöcke enthalten, jedoch selbst nicht innerhalb

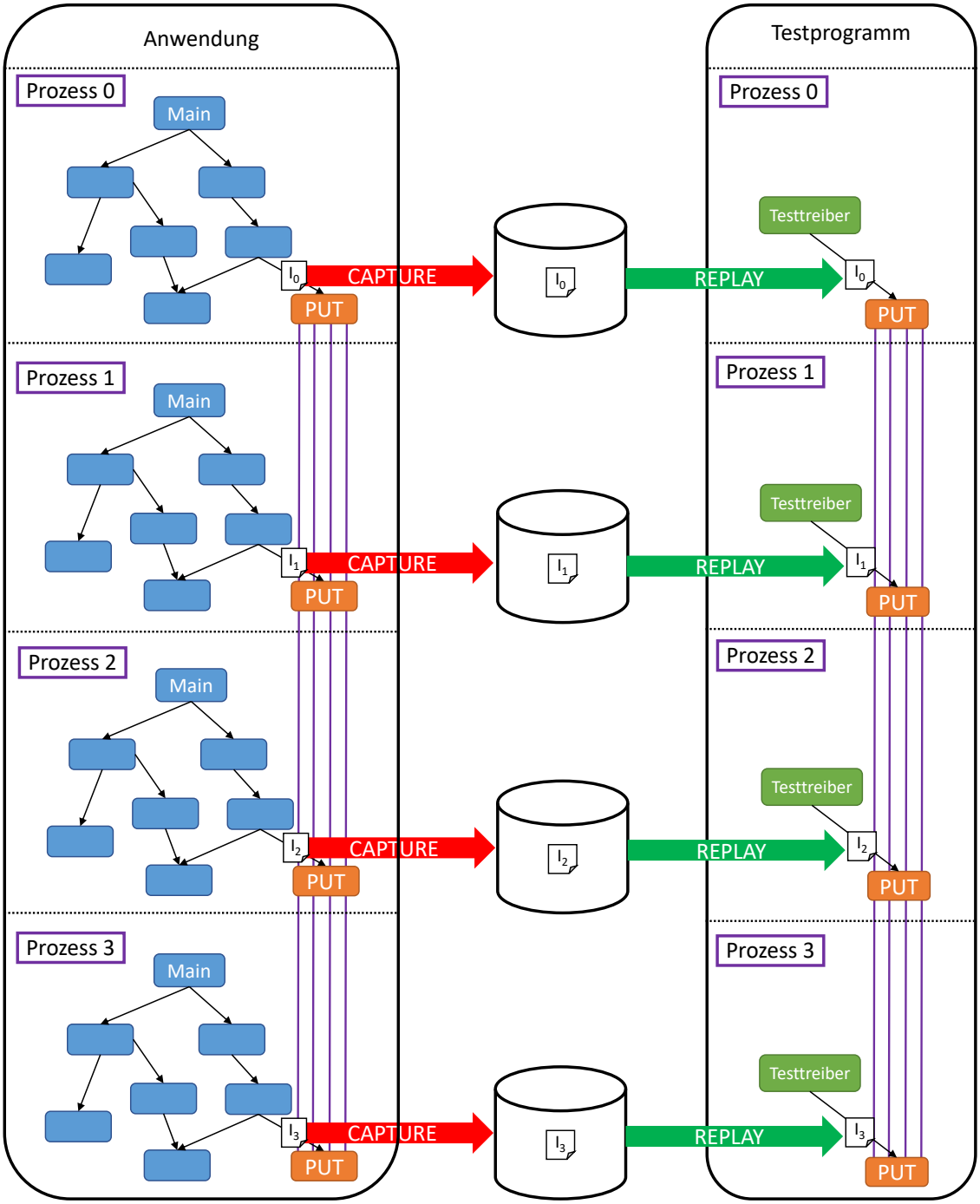


Abbildung 8.6: Capture & Replay mit vier parallelen Prozessen.
Die lilafarbenen Linien repräsentieren die Kommunikation zwischen den Prozessen innerhalb der PUT, welche beim Replay genauso abläuft wie beim Capture. Unterprozeduren der PUT sind hier aus Platzgründen nicht dargestellt, ebenso wie die Interprozesskommunikation außerhalb der PUT.

eines OpenMP-parallelisierten Blocks aufgerufen werden. OpenMP-Blöcke innerhalb der PUT sind beim C&R unproblematisch, da diese beim Replay genauso ablaufen wie beim Capture, sofern alle anderen Eingabedaten richtig berücksichtigt wurden. Grundsätzlich ließe sich auch die Ausführung einer PUT innerhalb eines OpenMP-Blocks per C&R reproduzieren, jedoch wäre es dabei ungleich schwerer zu steuern, welche konkrete Ausführung der PUT aufgezeichnet und reproduziert wird.

Die meisten Prozeduren in den untersuchten Klimamodellen erfüllen die genannten Voraussetzungen für MPI- und OpenMP-Parallelisierung. Abschnitt 10.3 beschäftigt sich mit Ausnahmen von dieser Regel und möglichen Vorgehensweisen mit diesen umzugehen.

Dateiinhalte

Das Verhalten einer PUT kann auch von Daten, welche innerhalb der Prozedur aus dem Dateisystem eingelesen werden, beeinflusst werden. In Klimamodellen geschieht das Einlesen von Daten im Wesentlichen zu Beginn einer Simulation in der Initialisierungsphase. Doch auch während einer Simulation müssen ggf. Antriebskraftdaten aus Dateien eingelesen werden. In beiden Fällen sind normalerweise spezielle Infrastrukturkomponenten für das Einlesen der Daten zuständig. Soll nicht gerade eine dieser Infrastrukturprozeduren getestet werden, können Dateiinhalte als Eingabedaten somit vernachlässigt werden. Ansonsten kann man davon ausgehen, dass dieselben einzulesenden Dateien beim Replay zur Verfügung gestellt werden können wie beim Capture.

Drittsysteme

Drittsysteme wie etwa Datenbanken werden von Klimamodellen eher selten verwendet. In gekoppelten Modellen kommt es vor, dass einzelne Submodelle miteinander Daten austauschen müssen, beispielsweise das Atmosphärenmodell mit dem Ozeanmodell. In der Regel werden diese Teilmodelle jedoch zu einer Anwendung zusammengefügt. Der Datenaustausch findet dann systemintern über einen sogenannten Koppler statt, welcher selbst eine Komponente des Modells, ggf. in Form einer Bibliothek, ist. Es gibt jedoch auch Architekturen, in denen Submodelle sowie Koppler separate Anwendungen sind, die parallel zueinander ausgeführt werden.

Sollte es der Fall sein, dass eine PUT mit einem Drittsystem kommuniziert, müsste dieses beim Replay ebenso zur Verfügung stehen wie beim Capture. Alternativ könnten die vom Drittsystem gesendeten Daten ebenso aufgezeichnet werden und über ein *Testdouble* (siehe auch Abschnitt 3.3) beim Replay zur Verfügung gestellt

werden. Die automatische Erstellung von Testdoubles wird in dieser Arbeit nicht behandelt.

8.2.6 Ausgabedaten

Welche Ausgabedaten zur Überprüfung eines Testlaufs herangezogen werden liegt grundsätzlich im Ermessen der EntwicklerIn. Dabei spielt eine Rolle, welches Ziel mit dem jeweiligen Test verfolgt wird. Soll eine vollständige Validierung vorgenommen werden, müssen analog zu den Eingabedaten alle relevanten Ausgabedaten identifiziert und im Test reproduziert werden. Grundsätzlich unterteilen sich die Ausgabedaten in dieselben Formen wie die Eingabedaten. Statt der eingelesenen oder empfangenen Daten sind es hier die jeweils geschriebenen bzw. gesendeten Daten. Gegebenenfalls müssen auch Daten, die nicht verändert werden (sollen), zu den Ausgabedaten gezählt werden, wenn gerade deren Unveränderlichkeit getestet werden soll. Folgende Besonderheiten sind zudem bei den Ausgabedaten zu berücksichtigen:

- Handelt es sich bei der PUT um eine FUNKTION, gehört zusätzlich der Rückgabewert der FUNKTION zu den Ausgabedaten.
- Aliase erschweren die Identifikation und Reproduktion von Ausgabedaten. Dies ist in Fortran insbesondere im Zusammenhang mit Zeigervariablen und ARGUMENTEN, welche grundsätzlich per Referenz übergeben werden, relevant. Verweisen beispielsweise eine Variable a und eine Variable b auf dieselbe Speicherstelle und der Wert der Variable a wird innerhalb der PUT oder einer ihrer Unterprozeduren verändert, ändert sich ebenso der Wert der Variable b, ohne dass dies im Quelltext erkennbar ist. Soll jedoch im Rahmen eines Tests sowohl der Wert von a als auch der von b validiert werden, muss dafür Sorge getragen werden, dass auch im Test a und b auf dieselbe Speicherstelle verweisen. In diesem Zusammenhang ist auch zu berücksichtigen, dass Fortran-Compiler zwar normalerweise verhindern, dass reine IN-ARGUMENTE direkt verändert werden, über Aliase dies aber dennoch möglich ist.
- Werden in der PUT Dateien verändert oder Daten an Drittsysteme gesendet, sollten im Test hierfür Kopien bzw. spezielle Testsysteme verwendet werden, um keine ungewollten Veränderungen an Produktivdaten vorzunehmen.

8.3 Softwareunterstützung

Im vorangegangenen Abschnitt wurde gezeigt, wie C&R dazu genutzt werden kann, Unittests für einzelne Prozeduren in Klimamodellen zu erstellen. Um zu einer Aufwandsreduzierung bei der Erstellung von Unittests zu kommen, müssen wesentliche

Teile des C&R-Prozesses in Form eines Softwarewerkzeugs automatisiert wurden. Daraus ergeben sich die folgenden Fragen, welche in diesem Abschnitt diskutiert werden:

- Aus welchen Arbeitsschritten besteht der C&R-Prozess? (Abschnitt 8.3.1)
- Welche dieser Arbeitsschritte sollten im Sinne der Aufwandsreduktion und des anwendungsorientierten Ansatzes automatisiert werden, welche dagegen manuell durchgeführt? (Abschnitt 8.3.2)
- Welche alternativen Verfahren stehen für die Automatisierung zur Verfügung? (Abschnitte 8.3.3–8.3.5)
- In welchem Rahmen lässt sich der C&R -Prozess an individuelle Anforderungen der BenutzerInnen[12] anpassen? (Abschnitte 8.3.6 und 8.3.7)
- Was ist bei der Gestaltung der Benutzungsschnittstelle zu beachten? (Abschnitt 8.3.8)

Neben Gestaltungsalternativen wird jeweils auch die in dieser Arbeit getroffene Wahl benannt und begründet. Eine konkrete Umsetzung eines entsprechenden Softwarewerkzeugs wird im nachfolgenden (Kapitel 9) vorgestellt.

8.3.1 Arbeitsschritte

Ein C&R-Prozess zur Erstellung eines Unittests beinhaltet die folgenden Schritte:

1. Bestimmen der PUT
2. Identifizieren der Eingabedaten
3. *Identifizieren der Ausgabedaten*
4. Bestimmen des Capturekontexts (Konfiguration der Originalanwendung)
5. Bestimmen des Capturezeitpunkts (aufzuzeichnende Ausführung der PUT)
6. Vorbereiten des Capture
7. **Capture: (Kompilieren und) Ausführen der Originalanwendung**
8. Erstellen eines Testtreibers
9. Bestimmen eines Testorakels

[12] Mit „BenutzerIn“ ist hier und im Folgenden die BenutzerIn eines Softwarewerkzeugs gemeint, dass den C&R-Prozess unterstützt, also die EntwicklerIn bzw. TesterIn einer zu testenden Anwendung.

10. *Erstellen eines automatischen Testorakels*
11. **Replay: (Kompilieren und) Ausführen des Testprogramms**
12. Bewerten der Tests

Schritt 1 ist eher impliziter Natur. Der Wunsch eine bestimmte Prozedur zu testen ist eher Auslöser des Prozesses als das das Bestimmen der Prozedur Teil desselben wäre. Einem Softwarewerkzeug, das diesen Prozess unterstützt, muss die PUT aber in geeigneter Weise übermittelt werden, daher ist dieser Schritt hier mit aufgeführt. Auch Schritt 9 ist eher implizit und manifestiert sich ggf. durch Schritt 10. Das Testorakel kann sich auch von Testlauf zu Testlauf ändern. Ggf. reicht in frühen Versionen das implizite Orakel (Test kompiliert und lässt sich ausführen) aus, während später eine automatische Validierung von Ausgabedaten hinzugefügt wird.

Die zu Schritt 6 gehörenden Tätigkeiten hängen stark von der gewählten Aufzeichnungsstrategie ab. Diese wird in Abschnitt 8.3.4 diskutiert.

Die kursiv hervorgehobenen Schritte sind optional. Die Erstellung eines automatischen Orakels (Schritt 10) ist nicht notwendig, wenn im jeweiligen Anwendungsfall das implizite oder ein menschliches Orakel ausreicht. Ist dies der Fall kann ggf. auch auf das Identifizieren der Ausgabedaten (Schritt 3) verzichtet werden. Auch wenn anstelle der Funktionalität andere Aspekte getestet werden sollen (z.B. Geschwindigkeit) oder der Test zum Debuggen verwendet wird, werden die Ausgabedaten u.U. nicht benötigt. Jedoch wird auch in diesen Fällen die EntwicklerIn meist zunächst überprüfen wollen, ob die Reproduktion der PUT-Ausführung sich im Test identisch zu der Ausführung in der Originalanwendung verhält. Dafür sind die Ausgabedaten wiederum nötig.

In Schritt 8 und 11 sowie im Folgenden wird zwischen den Begriffen *Testtreiber* und *Testprogramm* unterschieden. Mit Testtreiber wird der Teil des Testprogramms verstanden, der für das Laden der Eingabedaten sowie das Ausführen der PUT zuständig ist. Das Testprogramm ist die gesamte ausführbare Einheit, die ggf. auch das automatische Testorakel enthält. Diese Unterscheidung entspricht nicht exakt der allgemeinen Verwendung in der Literatur (siehe z.B. Spillner und Linz, 2012, S. 216), erscheint für diese Arbeit jedoch zweckmäßig.

Die Schritte beschreiben die erstmalige Erstellung und Ausführung eines Unittests per C&R. Die Durchführung der Schritte muss dabei nicht streng in der aufgeführten Reihenfolge geschehen. Dieser Aspekt wird in Abschnitt 8.3.6 detaillierter diskutiert. Nach der ersten Ausführung folgt ein Zyklus, in der Änderungen an der PUT und ggf. dem Testprogramm vorgenommen werden und erneut die Schritte 11 und 12 durchgeführt werden.

8.3.2 Automatisierung

Ein Softwarewerkzeug, das den oben beschriebenen Prozess unterstützt, sollte einerseits die aufwendigsten Schritte automatisieren und andererseits der BenutzerIn, d.h. der EntwicklerIn, so viel Entscheidungsfreiheit wie möglich einräumen. Die aufwendigsten Schritte des Prozesses sind:

Schritt 2 Wie in Abschnitt 8.2.5 gezeigt, bestehen die Eingabedaten einer Prozedur nicht nur aus der Liste ihrer DUMMYARGUMENTE. Um die vollständige Menge der Eingabedaten zu erhalten, müssen nicht nur die PUT, sondern auch sämtliche Unterprozeduren analysiert werden.

Schritt 3 Geht man davon aus, dass sämtliche Ausgabedaten zur Validierung der Reproduktion herangezogen werden sollen, müssen diese gleichfalls durch Analyse der PUT und ihrer Unterprozeduren zusammengetragen werden.

Schritt 6 Welche Maßnahmen nötig sind, um das Capture vorzubereiten hängt maßgeblich von der verwendeten Capturestrategie ab. Verschiedene Strategien werden in Abschnitt 8.3.4 unter dem Aspekt der Anwendungsorientierung diskutiert. Wächst der für die Vorbereitung des Capture benötigte Aufwand mit der Menge der Eingabedaten, ist diese ebenfalls ein Kandidat für eine Automatisierung.

Schritt 8 Im Testtreiber müssen u.a. sämtliche aufgezeichneten Daten geladen und die Variablen der Eingabedaten korrekt belegt werden. Je nach Replaystrategie (siehe Abschnitt 8.3.4) ist hierzu u.U. sehr viel Code nötig, der mit der Menge der Eingabedaten wächst. Diesen zu erstellen kann nicht nur sehr aufwendig sein, sondern auch sehr fehleranfällig. Daher sollte auch dieser Schritt automatisiert werden.

Schritt 10 Auch der Aufwand für die Erstellung von automatischen Testorakeln hängt von der hierfür gewählten Strategie ab (siehe auch Abschnitt 8.3.5). Hier gilt ebenfalls: Wächst der Aufwand mit der Menge der Ausgabedaten, wäre eine Automatisierung angebracht. Sollen für das Testorakel Referenzdaten aus der Originalanwendung gewonnen werden, ist deren Beschaffung bereits Teil der Schritte 6 und 7.

Schritt 12 Der Aufwand für die Bewertung des Tests hängt maßgeblich von der Wahl des Testorakels ab. Wurde ein automatisches Orakel erstellt, ist der Aufwand gering. Ist ein menschliches Orakel nötig oder müssen Daten manuell verglichen werden, ist dieser hoch. Eine Automatisierung dieses Schritts ergibt sich somit aus der Durchführung des Schritts 10.

Die Schritte 1, 4, 5 und 9 definieren die wesentlichen Eigenschaften des zu erstellenden Tests und sollen daher in der Hand der EntwicklerIn bleiben. Bzgl. Schritt 1 ist diese Annahme trivial, da es, wie bereits diskutiert, das grundlegende Ziel des C&R-Prozesses ist, einen Unittest für eine von der EntwicklerIn vorgegebene PUT zu erstellen.

Zu Schritt 4 und 5 wäre es grundsätzlich denkbar, ein System zu erstellen, dass anhand definierter Testziele automatisch geeignete Konfigurationen und/oder Capturezeitpunkte sucht. Ähnlich wie bei einigen der in Kapitel 7 vorgestellten Ansätze würde so nicht nur die Test- und Testdatenerstellung automatisiert, sondern auch die Testfallauswahl. Dies übersteigt jedoch den Rahmen dieser Arbeit.

Bzgl. der Wahl des Testorakels (Schritt 9) wäre es denkbar, wenn das C&R-System ein definiertes Standardverhalten vorgibt, welches von der EntwicklerIn von Fall zu Fall geändert werden kann. Eine naheliegendes Standardvorgehen wäre hier beispielsweise die vollständige Validierung aller Ausgabedaten mit Hilfe aufgezeichneter Referenzdaten aus der Originalanwendung. Dies würde es ermöglichen, für jeden neu erstellten Test zunächst zu überprüfen, ob die Reproduktion erfolgreich ist.

Da Klimamodelle selbständig, d.h. ohne Benutzerinteraktionen, ablaufen, ist Schritt 7 ohnehin automatisiert, sofern die jeweilige Capturestrategie während der Aufzeichnung der Daten keine Benutzerinteraktion benötigt. Daraus folgt, dass auch die Ausführung des Tests (Schritt 11) automatisiert läuft.

8.3.3 Analysestrategien

Um anstatt eines vollständigen Speicherabzugs der Originalanwendung gezielt nur die benötigten Eingabedaten einer PUT aufzuzeichnen, muss bekannt sein, welche dies sind. Möchte eine EntwicklerIn nach dem Test-First-Prinzip einen Test für eine noch zu schreibende Prozedur erstellen, wird sie selbst eine Liste der (voraussichtlich) benötigten Daten erstellen müssen. Soll ein Test für eine bereits existierende Prozedur erstellt werden, ist es hingegen möglich, die Liste der Eingabedaten durch eine (automatisierte) Analyse der PUT und ihrer Unterprozeduren zu erstellen. Für die Analyse von Programmen oder Programmteilen stehen zwei grundlegende Strategien zur Verfügung: die dynamische und die statische Analyse.

Bei der *dynamischen Analyse* wird das zu analysierende Programm ausgeführt und sein Verhalten beobachtet. Im Fall der Eingabedatenanalyse würde dies bedeuten, die PUT in der Originalanwendung auszuführen und Speicherzugriffe, Dateisystemoperationen etc. zu protokollieren. Ein wesentlicher Nachteil der dynamischen Analyse ist, dass nur die zum Zwecke der Analyse ausgeführten Programmpfade berücksichtigt werden können. Das bedeutet, dass auch nur die Eingabedaten erfasst werden können, die in den analysierten Pfaden benötigt werden. Dies wäre zwar ausreichend,

wenn dies genau die zu testenden Programmpfade sind. Dann würden auch für einen Test ausschließlich diese Eingabedaten benötigt. Soll jedoch auf Basis eines aufgezeichneten Eingabedatensatzes weitere Testfälle abgeleitet werden, um andere Pfade zu testen (siehe auch Abschnitt 8.2.3), werden u.U. weitere Eingabedaten benötigt.

Bei der *statischen Analyse* wird auf Basis des Programmcodes auf das Verhalten des zu analysierenden Programms geschlossen. Für die Eingabedatenanalyse bedeutet dies, dass im Programmcode nach den benötigten Eingabedaten gesucht würde. Der Programmcode kann dafür in Quelltextform analysiert werden, als Binärcode, in einer vom Compiler erstellte Zwischenform oder in einer anderen transformierten Form. Vorteil dieses Vorgehens ist, dass anders als bei der dynamischen Analyse in einem Analysedurchgang die Eingabedaten unabhängig von Programmpfaden erfasst werden können. Nachteil der statischen Analyse ist, dass dynamische Programm- und Datenstrukturen durch eine statische Analyse nicht erfasst werden können. Dies betrifft zum Beispiel polymorphe Programmstrukturen, die etwa durch Funktionspointer oder Vererbung entstehen. Auch Dateipfade oder Arrayindizes, die erst zur Laufzeit feststehen, können so nicht erfasst werden. Allerdings dürfte die Anzahl an Dateioperationen in der Regel so überschaubar sein, dass diese auch manuell identifiziert werden könnten. Arrayindizes wären nützlich, um nur diese Bereiche von Arrays aufzuzeichnen, die tatsächlich benötigt werden. Allerdings würde auch dieses Vorgehen die Möglichkeit zur Ableitung von Testfällen, in denen andere Arraybereiche benötigt würden, einschränken.

Sollen für die Validierung der PUT-Reproduktion sämtliche Ausgabedaten aufgezeichnet werden, können auch diese für bestehende Prozeduren per Analyse identifiziert werden. Dabei gelten grundsätzlich dieselben Überlegungen wie hier für die Eingabedaten dargestellt.

Aufgrund der Unabhängigkeit von konkreten Ausführungspfaden wird für diese Arbeit eine statische Analyse verwendet, sowohl für die Identifikation der Ein- als auch für die Ausgabedaten. Die konkrete Implementation wird in Abschnitt 9.5 diskutiert. In wie fern die Nichtanalysierbarkeit polymorpher Programmstrukturen in der Praxis bei Klimamodelle überhaupt ein Problem darstellt, wird im Rahmen der Evaluation (Kapitel 11) zu untersuchen sein. Es bietet sich daher an, bei der Identifikation der Ausgabedaten nach der gleichen Methode vorzugehen wie bei den Eingabedaten.

8.3.4 Strategien zum Aufzeichnen und Laden von Ein- und Ausgabedaten

Analog zu den unterschiedlichen Checkpoint-/Restart-Strategien (siehe Abschnitt 8.1.3) kann auch das Aufzeichnen und Laden der Ein- und Ausgabedaten beim C&R entweder auf Ebene der Anwendung oder auf Ebene der Laufzeitumgebung geschehen. Bei der Laufzeitumgebung könnte es sich dabei um das Betriebssystem handeln oder

eine virtuelle Maschine, auf der die Originalanwendung oder der Test ausgeführt wird. Um die Aufzeichnung auf dieser Ebene zu realisieren, würde eine dritte Anwendung benötigt werden, die die Ausführung der Originalanwendung überwacht und zum gewünschten Zeitpunkt die gewünschten Daten aufzeichnet. Beim Replay müssten die Daten dann in das laufende Testprogramm injiziert werden. Die Funktionalität wäre ähnlich der eines Debuggers, da sie ebenso das Setzen von Haltepunkten sowie das Inspizieren und Modifizieren von Daten beinhaltet. Die Handhabbarkeit dieses Ansatzes hinge primär von der Handhabbarkeit der eingesetzten Anwendung ab. Modifikationen von Testdaten zur Ableitung weiterer Testfälle müssten bei diesem Vorgehen außerhalb des Testprogramms vorgenommen werden.

Beim Capture auf Anwendungsebene wird der Programmcode der Originalanwendung instrumentiert, indem Anweisungen zur Aufzeichnung von Daten (*Capturecode*) in die PUT oder an die Aufrufstelle eingefügt werden. Dies kann auf Ebene des Quelltextes, des Binärcodes oder einer von Compiler erstellten Zwischenform geschehen. Das Testprogramm enthält bei diesem Vorgehen wiederum Anweisungen zum Laden der Daten (*Replaycode*). Auch Mischformen sind möglich, zum Beispiel könnten auf Ebene der Laufzeitumgebung aufgezeichnete Daten auf Anwendungsebene geladen werden.

In dieser Arbeit werden sowohl Capture als auch Replay auf Anwendungsebene realisiert. Das Capture wird dabei durch Instrumentierung des Quelltextes der Originalanwendung umgesetzt. Das Replay durch Testtreiber, die den Quellcode zum Laden der aufgezeichneten Daten enthalten. Sowohl der Capturecode zum Aufzeichnen als auch der Replaycode zum Laden der Ein- und Ausgabedaten werden dabei automatisiert generiert. Dieser Ansatz erscheint aus anwendungsorientierter Sicht am vorteilhaftesten zu sein, da es mit dem gewohnten Material der EntwicklerInnen arbeitet: Fortran-Code. Den EntwicklerInnen wird so ermöglicht, in den C&R-Prozess durch Veränderung des generierten Codes einzugreifen. So kann beispielsweise auch der Capturezeitpunkt in Form einer beliebigen Bedingung in Fortran definiert werden. Die Handhabbarkeit dieses Ansatzes hängt dabei entscheidend von der Lesbarkeit und Verständlichkeit des Capture- und Replaycodes ab. Um abgeleitete Testfälle zu erstellen, können bei diesem Vorgehen entweder die aufgezeichneten Daten außerhalb der Anwendung bzw. des Testprogramms modifiziert oder die geladenen Testdaten im Testprogramm programmatisch verändert werden.

Das Aufzeichnen und Laden von Ein- und Ausgabedaten auf Anwendungsebene ist nur möglich, wenn Capture- und Replaycode auf sämtliche Ein- und Ausgabedaten zugreifen können. Um dies für gekapselte Daten, beispielsweise private Modulvariablen, zu ermöglichen, gibt es mehrere Möglichkeiten:

a) Capture- und Replaycode verwenden die öffentlichen Schnittstellen der betreffenden Module oder Bibliotheken, um die gekapselten Daten abzufragen bzw.

zu setzen. Dieser Weg funktioniert nur, wenn die Module oder Bibliotheken entsprechende Operationen bereitstellen. Die automatische Generierung von Capture- und Replaycode wird bei diesem Vorgehen durch die individuelle Gestaltung der Schnittstellen erschwert. Manuelle Eingriffe oder zusätzliche semantische Analysen der Module sind notwendig.

b) Der Code zum Aufzeichnen und Laden von gekapselten Daten wird in die betreffenden Module oder Bibliotheken eingefügt, deren Schnittstellen werden um Operationen, die diesen Code aufrufen, erweitert. Um die hinzugefügten Operationen vor „Missbrauch“ zu schützen, könnte mit Hilfe von Präprozessordirektiven dafür gesorgt werden, dass diese nur im C&R-Modus zur Verfügung stehen. Der Nachteil dieses Vorgehens ist, dass sich Capture- und Replaycode über die gesamte Anwendung verteilen, was deren Lesbarkeit und Verständlichkeit deutlich verringert. Es ist zudem nur dann mit angemessenem Aufwand durchzuführen, wenn die jeweiligen Module oder Bibliotheken im Quellcode vorliegen.

c) Private Daten werden öffentlich gemacht. Auch für dieses Vorgehen sollten die betreffenden Module oder Bibliotheken im Quellcode vorliegen. Um die Kapselung nicht zu zerstören, könnte auch hierbei mit Hilfe von Präprozessordirektiven dafür gesorgt werden, dass die Zugreifbarkeit der eigentlich privaten Daten nur im C&R-Modus erlaubt ist.

Vorgehen b) scheint mit dem Ziel dieser Arbeit, eine anwendungsorientierte Lösung zu schaffen, am stärksten im Widerspruch zu stehen, da hierbei Capture- und Replaycode über alle Module, von denen private Daten benötigt werden, verstreut wird. Eine manuelle Bearbeitung dieses Codes wird dadurch unnötig erschwert. Am erfolgversprechendsten erscheint daher das Vorgehen c). Dieses stößt bei Bibliotheken, die nur im Binärcode vorliegen, an seine Grenzen. In solchen Fällen könnte auf Vorgehen a) zurückgegriffen werden. Private Daten von nur binär vorliegenden Bibliotheken, die nicht über die öffentliche Schnittstelle abgefragt oder gesetzt werden können, bleiben auch bei dieser Strategie außen vor. Im Rahmen der Evaluation ist daher zu untersuchen, inwiefern dies in der Praxis zu Problemen führt.

8.3.5 Testorakelstrategien

Die Wahl eines geeigneten Testorakels hängt stark vom jeweiligen Anwendungsfall und den Testzielen ab, daher sollte ein C&R-Testsystem kein allgemeines Vorgehen vorschreiben. Als Standardvorgehen, welches von Fall zu Fall verändert werden kann, bietet sich jedoch ein Vergleich sämtlicher Ausgabedaten der PUT mit Referenzdaten an, welche ebenfalls aus der Originalanwendung gewonnen werden. Hierzu werden neben den Eingabedaten auch sämtliche Ausgabedaten der PUT beim Capture

aufgezeichnet. Zudem wird Code für den Datenvergleich als Teil des Testprogramm generiert. So kann bei Erstellung eines neuen Tests zunächst validiert werden, ob die Reproduktion der PUT erfolgreich war, d.h. sie sich im Test genauso verhält wie in der Originalanwendung. Anschließend kann die EntwicklerIn das Testorakel manuell an ihre tatsächlichen Anforderungen anpassen.

Als alternative Testorakel bieten sich beispielsweise an:

Implizites Orakel In einigen Fällen, speziell wenn eine neue Prozedur entwickelt wird, für die es noch keine Referenzdaten gibt, kann es bereits nützlich sein, nur zu überprüfen, ob ein Test kompilierbar und lauffähig ist.

Menschliches Orakel Möchte die EntwicklerIn die Testausgabedaten selbst begutachten, kann es manchmal nötig sein, diese in geeignete Darstellungsformen zu bringen.

Externes Orakel Anstatt den Vergleich von Test- und Referenzdaten im Testprogramm vorzunehmen, ließen sich die Testausgabedaten ebenso aufzeichnen wie die Referenzdaten und mit Hilfe externer Programme vergleichen.

Teilvalidierung Anstelle sämtlicher Ausgabedaten kann eine interessante Teilmenge zur Validierung ausgewählt werden.

Überprüfung spezifischer Eigenschaften Können keine exakten Referenzdaten zur Validierung verwendet werden, könnte es stattdessen möglich sein, nur bestimmte Eigenschaften der Ausgabedaten zu überprüfen (z.B. nicht negativ, größer als x etc.). Auch könnten metamorphe Relationen verwendet werden, um die Ergebnisse mehrerer Testfälle miteinander zu vergleichen (siehe Abschnitt 3.4.2)

Das Testorakel kann sich auch von Testausführung zu Testausführung unterscheiden. Beispielsweise könnte in einigen Fällen das implizite Orakel ausreichen, in anderen eine Teilvalidierung und wieder ein anderes Mal manuell die Ergebnisse begutachtet werden.

8.3.6 Reihenfolge

Die in Abschnitt 8.3.1 aufgeführten Schritte des C&R-Prozesses zur Erstellung eines Unittests müssen nicht streng in der dort aufgeführten Reihenfolge abgearbeitet werden. Eine anwendungsorientierte Softwarelösung sollte es ermöglichen, diese an die aktuellen Anforderungen anzupassen. Die BenutzerIn sollte darüber hinaus die Möglichkeit haben, in den Prozess eingreifen zu können, um Zwischenergebnisse ggf.

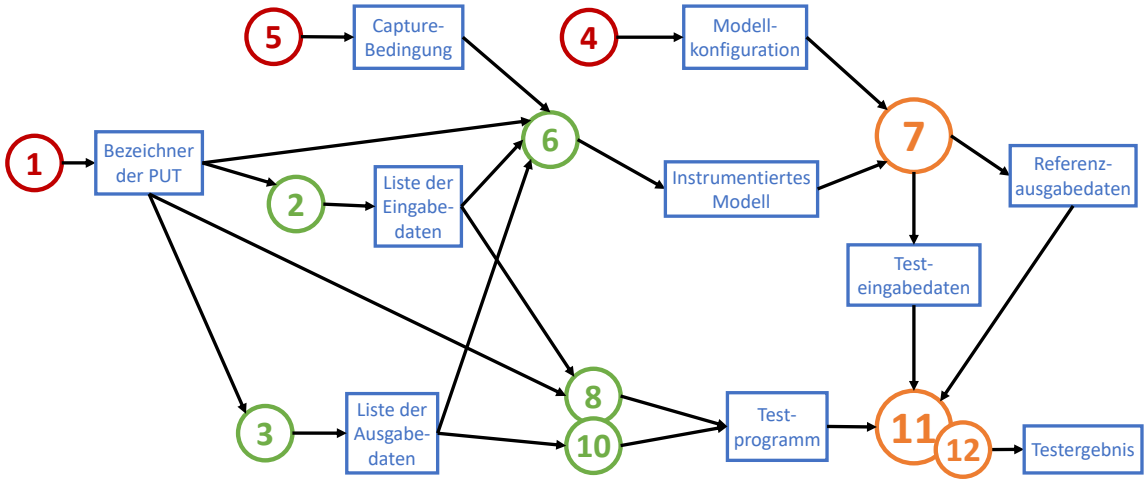


Abbildung 8.7: Abhängigkeiten zwischen den Schritten des C&R-Prozesses zur Erstellung eines Unittests

eigenständig zu verändern, um so Einfluss auf den weiteren Prozess nehmen zu können. Dennoch ist die Reihenfolge der Schritte auch nicht beliebig, da einzelne Schritte Zwischenergebnisse benötigen, die in anderen Schritten produziert werden.

In Abbildung 8.7 sind die Abhängigkeiten zwischen den Schritten und den Zwischenergebnissen, die sie produzieren dargestellt. Die runden Elemente stellen die Schritte aus Abschnitt 8.3.1 dar, die eckigen Elemente die Zwischenergebnisse. Da Zwischenergebnisse und Abhängigkeiten zum Teil von den gewählten Strategien für Analyse, Capture und Replay sowie vom Testorakel abhängen, sind sie hier exemplarisch für die in den vorangegangenen Abschnitten erläuterten Vorgehensweisen in dieser Arbeit aufgeführt. Dabei sind die Schritte, die von der EntwicklerIn durchzuführen sind, rot dargestellt und die automatisierten Schritte grün. Das eigentliche Capture & Replay, welches aus der Ausführung der instrumentierten Originalanwendung (Klimamodell) bzw. des erzeugten Tests besteht, ist orange dargestellt.

Bei den abgebildeten Zwischenergebnissen handelt es sich um Artefakte, die im Laufe des Prozesses benötigt werden. Die Schritte, die von der EntwicklerIn durchzuführen sind, sind jeweils ohne Vorbedingung dargestellt. Natürlich muss sich die EntwicklerIn im Klaren sein, welche Prozedur sie testen möchte (Schritt 1), bevor sie beispielsweise entscheiden kann, zu welchem Zeitpunkt die Aufzeichnung stattfinden soll. Aber dieses Wissen befindet sich in der Regel im Kopf der EntwicklerIn, es muss daher kein Artefakt erzeugt werden, damit sie diese Entscheidungen treffen kann.

Der Schritt 9 (Bestimmung eines Testorakels) ist in der Grafik nicht enthalten, da diese das Standardvorgehen beim Erzeugen eines neuen Tests darstellt. Bei diesem

soll immer ein automatisches Orakel erzeugt werden, welches eine vollständige Validierung auf Basis sämtlicher Ausgabedaten erzeugt.

Die Überlappung von Schritt 11 und 12 in der Grafik soll verdeutlichen, dass durch die Integration des Testorakels in das Testprogramm die Bewertung der Tests Teil der Testausführung ist. Das Testergebnis besteht in diesem Fall aus der Meldung, dass Test- und Referenzausgabedaten identisch sind oder aus einer Auflistung der Abweichungen. Andere Testorakel können von der EntwicklerIn im Anschluss an die initiale Testerzeugung manuell erstellt werden. Grundsätzlich folgt auf die erstmalige Testerstellung ein Zyklus aus wiederholten Testausführungen, Änderungen der PUT und Änderungen des Testprogramms. Dieser Zyklus ist in der Grafik nicht eingezeichnet.

Um automatisiert die Ein- und Ausgabedaten zu identifizieren (Schritt 2 und 3) und um die Instrumentierung des Quellcodes vorzunehmen (Schritt 6) wird jedoch zwingend die Information, um welche Prozedur es geht, benötigt. Für Schritt 6 wird zudem die Bedingung, die den Capturezeitpunkt beschreibt, benötigt sowie die zuvor zu ermittelnden Ein- und Ausgabedaten. Schritt 6 erzeugt dann das instrumentierte Modell, welches zur Durchführung des Capture benötigt wird (Schritt 7). Hierfür wird das Klimamodell in der Konfiguration ausgeführt, welche in Schritt 4 ausgewählt wird.

Unabhängig von der Instrumentierung der Originalanwendung, kann der Testtreiber erstellt werden (Schritt 8). Hierzu wird ebenfalls der Name der PUT sowie die Liste der Eingabedaten benötigt, um die PUT im Test mit diesen Eingabedaten aufrufen zu können. Zur Erstellung des Testtreibers gehören ggf. auch Änderungen am Anwendungscode, um die Testbarkeit herzustellen, wie beispielsweise das Zugreifbarmachen privater Daten (vgl. Abschnitt 8.3.3). Um das automatische Testorakel in das Testprogramm integrieren zu können (Schritt 10) wird zudem die Liste der Ausgabedaten benötigt. Da beide Schritte Teil der Erstellung des Testprogramms sind, werden diese ggf. gemeinsam ausgeführt, daher auch hier die Überlappung. Um letztendlich das Testprogramm ausführen zu können (Schritt 11: Replay), muss jedoch zuvor die instrumentierte Originalanwendung ausgeführt werden (Schritt 7: Capture), um Eingabe- und Referenzdaten für den Test zu erzeugen. Grundsätzlich muss das Capture nur einmal ausgeführt werden, um einen Satz an Testdaten zu gewinnen, mit welchem dann wiederholt der Test ausgeführt werden kann. Der Capturecode kann daher nach einmaliger Ausführung aus dem Modell wieder entfernt werden[13]. Ggf. muss der Captureprozess jedoch wiederholt werden, wenn die Testdaten aktualisiert werden sollen.

[13] Mutmaßlich bedeutet dies in der Praxis, dass die EntwicklerIn die Änderungen mit Hilfe der Versionsverwaltung zurücksetzt bzw. von einem „Capture-Zweig“ zurück auf einen Entwicklungszweig wechselt.

8.3.7 Anpassbarkeit

Trotz vieler Gemeinsamkeiten unterscheiden sich einzelne Klimamodelle voneinander. Dies betrifft nicht nur die implementierten Algorithmen, sondern auch technische Rahmenbedingungen, wie beispielsweise eingesetzte Bibliotheken, Abstraktionsebenen in der Softwarearchitektur oder die Buildinfrastruktur. Daneben pflegen unterschiedliche Entwicklungsteams unterschiedliche Programmierkulturen. Ein Beispiel hierfür sind etwa unterschiedliche Codekonventionen, z.B. zur Benennung von Variablen oder zur Formatierung des Quelltextes.

Eine anwendungsorientierte C&R-Software sollte sich an die modell- und teamspezifischen Gegebenheiten anpassen lassen, anstatt dass sich andersherum Modell und Team an die C&R-Software anpassen. Dies betrifft unter anderem die Abfolge der notwendigen Schritte, welche im Abschnitt zuvor diskutiert wurde. Ein wichtiger Aspekt ist jedoch auch der von der Software generierte Code, d.h. auf der einen Seite der Code zur Instrumentierung des Modells und auf der anderen Seite der Code des Testprogramms. Eine „One-Fits-All“-Lösung dürfte hier nur schwer den spezifischen Anforderungen einzelner Modelle und Entwicklungsteams gerecht werden. Stattdessen sollte es möglich sein, den generierten Code an die jeweiligen Anforderungen anzupassen. Das Beispiel Codekonventionen wurde bereits erwähnt. Weitere Beispiele anzupassender Aspekte sind unter anderem:

Speichertechnologie Um Ein- und Ausgabedaten müssen beim Capture in geeigneter Form abgespeichert werden. Dies kann zum Beispiel in Datenbanken oder im Dateisystem geschehen. Wird Letzteres gewählt, könnte zum Beispiel das Dateiformat Gegenstand der Anpassung sein. Darüber hinaus verfügen Klimamodelle in der Regel bereits über Abstraktionsschichten für Dateisystemoperationen. Auf diese könnte bei der Speicherung der C&R-Daten zurückgegriffen werden.

Externe Bibliotheken Verwendet ein Modell externe Bibliotheken, die auch im Test benötigt werden, müssen diese ggf. im Testprogramm eingebunden und initialisiert werden.

Testorakel Auch das automatische Testorakel kann auf unterschiedliche Art und Weise implementiert werden. Für Fortran stehen zum Beispiel mehrere Unittestframeworks zur Verfügung (siehe auch Fortran Wiki, 2018), welche hierfür verwendet werden könnten.

Derartige Anpassungen sollten jedoch nicht jedes Mal vorgenommen werden müssen, wenn ein einzelner Test erstellt werden soll. Vielmehr sollte es möglich sein, diese Anpassungen einmalig für ein bestimmtes Modell vorzunehmen und einzelnen EntwicklerInnen zur Verfügung zu stellen, so dass diese nur noch die genannten Schritte

durchführen müssen. Ziel der Softwareunterstützung ist schließlich die Hürde zur Erstellung von Unittests zu senken. Hier bietet sich zudem ein arbeitsteiliges Vorgehen an. Die einmalige Anpassung der Software an das jeweilige Modell könnte von erfahreneren, technisch versierteren EntwicklerInnen vorgenommen werden, ggf. mit Unterstützung externer BeraterInnen, während die Erstellung von Tests von allen EntwicklerInnen durchgeführt wird. Geeignete Standardeinstellungen sollten insgesamt den Einstieg erleichtern. Dabei steht es nicht im Widerspruch mit den Zielen dieser Arbeit, wenn die modellspezifische Anpassung der Software mit einem (gefühlt) hohen Aufwand verbunden ist, solange die Erstellung der Tests dadurch vereinfacht wird.

8.3.8 Benutzungsschnittstelle

Eine anwendungsorientierte Softwarelösung sollte sich an die gewohnte Arbeitsumgebung der BenutzerInnen anpassen. Die gewohnte Arbeitsumgebung von EntwicklerInnen von Klimamodellen ist die Linux/Unix-Kommandozeile (siehe auch Abschnitt 6.3). Die C&R-Software sollte sich daher auch über die Kommandozeile steuern lassen. Dies hat den weiteren Vorteil, dass sie auch dann einsetzbar ist, wenn keine grafische Oberfläche zur Verfügung steht, zum Beispiel beim Fernzugriff auf Hochleistungsrechner. Darüber hinaus ermöglicht es der EntwicklerIn den eigenen, individuellen Arbeitsablauf mit Hilfe von Skripts weitergehend zu automatisieren.

Im Hinblick auf die oben beschriebene Arbeitsteilung zwischen denjenigen, die die einmalige modellspezifische Einrichtung der Software vornehmen und den „normalen“ EntwicklerInnen, die mit Hilfe der Software Tests erstellen, sollte es eine Trennung zwischen den allgemeinen modellspezifischen Einstellungen und den von Fall zu Fall benötigten Angaben (Bezeichner der PUT, Capturebedingung) geben. Dabei bietet es sich an, die allgemeinen Einstellungen beispielsweise in Konfigurationsdateien vorzunehmen, die an alle EntwicklerInnen verteilt und ggf. im Versionsverwaltungssystem gepflegt werden können. Um bei der Definition des Capturezeitpunkts in Form der Capturebedingung größtmögliche Flexibilität zu ermöglichen, sollte diese als Teil des Capturecodes in Fortran formuliert werden können.

8.4 Weitere Einsatzmöglichkeiten

Hauptziel dieser Arbeit ist es, eine Unterstützung für die Erstellung von Unittests einzelner Prozeduren zu erschaffen. Der C&R-Ansatz eignet sich jedoch auch für andere Einsatzzwecke. So können die mit diesem Verfahren erstellten ausführbaren Einheiten, die die isolierten Prozeduren enthalten, beispielsweise auch beim Debugging eingesetzt werden. Gelingt es, eine fehlerhafte Ausführung der PUT per C&R zu

reproduzieren, lässt sich diese mit Hilfe der isolierten Prozeduren leichter und schneller analysieren. Auch die aufgezeichneten Daten können in diesem Fall Hilfsmittel sein, um die Fehlerursache einzugrenzen. Um die fehlerhafte Ausführung reproduzieren zu können, muss jedoch der Zeitpunkt, an dem diese auftritt, bekannt sein. Sind einzelne Bedingungen, die zum Fehler führen, bekannt, können diese als Capturebedingung verwendet werden. Ggf. ist es notwendig, die fehlerhafte Anwendung zweimal auszuführen, wobei man im ersten Durchlauf die Anzahl der Ausführungen bis zum Fehler mitzählen lässt und beim zweiten Mal die ermittelte Ausführung aufzeichnet. Dies setzt allerdings ein deterministisches Auftreten des Fehlers voraus.

Für solche Zwecke erstellte Datensätze eignen sich im Rahmen von Regressionstests wiederum sehr gut, um zu überprüfen, dass die mit dieser Hilfe beseitigten Fehler in späteren Versionen nicht erneut auftreten. Stellt sich jedoch heraus, dass ein Fehler durch eine fehlerhafte externe Bibliothek, einen fehlerhaften Compiler oder fehlerhafte Hardware verursacht wird, können die isolierten Prozeduren den Herstellern im Rahmen von Bugreports zur Verfügung gestellt werden.

Neben der Fehlerursachensuche erleichtern kleinere Ausführungseinheiten auch jegliche Form der dynamischen Analyse. Da bei diesen die zu analysierende Software ausgeführt werden muss, lassen sie sich gezielter und schneller durchführen, wenn nur der zu analysierende Teil ausgeführt wird. Ein Beispiel hierfür wären etwa Performanceanalysen.

8.5 Verwandte Ansätze

Die Idee, einzelne Codeabschnitte einer größeren Anwendung mit Hilfe von C&R zu isolieren bzw. Unittests auf dieser Grundlage zu erstellen, ist nicht neu. Im Folgenden werden vergleichbare Ansätze vorgestellt und mit dieser Arbeit verglichen. Einige Autoren verwenden ebenfalls die Begriffe Capture und Replay, andere benutzen andere Bezeichnungen. Hervorzuheben ist *KGEN* (Abschnitt 8.5.6), welches parallel zu dieser Arbeit entstand und ebenfalls für in Fortran geschriebene Klimamodelle entwickelt wurde.

8.5.1 C&R für Java-Anwendungen

Orso und Kennedy (2005) bzw. Joshi und Orso (2007) beschreiben einen Ansatz und einen Prototyp namens *SCARPE* für ein aktionsbasiertes „selektives Capture & Replay“ von Java-Anwendungen. Ziel ist es, reale Daten, die ein Subsystem einer Java-Anwendung im Produktiveinsatz zu verarbeiten hat, aufzuzeichnen, um diese für Analysen und Tests verwenden zu können. Dazu werden sämtliche Interaktionen

(u.a. Methodenaufrufe und Feldzugriffe[14]) des Subsystems, welches eine zuvor ausgewählte Menge von „zu beobachtenden“ Klassen ist, mit seiner Umgebung, d.h. mit allen nicht beobachteten Systemteilen, aufgezeichnet. Anschließend können diese mit dem isolierten Subsystem wiederholt werden. Es geht dabei also nicht um einzelne Prozedur-/Methodenaufrufe, sondern um Serien von derartigen Interaktionen. Da die Autoren weder beschreiben, wie der Startzeitpunkt für eine Aufzeichnung festgelegt wird, noch wie das Subsystem in einen Zustand gebracht wird, den es zu Beginn einer Aufzeichnung inne hatte, ist davon auszugehen, dass immer vom Anwendungsstart an aufgezeichnet wird. Die aufzuzeichnenden Daten werden dabei zur Laufzeit ermittelt. Als einen möglichen Anwendungsfall nennen die Autoren das Erzeugen von Unittests auf Basis von Systemtests, geben jedoch zu bedenken, dass die mit ihrer Technik aufgezeichneten Daten ggf. nur eingeschränkt für Regressionstests zu nutzen sind, wenn neuere Versionen des zu testenden Subsystems weitere Daten benötigen, die nicht mit aufgezeichnet wurden.

Das Erzeugen von Unittests aus Systemtests wird als Hauptziel von David Saff u. a. (2005) genannt, welche einen sehr ähnlichen Ansatz für Java-Systeme wie Orso, Kennedy und Joshi verfolgen. Die Autoren bezeichnen dieses Vorgehen als „Test Factoring“.

Im Gegensatz zu den Ansätzen von Orso/Kennedy/Joshi und Saff u.a. basiert der Ansatz von Sebastian Elbaum u. a. (2009) nicht auf der Aufzeichnung von Objektinteraktionen, sondern auf der Erfassung des Programmzustands vor und nach der Ausführung einer Methode, ähnelt in dieser Hinsicht somit dem zustandsbasierten Vorgehen in dieser Arbeit. Auch diese Autoren zielen damit auf die Ableitung von Unittests aus Systemtests ab. Sie nennen dies „Carving and Replay“. In ihrem Artikel präsentieren sie zudem einen allgemeinen Begriffsrahmen für diese Methode. So führen sie beispielsweise die Unterscheidung zwischen zustandsbasiertem (*state based*) und aktionsbasiertem (*action based*) Vorgehen ein. Darüber hinaus schlagen sie mehrere Vorgehen, sog. *Projektionen (projections)* vor, um für ein Java-Programm den aufzuzeichnenden Zustand auf die benötigten Daten zu reduzieren.

Allen Java-Lösungen ist gemein, dass sie die Instrumentierung zum Zwecke des Aufzeichnens und Laden der Daten auf Bytecodeebene vornehmen. Menschliche Eingriffe in den C&R-Prozess sind nicht vorgesehen.

8.5.2 A Code Isolator

Einen ähnlichen Ansatz wie diese Arbeit, jedoch mit unterschiedlichem Ziel, beschreiben Yoon-Ju Lee und Mary Hall (2005). Ihr *Code Isolator* ist in der Lage, beliebige

[14] In objektorientierten Sprachen wie Java versteht man unter *Feldern* die objekt- oder klassenweiten Variablen.

Codeabschnitte einer größeren wissenschaftlichen Anwendung in eine Prozedur auszugliedern und diese mit Hilfe von „repräsentativen Eingabedaten“ auszuführen. Die Eingabedaten werden auch hier durch Ausführung von der Originalanwendung gewonnen. Hauptziel ist es, die gezielte Leistungsoptimierung des isolierten Codeteils zu vereinfachen, indem diese unabhängig von Originalanwendung ausgeführt werden kann. Yoon-Ju Lee und Mary Hall zielen dabei besonders auf die Optimierung des Cacheverhaltens ab. Zu diesem Zwecke kann der Code Isolator sogar den Cachestatus zum Zeitpunkt der Ausführung des fraglichen Codeabschnitts aus der Originalanwendung wiederherstellen.

Der Code Isolator baut auf dem *Stanford SUIF Compiler* (Wilson u. a., 1994) auf und unterstützt Fortran- und C-Code. Evaluiert wurde er mit einem kleinen Codeabschnitt (etwa acht Zeilen FORTRAN 77) aus dem proprietären Finite-Elemente-Berechnungsprogramm *LS-DYNA* (lstc.com, LS-DYNA). Dabei wurde ein großer Teil der Arbeit, um den Cachestatus zu reproduzieren, manuell durchgeführt. Inwieweit der Code Isolator in der Lage wäre, auch größere Prozeduren aus Klimamodellen zu reproduzieren, ist unklar. Das Programm ist nicht mehr verfügbar. Anwendungsorientierte Aspekte diskutieren Lee und Hall in ihrem Artikel nicht.

8.5.3 Codelet Finder

Ebenfalls auf rechenintensive, wissenschaftliche Anwendungen in C und Fortran zielt der *Codelet Finder* ab, welcher von Chadi Akel u. a. (2013) beschrieben wird. Auch dieser soll vornehmlich Optimierungsaufgaben unterstützen. Zu diesem Zwecke analysiert der Codelet Finder eine Anwendung mit einem Profiler, um besonders rechenintensive Codeabschnitte, sog. *Hotspots* zu finden. In diesen Hotspots werden alle Schleifen markiert und anschließend als unabhängig von der Originalanwendung laufende Programme extrahiert. Diese extrahierten Schleifen nennen die Autoren *Codelets*. Der Anfangszustand für die Ausführung eines Codelets wird durch Ausführung der Originalanwendung gewonnen, indem ein vollständiger Speicherabzug vor Ausführung des Codelets aufgezeichnet wird. Dieses Vorgehen unterscheidet den Codelet Finder vom Code Isolator und insbesondere auch von dieser Arbeit, in der die Reduktion der auszuzeichnenden Daten ein wesentliches Ziel ist.

8.5.4 CERE

Eine Weiterentwicklung des Codelet Finders ist der *Codelet Extractor and REplayer (CERE)* von Pablo De Oliveira Castro u. a. (2015). Dieser arbeitet nicht mit vollständigen Speicherabzügen, sondern speichert nur relevante Speicherseiten. Auch hier ist der Einsatzzweck die Leistungsoptimierung. Dazu wird zusätzlich auch der Cachestatus

mit aufgezeichnet. Die Instrumentierung der Originalanwendung sowie die Erstellung des isolierten Codelets findet auf Ebene der Zwischensprache der *LLVM Compiler Infrastructure* (Lattner und Adve, 2004) statt. Dies hat den Vorteil, dass CERE sämtliche LLVM-unterstützen Sprachen[15] unterstützt, jedoch schränkt es zum einen die Portabilität des isolierten Codelets ein, da es nur mit LLVM-kompatiblen Compilern übersetzt werden kann, und zum anderen erschwert es manuelle Änderungen am Capture- und Replaycode.

8.5.5 ALM Function Test Framework

Während dieser Arbeit erschienen mehrere Veröffentlichungen, die ein C&R-System beschreiben, das zur Erstellung von Unittests für einzelne Prozeduren des *Community Land Model (CLM)* bzw. des *ACME Land Model (ALM)* verwendet wird (Wang, Xu u. a., 2014; Wang, Yao und Winkler, 2016; Yao, Jia u. a., 2016; Yao, Wang u. a., 2017). Das CLM ist die Landkomponente des *Community Earth System Model (CESM)*, welches federführend vom US-amerikanischen NCAR entwickelt wird (siehe auch Lawrence, Oleson u. a., 2011; Hurrell u. a., 2013), während das ALM zum *Accelerated Climate Modeling for Energy (ACME)* gehört, welches eine CESM-Abspaltung des US-Department of Energy ist (siehe auch Yao, Jia u. a., 2016). Die Autoren verwenden verschiedene Namen für das beschriebene System, z.B. *ALM Function Test Framework* (Wang, Yao und Winkler, 2016), *ALM Unit Testing Platform* (Wang, Yao und Winkler, 2016; Yao, Jia u. a., 2016) oder schlicht *Unit Testing Framework (UTF)* (Yao, Wang u. a., 2017).

In dem beschriebenen C&R-System werden die Daten aus der Originalanwendung nicht zur späteren Verwendung zwischengespeichert, sondern live an die Unittests übertragen (Abbildung 8.8). Neben den Eingabedaten der Unittests werden auch Ausgabedaten zur Validierung aus der Originalanwendung übertragen. Die Autoren nennen dieses Prinzip *In Situ Data Infrastructure* (Yao, Jia u. a., 2016). Das Ziel hinter der Datendirektübertragung ist es, kostenintensive Dateisystemoperationen zu sparen (vgl. Yao, Wang u. a., 2017). Ein Geschwindigkeitsvorteil der Unittests gegenüber Ende-zu-Ende-Tests, welcher in dieser Arbeit angestrebt wird, dürfte sich auf diese Weise nicht ergeben, da diese parallel zu den Unittests immer mitlaufen müssen. Dieses Vorgehen eignet sich daher mutmaßlich nicht dazu, die Testfrequenz während der aktiven Entwicklung zu erhöhen.

Um die Übertragung der Daten aus der Originalanwendung zu ermöglichen, wird diese instrumentiert. Capture- und Replaycode werden wie in dieser Arbeit auf Grundlage einer statischen Analyse, mit der u.a. die verwendeten globalen Variablen ermittelt

[15] Castro u. a. (2015) führen hier beispielhaft C, C++, Fortran und D an, allerdings war die Fortran-Unterstützung für LLVM in der Vergangenheit sehr eingeschränkt. Seit 2017 gibt es mit *Flang* (GitHub, Flang) jedoch einen LLVM-kompatiblen Compiler für Fortran.

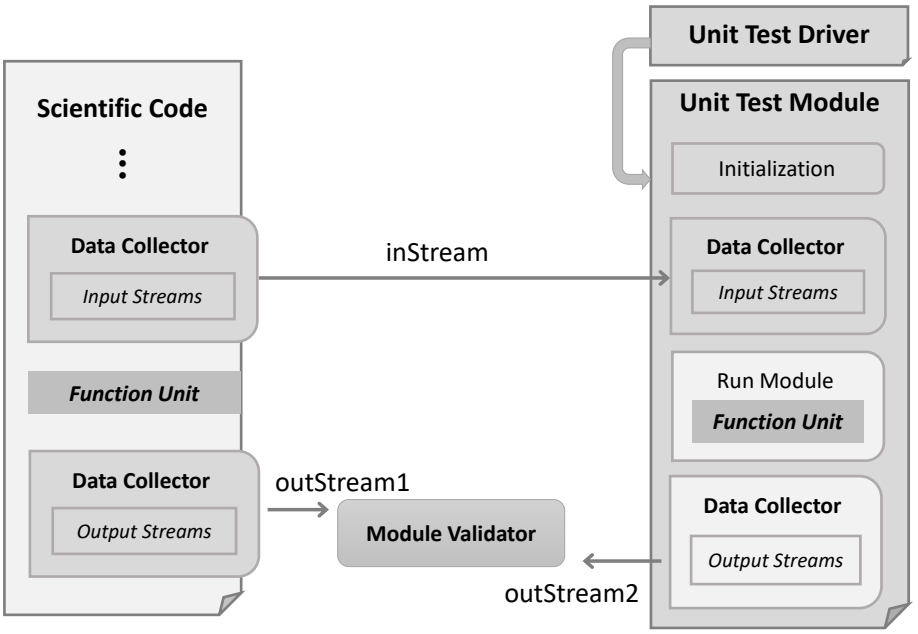


Abbildung 8.8: Schematische Darstellung der Funktionsweise des ALM Function Testing Frameworks (Wang, Yao und Winkler, 2016)

werden, automatisch erzeugt. Mit Hilfe der Analyse werden auch Initialisierungsprozeduren ermittelt, die in der Originalanwendung vor der zu testenden Prozedur aufgerufen werden und für das Allozieren und Initialisieren von Speicherblöcken zuständig sind. Diese werden in den Unittest übernommen. Infrastrukturbibliotheken, z.B. für MPI oder I/O werden in den Unittests durch Platzhalter ersetzt (vgl. Yao, Jia u. a., 2016). Modifikationen des Capture- und Replaycodes durch die BenutzerInnen scheinen nicht vorgesehen.

Das ALM Function Test Framework wurde speziell für das CLM/ALM entwickelt und ist keine allgemein verfügbare Softwarelösung zum Testen vergleichbarer Anwendungen, wie sie in dieser Arbeit angestrebt wird. Anwendungsorientierte Aspekte werden von den Autoren zumindest erwähnt, wie etwa in Yao, Jia u. a., 2016:

> *„After experimenting with several test tools in scientific code, we repeatedly had issues with scalability and usability. Based on former experiences, we wanted to build a test platform that would*
>
> - *be widely used by developers and module builders to fix problems and collect variables,*
> - *integrate smoothly into the existing workflow, and*
> - *produce test suites automatically and provide validation.“*

Wie das System konkret benutzt wird, wie es hilft „Probleme zu fixen“ und wie der „existierende Workflow“ aussieht, wird leider nicht diskutiert.

8.5.6 KGEN

Die Arbeit, die dieser am nächsten kommt, stammt von Youngsung Kim u. a. (2016). Ihr *Kernel GENerator (KGEN)* ist ein C&R-System, das ebenfalls auf große Fortran-Anwendungen ausgerichtet ist. Es wurde gleichzeitig zu dieser Arbeit am NCAR zum Testen des Klimamodells CESM entwickelt. Als Einsatzzwecke werden u.a. Debugging und Leistungsoptimierung genannt.

Die Funktionsweise ähnelt dem in Abschnitt 8.3.1 beschriebenen Prozess. Anstelle einzelner Prozeduren lassen sich jedoch beliebige Codeabschnitte, von den Autoren *Kernel* genannt, extrahieren. Diese werden von der BenutzerIn mit Hilfe von Direktiven markiert. Auch KGEN ermittelt dann mit Hilfe einer statischen Analyse die benötigten Variablen und fügt automatisch Capturecode in die Originalanwendung ein und erzeugt eine unabhängig lauffähige Version des Kernels. Zudem werden automatisch Buildskripte erzeugt, um den generierten Code zu kompilieren und auszuführen. Der gesamte Prozess von der Codeanalyse bis zur Ausführung und Validierung des extrahierten Kernels (Abbildung 8.9) ist vollständig automatisiert und bietet kaum Eingriffsmöglichkeiten für die BenutzerIn, bis auf einige Kommandozeilenparameter, die beim Aufruf von KGEN übergeben werden müssen. Mit diesen lassen sich z.B. Dateipfade setzen, Compilerdirektiven für die Buildskripte setzen oder einzelne Module von der Analyse ausschließen.

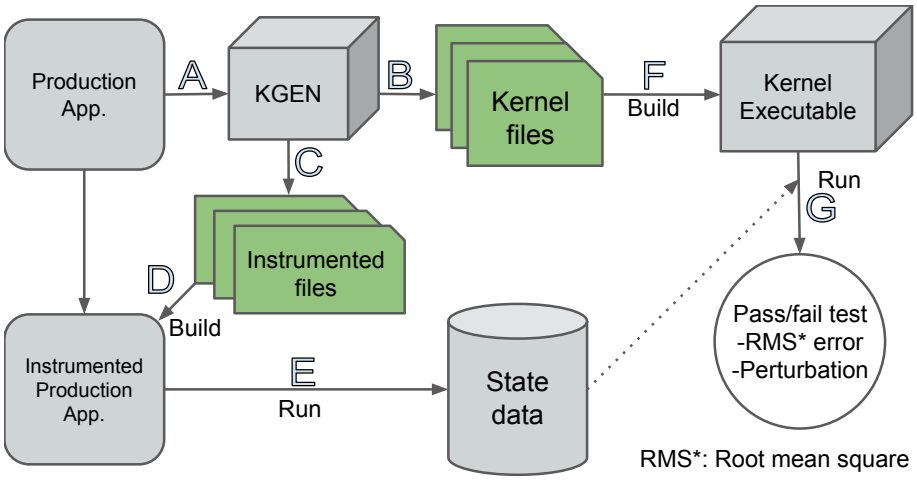


Abbildung 8.9: Schematische Darstellung der Funktionsweise von KGEN (Kim u. a., 2016)

Der von KGEN generierte Capturecode wird nicht nur lokal in das Modul, in dem sich der Kernel befindet, eingefügt, sondern über die gesamte Originalanwendung verteilt, indem der Code für die Serialisierung einzelner benutzerdefinierter Typen in die Module eingefügt wird, in denen die Typen jeweils definiert sind[16]. Dieser Serialisierungscode ist sehr komplex, eine Modifikation durch die BenutzerIn daher kaum möglich. KGEN serialisiert und speichert zudem alle Verbunddatentypen immer vollständig und beschränkt sich nicht auf die Komponenten, die vom Kernel tatsächlich benutzt werden.

KGEN ist frei verfügbar (GitHub, KGEN) und grundsätzlich für andere Fortran-Anwendungen einsetzbar. Die Funktionsweise und der vollautomatisierte Prozess sind jedoch stark auf die Anforderungen von CESM zugeschnitten. Entwickler anderer Klimamodelle, die KGEN ausprobiert haben, berichteten, dass die Verwendung entweder sehr schwierig oder unmöglich war. Die gleiche Erfahrung habe ich selbst gemacht. Ein Grund hierfür ist jedoch auch das implementierte Analyseverfahren, welches auf einem *abstrakten Syntaxbaum (abstract syntax tree, AST)* basiert. Um diesen zu erstellen, muss der gesamte Anwendungscode geparst werden. Der von KGEN verwendete Parser unterstützt(e) jedoch nicht alle aktuellen Fortran-Sprachkonstrukte. Enthält der Anwendungscode jedoch nicht unterstützte Konstrukte führt dies zum sofortigen Programmabbruch. Dieses Problem ließe sich jedoch durch Erweiterung des Parsers beheben und ist nicht konzeptioneller Natur. Ggf. unterstützen neuere Versionen von KGEN bereits weitere Sprachkonstrukte.

8.6 Zusammenfassung

In diesem Kapitel wurde das Capture-&-Replay-Verfahren vorgestellt und dargelegt, wie dieses für die Erstellung von Unittests in der Klimamodellierung eingesetzt werden kann. Bei diesem Vorgehen werden Testdaten für Unittests aus existierenden E2E-Tests gewonnen, indem Ein- und ggf. auch Ausgabedaten einer zu testenden Prozedur aufgezeichnet werden, wenn diese innerhalb eines E2E-Tests ausgeführt wird. Darauf aufbauend wurden die Arbeitsschritte diskutiert, aus denen ein C&R-Prozess besteht, welche davon automatisiert werden und welche Verfahren hierzu verwendet werden sollen, wie sich der Prozess an die individuellen Anforderungen der BenutzerInnen anpassen lässt und wie eine Benutzungsschnittstelle gestaltet sein soll. Die Ergebnisse dieser Diskussion bilden die Grundlage eines Softwarewerkzeugs, welches im nachfolgenden Kapitel vorgestellt wird. Hierzu gehören u.a.:

[16] KGEN verwendet hier konsequent das in Abschnitt 8.3.4 beschriebene Vorgehen b), sowohl für private als auch für öffentliche Daten.

- Zu automatisieren sind die Identifizierung von Ein- und Ausgabedaten der PUT, das Capture bzw. die Vorbereitung der Originalanwendung für das Capture, die Erstellung eines Testtreibers sowie eines automatischen Standardtestorakels.
- Ein- und Ausgabedaten der PUT werden mit Hilfe einer statischen Quellcodeanalyse identifiziert.
- Das Capture geschieht durch Instrumentierung der PUT in der Originalanwendung.
- Das Laden der aufgezeichneten Daten für das Replay geschieht direkt im Testtreiber.
- Als Standardtestorakel werden sämtliche Ausgabedaten der PUT mit aufgezeichneten Referenzdaten aus der Originalanwendung verglichen.
- Für Instrumentierung und Testprogramm wird Fortran-Code generiert, welcher von der BenutzerIn bearbeitet werden kann.
- Die Reihenfolge der notwendigen Schritte soll durch das Softwarewerkzeug nur insoweit vorgegeben sein, wie diese logisch bedingt ist.
- Das Werkzeug soll sich an die Zielumgebung anpassen und nicht andersherum, dies betrifft insbesondere auch den für die Instrumentierung der Originalanwendung und den für das Testprogramm zu generierenden Code.
- Umgebungsspezifische Einstellungen sollen in Konfigurationsdateien festgelegt werden, die zwischen BenutzerInnen ausgetauscht werden können.
- Das Softwarewerkzeug soll sich über die Linux/Unix-Kommandozeile steuern lassen.

Kapitel 9

Umsetzung

In diesem Kapitel wird das Softwarewerkzeug *FortranTestGenerator* vorgestellt. Der FortranTestGenerator ist eine Umsetzung des im vorherigen Kapitel vorgestellten Capture-&-Replay-Verfahrens (C&R) zum Erstellen von Unittests für einzelne Prozeduren größerer Fortran-Anwendungen. Erstmalig beschrieben wurde er in Hovy und Kunkel, 2016. Er automatisiert die folgenden Prozessschritte (siehe auch Abschnitt 8.3.1) für eine von der BenutzerIn vorgegebene zu testende Prozedur (PUT):

- Identifikation von Ein- und Ausgabedaten mit Hilfe statischer Codeanalyse
- Vorbereitung des Capture durch Instrumentierung der Originalanwendung (*Capturecode*)
- Erstellung des Testtreibers und des automatischen Testorakels (*Replaycode*)

Außerdem verändert der FortranTestGenerator von der PUT verwendete Module, indem es private Modulvariablen, welche zu den Ein- oder Ausgabedaten der PUT gehören, öffentlich macht (siehe auch Abschnitt 8.3.4). Der hierzu erzeugte Code, wird hier *Exportcode* genannt. Sämtlicher vom FortranTestGenerator generierter Code kann mit Hilfe sog. *Templates* an die Anforderungen der jeweiligen Umgebung angepasst werden.

Die grundlegende Architektur des Systems wird in Abschnitt 9.1 vorgestellt. In Abschnitt 9.2 wird das Vorgehen bei der Entwicklung des FortranTestGenerators erläutert. In den Abschnitten 9.3, 9.4 und 9.5 wird die für die Identifizierung von Ein- und Ausgabe zuständige Komponente, FortranCallGraph, vorgestellt, sowie die darin verwendeten Verfahren zur Codeanalyse. Mit der Funktionsweise des eigentlichen FortranTestGenerator-Programms, welches für die Codegenerierung für Instrumentierung und Testprogramm zuständig ist, befasst sich Abschnitt 9.6. Der Arbeitsablauf der Testerzeugung mit Hilfe des FortranTestGenerators wird in Abschnitt 9.7 diskutiert, die Anpassbarkeit des Softwarewerkzeugs in Abschnitt 9.8. Abschnitt 9.9 befasst sich schließlich mit den für die Verwendung des FortranTestGenerators benötigten externen Softwarepaketen.

9.1 Grundlegende Architektur

Das Gesamtsystem besteht aus zwei Komponenten: dem Analyseprogramm *FortranCallGraph (fcg)* und dem eigentlichen Testgenerator *FortranTestGenerator (FTG)*[17]. Beide Programme sind in *Python* geschrieben und unter einer Open-Source-Lizenz verfügbar (GitHub, fortesg). Python wurde gewählt, da diese Sprache eine hohe Akzeptanz im Bereich der wissenschaftlichen Softwareentwicklung genießt (siehe auch Millman und Aivazis, 2011). Eine Implementation direkt in Fortran oder C/C++ erschien nicht zweckmäßig und der Aufgabe nicht angemessen.

Abbildung 9.1 zeigt das grundlegende Zusammenspiel der beiden Komponenten. Mit grünen Pfeilen sind die Aufrufbeziehungen und mit roten Pfeilen der Datenfluss dargestellt, wobei allerdings auch beim Aufruf Daten in Form von Kommandozeilen- oder Funktionsparametern gesendet werden.

Um Capture-, Export und/oder Replaycode für eine PUT zu erstellen, ruft die BenutzerIn (grün) das FortranTestGenerator-Programm auf der Kommandozeile auf und übergibt dabei als Parameter den Namen der PUT und den Namen des *PUT-Moduls* (Modul, in dem sich die PUT befindet). FTG ruft dann zunächst fcg auf, welches die Originalanwendung analysiert und eine Liste der Ein- und Ausgabedaten der PUT zurückliefert. fcg ist zwar ein eigenständiges Programm, das ebenfalls über die Kommandozeile aufgerufen werden kann, allerdings wird es von FTG wie eine Bibliothek verwendet, d.h. die Kommunikation findet über eine *Programmierschnittstelle (application programming interface, API)* statt. Neben den Namen der Variablen, die zu den Ein- und Ausgabedaten gehören, liefert fcg auch Kontextinformationen, wie etwa Datentypen von Variablen, Prozedurargumente oder in welchen Dateien und in welchen Quelltextzeilen bestimmte Elemente zu finden sind.

Auf Basis der von fcg gelieferten Informationen generiert FTG den gewünschten Code. Capture- und Exportcode werden direkt in die entsprechenden Module der Originalanwendung eingefügt, der Replaycode wird als neue Datei abgelegt. Capture- und Replaycode sind dabei unabhängig von der aufzuzeichnenden Ausführung der PUT, d.h. sie enthalten Aufzeichnungs- und Ladeoperationen für alle Variablen, die in irgendeinem Ausführungspfad der PUT benötigt werden. Mit demselben Code können daher mehrere Testdatensätze aufgezeichnet und geladen werden. Zudem spielt es keine Rolle, ob zunächst der Capturecode ausgeführt und dann der Replaycode erzeugt wird oder andersherum. Für einzelne Ausführungen kann dies jedoch bedeuten, dass mehr Variablen aufgezeichnet und geladen werden als konkret benötigt.

[17] Beide Namen waren ursprünglich Arbeitstitel, welche im Laufe der Arbeit nie geändert wurden. Aufgrund der Ähnlichkeit der beiden Abkürzungen „FTG“ und „fcg“ wird Letztere in dieser Arbeit in Kleinbuchstaben gesetzt, um sie für eine bessere Lesbarkeit typografisch voneinander abzuheben.

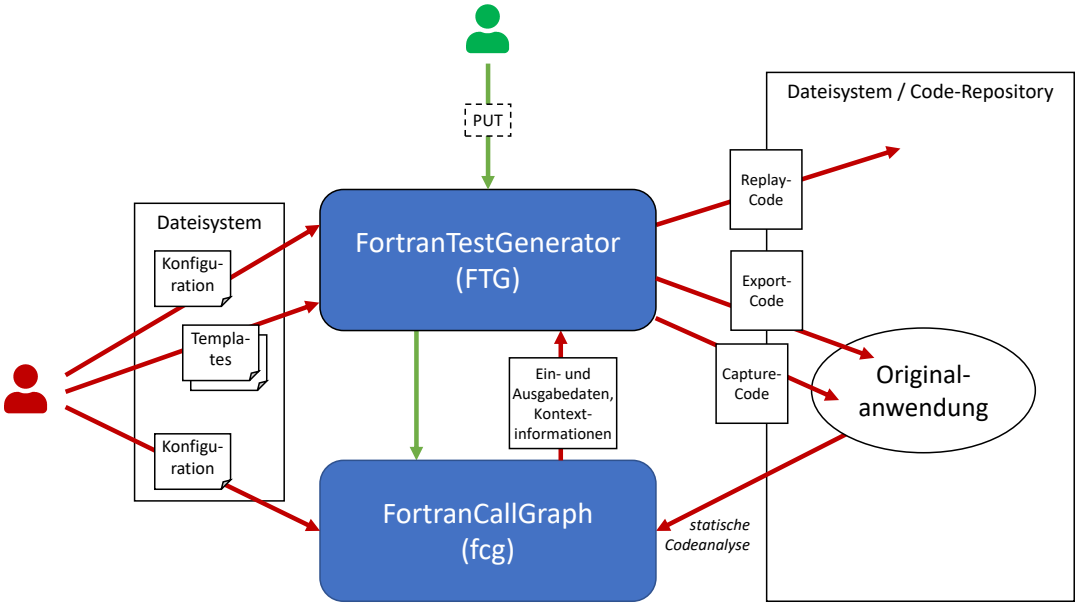


Abbildung 9.1: Grundlegende Architektur des FortranTestGenerator-Systems. Grüne Pfeile: Aufrufe, rote Pfeile: Datenfluss

Sowohl FTG als auch fcg benötigen jeweils eine Konfigurationsdatei, in denen grundlegende Einstellungen anzugeben sind, wie etwa Ordnerpfade der Originalanwendung oder welche Module oder Variablen von der Analyse ausgeschlossen werden sollen etc. FTG benötigt darüber hinaus einen Satz Templates als Vorlagen für den zu generierenden Code. Die Templates erlauben es, den zu generierenden Code an die jeweilige Umgebung anzupassen (siehe auch Abschnitt 9.6). Konfigurationsdateien und Templates werden aus dem Dateisystem gelesen und können von der BenutzerIn oder einer anderen Person (rot) angelegt bzw. angepasst werden. Hierbei kann die in den Abschnitten 8.3.7 und 8.3.8 beschriebene Arbeitsteilung zum Zuge kommen. So könnte es sich bei der BenutzerIn etwa um eine „normale“ EntwicklerIn handeln, die einen Test für eine Prozedur, an der sie gerade arbeitet, erstellen möchte. Konfiguration und Templates könnten wiederum von einem technisch versierten Teammitglied und ggf. mit externer Unterstützung erstellt werden.

9.2 Vorgehen

Entwurf und Implementation von FTG und fcg berücksichtigen neben grundlegenden softwaretechnischen Prinzipien zum einen den anwendungsorientierten Anspruch dieser Arbeit als auch die Umsetzbarkeit in dessen zeitlichem Rahmen. Daraus ergeben sich unter anderem die folgenden Leitlinien:

Iteratives Vorgehen Von Beginn an wurden FTG und fcg immer wieder an realen Beispielen aus dem Klimamodell ICON erprobt. Einzelne Funktionalitäten wurden nach praktischer Relevanz priorisiert, zum Beispiel die Unterstützung einzelner Fortran-Konstrukte danach, wie häufig sie in ICON tatsächlich verwendet werden. Schnittstellen und Architektur wurden kontinuierlich angepasst, um die Erstellung neuer Funktionalitäten zu ermöglichen bzw. zu erleichtern. Zudem flossen Erkenntnisse aus einer Benutzerstudie (siehe auch Abschnitt 11.3) in die laufende Entwicklung mit ein.

Test First Bevor die Unterstützung eines weiteren Fortran-Konstrukts implementiert wurde, wurde in vielen Fällen zunächst ein Test auf Basis eines Minibeispiels, welches das Sprachkonstrukt enthält, erstellt. Anschließend wurde die Unterstützung implementiert, so dass dieser Test erfolgreich bestanden werden kann. Die so erstellten Tests wurden ebenso für Regressionstests als Ergänzung zu den Tests mit dem realen ICON-System verwendet.

Robustheit vor Vollständigkeit Da es im Rahmen dieser Arbeit nicht möglich war, eine Unterstützung für den gesamten Fortran-Sprachumfang zu implementieren, wurde die Funktionsweise der Codeanalyse so entworfen, dass sie auch beim Vorkommen nicht unterstützter Sprachkonstrukte noch so gute Ergebnisse wie möglich liefert. Anstelle eines Programmabbruchs erhält die BenutzerIn so die Möglichkeit, ggf. unvollständige Ergebnisse manuell nachzubessern.

Lieber zu viel als zu wenig Zwar ist es das Ziel, durch Analyse der tatsächlich benötigten Ein- und Ausgabedaten die Menge der aufzuzeichnenden und zu ladenden Daten zu reduzieren, dennoch sollte so gut es geht gewährleistet sein, dass die Reproduktion der PUT-Ausführung identisch mit der Ausführung innerhalb der Originalanwendung ist. Außerdem soll die Ableitung von Testfällen möglichst wenig eingeschränkt werden. Darüber hinaus ist der Aufwand für die EntwicklerIn eine überflüssige Variable aus Capture- und Replaycode zu entfernen geringer als der Aufwand eine zusätzlich benötigte zu ergänzen. Daher wird in Zweifelsfällen eine Variable vorsorglich in die Eingabedaten mit aufgenommen, anstatt sie wegzulassen. Für Ausgabedaten gilt diese Regel nicht so streng, da diese zwar der Validierung dienen, aber nicht für eine erfolgreiche Reproduktion notwendig sind.

9.3 FortranCallGraph

FortranCallGraph (fcg) ist ein Programm zur statischen Analyse von Fortran-Code. Es wird vom FortranTestGenerator verwendet, um für eine gegebene Prozedur eine

Liste ihrer Ein- und Ausgabevariablen zu erstellen und Kontextinformationen abzufragen. Der Name „FortranCallGraph“ stammt daher, dass die erste Funktionalität des Programms die Erstellung eines *Aufrufgraphs (call graph)* für eine gegebene Prozedur war. Dieser Aufrufgraph bildet zudem eine wichtige Grundlage für die weiteren Analysen.

fcg kann über eine API sowie über eine *Kommandozeilenschnittstelle (command-line interface, CLI)* gesteuert werden. Grundlegende Einstellungen bezüglich der Zielanwendung werden in einer Konfigurationsdatei festgelegt. Details zu den Konfigurationsoptionen und dem vollständigen CLI sind in Anhang C zu finden.

Beispiel 9.1: FortranCallGraph-CLI

```
$> ./FortranCallGraph -p tree example_module example_procedure
```

Der Aufruf von fcg mit der Option `-p tree` bewirkt die Ausgabe des Aufrufgraphen der Prozedur `example_procedure` aus dem Modul `example_module` in einer baumähnlichen Darstellung.

```
$> ./FortranCallGraph -a all example_module example_procedure
```

Der Aufruf mit der Option `-a all` bewirkt die Ausgabe einer Liste aller Dummyargumente und globaler Variablen bzw. deren Basiskomponenten, die von `example_procedure` oder einer ihrer Unterprozeduren tatsächlich verwendet werden.

9.4 Aufrufgraph

Um eine Liste der Ein- und Ausgabevariablen einer Prozedur zu erstellen, muss nicht nur die Prozedur selbst, sondern auch jede Unterprozedur untersucht werden. Hierzu erstellt fcg zunächst einen Aufrufgraph, der ausgehend von der PUT sämtliche Aufrufbeziehungen zwischen den beteiligten Prozeduren enthält.

9.4.1 Nutzung von Compilerzwischenprodukten

Fortran ermöglicht das Überladen von Prozeduren, das bedeutet, dass unterschiedliche Prozeduren unter demselben Bezeichner verwendet werden können und der Compiler anhand der Typen der Argumente entscheidet, welche Prozedur tatsächlich aufgerufen wird (siehe auch Abschnitt 2.4.9). Somit lässt sich dies aus dem Quelltext

der Aufrufstelle nicht immer ohne Weiteres erkennen. Hierzu müssten die als Argumente übergebenen Ausdrücke, welche beliebig komplex sein können, analysiert und ihre Typen abgeleitet werden, so wie es der Compiler tut. Dies zu implementieren würde einen großen Aufwand bedeuten.

Daher extrahiert fcg den Aufrufgraphen nicht direkt aus dem Quelltext, sondern aus einer vom Compiler erstellten Zwischenrepräsentation des Anwendungscodes, in welcher die überladenen Prozeduren bereits aufgelöst sind. Implementiert wurde die Extraktion für den vom *GNU Fortran Compiler* (GNU, GFortran) erstellten Assemblercode für *x86-Architekturen*. Die Architektur von fcg erlaubt jedoch die Erweiterung der unterstützten Formate.

Leider führt dieses Vorgehen zu einigen Einschränkungen in der Benutzerfreundlichkeit. Zum einen bedeutet dies, dass die BenutzerIn den Assemblercode bereitstellen muss. Hierzu muss sie die Originalanwendung mit GFortran mit den Optionen `-S -g -O0` oder `-save-temps -g -O0` kompilieren.

- Mit der Option `-S` erzeugt der Compiler anstelle binärer Module Dateien, die den Assemblercode enthalten und beendet dann den Übersetzungsprozess.
- Mit `-save-temps` wird der Übersetzungsprozess zwar zu Ende geführt, jedoch alle Zwischenrepräsentationen in Form von Dateien abgespeichert. Neben dem Assemblercode betrifft dies in der Regel auch den Quelltext nach Verarbeitung durch den Präprozessor.
- Durch Verwendung der Option `-g` werden dem Assemblercode (und dem Binärcode) Debuginformationen hinzugefügt. Dies ist notwendig, um eine Zuordnung der Assemblerbefehle zu den Quelltextzeilen zu gewährleisten.
- `-O0` deaktiviert fast alle Compileroptimierungen, dadurch ist gewährleistet, dass der Assemblercode weitestgehend dem Originalquelltext entspricht.

Trotz `-O0` kann es vorkommen, dass der Compiler einige Codepfade „wegoptimiert“, so dass diese im Assemblercode nicht mehr auftauchen. Dies ist der Fall, wenn die Unerreichbarkeit eines Pfads allzu offensichtlich ist.

Beispiel 9.2: Unerreichbarer Pfad

```
1  LOGICAL, PARAMETER check = .TRUE.
2
3  IF (check) THEN
4    CALL a()
5  ELSE
6    CALL b()
7  END IF
```

> Abhängig vom Wert des PARAMETERS `check`, wird entweder die SUBROUTINE `a` oder die SUBROUTINE `b` aufgerufen. Da der Wert von `check` unveränderlich ist und auf `.TRUE.` festgelegt, kann niemals `b` ausgeführt werden. In diesem Fall würde der Compiler den `THEN`-Pfad der Fallunterscheidung, der den Aufruf von `b` enthält, nicht mit in den Assembler- und/oder Binärcode übernehmen.

Derartige Konstruktionen kommen in der Praxis jedoch sehr selten vor. Zudem ist es akzeptabel, wenn dieser Prozeduraufruf von der Analyse ignoriert wird, da er tatsächlich nie stattfindet (es sei denn jemand ändert den Wert der Konstante `check`). Ein relevanteres Problem, dass dieses Vorgehen mit sich bringt, sind Abweichungen, die sich durch den Einsatz von Präprozessordirektiven zwischen Quelltext und Assemblercode ergeben (siehe auch Abschnitt 9.5.4).

Darüber hinaus ist zu berücksichtigen, dass der von fcg erstellte Aufrufgraph nur statische Aufrufbeziehungen enthält, also solche, die zur Übersetzungszeit bekannt sind. Dynamische Aufrufbeziehungen, die erst zur Laufzeit bestimmt werden, beispielsweise durch den Einsatz von Funktionspointern oder Vererbung werden nicht erfasst. Dies ist eine grundsätzliche Beschränkung der statischen Codeanalyse (siehe auch Abschnitt 8.3.3).

9.4.2 Algorithmus

Der in dem Extraktor für den GNU-x86-Assemblercode implementierte Algorithmus ist recht einfach gehalten. Es existiert eine Datenstruktur `CallGraph`, die eine Menge von Prozeduren (welche anfangs nur die PUT enthält) und eine zunächst leere Menge von Aufrufen verwaltet. Ausgehend von der PUT wird jede Prozedur in ihrer jeweiligen Modul-Assemblerdatei gesucht, indem diese Zeile für Zeile nach dem Beginn der Prozedur durchsucht wird. Ist dieser gefunden, werden die folgenden Zeilen bis zum Ende der Prozedur nach Aufrufbefehlen durchsucht und jeder Aufruf, der eine im Assemblercode vorliegende Prozedur betrifft, der Menge der Aufrufe hinzugefügt. Jede aufgerufene Prozedur, die noch nicht in der Menge der Prozeduren enthalten ist, wird dieser hinzugefügt und der Vorgang rekursiv wiederholt.

9.4.3 Komplexität

Die Komplexität des beschriebenen Algorithmus ergibt sich aus der Anzahl der Prozeduren des zu erstellenden Aufrufgraphs und der Anzahl der Codezeilen in den jeweiligen Moduldateien. Für jede Prozedur wird einmal die jeweilige Datei durchsucht. Sei P die Menge der Prozeduren im Aufrufgraph, $|P|$ die Anzahl der Prozeduren und

g die Größe der Dateien[18]. Für die Komplexität f_A des Aufrufgraph-Algorithmus A ergibt sich somit:

$$f_A = O(g \cdot |P|) \tag{9.1}$$

Grundsätzlich kann man davon ausgehen, dass g und $|P|$ unabhängig sind. Zwar kann man sich Softwarearchitekturen vorstellen, in denen mit wachsender Anwendungsgröße auch die Größe der einzelnen Module wächst, aber in der Praxis kann man davon ausgehen, dass bei wachsender Anwendung eher die Anzahl der Dateien wächst und die Größe der einzelnen Dateien konstant ist bzw. nur innerhalb eines begrenzten Bereichs schwankt. Skalierender Faktor des Algorithmus bleibt somit $|P|$, von welchem die Komplexität linear anhängig ist:

$$f_A = O(|P|) \tag{9.2}$$

9.4.4 Caching

Um unnötige Neuerstellungen von Aufrufgraphen zu vermeiden, bietet fcg die Möglichkeit einen einmal erstellten Aufrufgraphen einer PUT in serialisierter Form abzuspeichern und später wiederzuverwenden. Erst wenn festgestellt wird, dass eine Assemblerdatei neuer ist als der gespeicherte Aufrufgraph, wird dieser neu erstellt. Diese Funktion war vor allem während der Entwicklung von fcg und FTG nützlich, bei der diese zu Testzwecken wiederholt mit derselben PUT aufgerufen wurden. In der Praxis dürften solche Situationen nicht so häufig vorkommen.

9.5 Statische Codeanalyse

Den Kern von FortranCallGraph bildet eine statische Codeanalyse zur Identifikation aller verwendeter Variablen einer PUT. Die Liste dieser Variablen verwendet der FortranTestGenerator für die Generierung von Ein- und Ausgabedaten. Als verwendet gilt jede Variable, wenn diese im Quelltext in irgendeiner Form referenziert wird. fcg unterscheidet dabei nicht zwischen lesendem oder schreibendem Zugriff.

Berücksichtigt werden von fcg dabei die Dummyargumente der PUT, der Rückgabewert, sollte es sich bei der PUT um eine Funktion handeln, sowie Modulvariablen des eigenen und fremder Module. Für Dummyargumente und Rückgabewerte mit primitivem Datentyp geht fcg vereinfachend davon aus, dass diese genau

[18] Ob hier die durchschnittliche oder maximale Dateigröße angenommen wird, spielt für die grundsätzlichen Überlegungen keine Rolle.

deshalb vorhanden sind, da sie auch benötigt werden und fügt diese immer der Liste der verwendeten Variablen hinzu. Für Dummyargumente und Rückgabewerte mit Verbunddatentyp werden dagegen nur die Basiskomponenten gelistet, die von der Prozedur tatsächlich benutzt werden. Modulvariablen werden nur dann gelistet, wenn diese auch verwendet werden. Bei Modulvariablen mit Verbunddatentypen werden ebenfalls nur die tatsächlich benötigten Basiskomponenten als Eingabedaten gewertet.

9.5.1 Verbunddatentypen

Verbunddatentypen können Objektkomponenten, die selbst einen Verbunddatentyp haben, enthalten sowie Basiskomponenten, die einen primitiven Datentyp haben (siehe auch Abschnitt 2.4.5). Basiskomponenten sind die Träger der eigentlichen Nutzdaten einer Datenstruktur, welche letztendlich für Berechnungen herangezogen werden und die Ergebnisse einer Prozedur beeinflussen. Daher werden nur diese von fcg berücksichtigt. Im Folgenden werden die Typen aus Beispiel 2.8 wiederverwendet:

Beispiel 9.3: Basiskomponenten

```
1  TYPE A
2    REAL :: r
3    LOGICAL :: l
4  END TYPE A
5
6  TYPE B
7    INTEGER :: i
8    TYPE(A) :: a
9  END TYPE B
```

```
10 SUBROUTINE example1(a1, b2, result)
11   TYPE(B), INTENT(in) :: a1
12   TYPE(B), INTENT(in) :: b2
13   REAL, INTENT(inout) :: result
14
15   IF (a1%l) THEN
16     result = result + a1%r
17   ELSE
18     result = result + b2%a%r
19   END IF
20 END SUBROUTINE example1
```

Bei einer Analyse der Subroutine `example1` durch fcg würden die Basiskomponenten `a1%l`, `a1%r` und `b2%a%r` als verwendet gewertet, da diese im Quelltext von `example1` referenziert werden, sowie das Dummyargument `result`, da dieses einen primitiven Typ hat.

Zwar ist es auch denkbar, dass neben den Nutzdaten auch strukturelle Informationen Berechnungen beeinflussen, derartige Konstruktionen sind jedoch eher die Ausnahme und werden daher von fcg vernachlässigt.

Beispiel 9.4: Strukturelle Daten

```
1  SUBROUTINE example2(a1, b2, result)
2    TYPE(A), POINTER, INTENT(in) :: a1
3    TYPE(B), TARGET, INTENT(in) :: b2
4    INTEGER, INTENT(inout) :: result
5
6    IF (ASSOCIATED(a1, b2%a)) THEN
7      result = result + 1
8    END IF
9  END SUBROUTINE example2
```

In der SUBROUTINE `example2` wird die Variable `result` nur dann inkrementiert, wenn die Zeigervariable `a1` mit `b1%a` assoziiert ist. Obwohl sie Einfluss auf das Ergebnis nehmen, werden hier weder `a1` noch `b2` oder `b2%a` als verwendet gewertet, da von ihnen in `example2` keine BASISKOMPONENTEN benutzt werden.

Ebenfalls selten zum Einsatz kommen rekursive Datenstrukturen, wie etwa verkettete Listen, da diese nicht so schnell verarbeitet werden können, wie Arrays. Da anhand des Quellcodes statisch nicht erkannt werden kann, wie groß eine solche Datenstruktur zur Laufzeit ist und wie tief damit die Rekursion, werden derartige Strukturen von fcg ebenfalls ignoriert und nicht weitergehend analysiert. Die BenutzerIn erhält jedoch eine Warnung hierüber und kann ggf. selbst Maßnahmen ergreifen.

Beispiel 9.5: Rekursive Datenstrukturen

```
1    TYPE Rek
2      INTEGER :: i, j
3      TYPE(Rek), POINTER :: next
4    END TYPE Rek
```

```
5    FUNCTION rekbeispiel(r)
6
7      TYPE(Rek), INTENT(in) :: r
8      INTEGER :: rekbeispiel
9
10     rekbeispiel = r%i + r%next%j
11
12   END FUNCTION rekbeispiel
```

In der Funktion `rekbeispiel` wird vom DUMMYARGUMENT `r` vom Typ `Rek` nur die KOMPONENTE `r%i` als verwendet gelistet. Der rekursive Zugriff

auf `r%next%j` wird hingegen ignoriert und eine Warnung ausgegeben:

```
WARNING: Ignored access to recursive data structure: r%next%j
```

9.5.2 Analyseverfahren

In fcg ist ein pragmatisches Analyseverfahren implementiert, welches auf direkter Textverarbeitung des Quellcodes basiert. Hierzu kommen im Wesentlichen reguläre Ausdrücke zum Einsatz, mit denen nach bestimmten Mustern im Quellcode gesucht wird. Die Fortran-Syntax eignet sich für diese Art der Analyse insofern gut, als dass die meisten Sprachelemente anhand ihrer Syntax lokal identifiziert werden können, d.h. ohne viel Kontextwissen durch Analyse einer einzigen Quelltextzeile. Ein paar wenige Ausnahmen gibt es hiervon. So sind beispielsweise Funktionsaufrufe syntaktisch nicht von Zugriffen auf Arrayvariablen zu unterscheiden, da für beide Runde Klammern verwendet werden. Zudem ist es in Fortran erlaubt, Schlüsselwörter wie `if`, `call` oder `subroutine` als Bezeichner zu verwenden. Da von Letzterem in der Praxis jedoch normalerweise kein Gebrauch gemacht wird, ignoriert fcg diese Problematik, d.h. das Verhalten ist in solchen Fällen nicht definiert. Darüber hinaus geht fcg davon aus, dass der zu analysierende Quellcode korrektes Fortran ist und auch von einem Compiler akzeptiert würde.

Dieses Vorgehen der direkten Textverarbeitung unterscheidet sich von anderen Ansätzen, in denen zunächst ein Modell des Quellcodes erstellt wird, wie etwa ein *abstrakter Syntaxbaum (AST)* (siehe z.B. Baxter u. a., 1998; Neamtiu, Foster und Hicks, 2005; Zoller und Schmolitzky, 2012; Kim u. a., 2016). Ein AST wird in der Regel mit Hilfe eines Parsers und semantischer Analyse, wie sie auch von Compilern verwendet werden, erstellt. Anstatt den Quelltext direkt zu untersuchen, wird die Analyse dann auf diesem Modell durchgeführt. Der Vorteil dieses Vorgehens ist, dass das Modell bereits vor der Analyse mit semantischen Informationen angereichert wird, auf die dann während der Analyse vereinfacht zugegriffen werden kann. Zum Beispiel könnte ein Modellelement, das einen Ausdruck repräsentiert, bereits Informationen über den Typ des Ausdrucks enthalten. Auf diese Weise lässt sich, sobald die Erstellung des Modells realisiert ist, die Analyse selbst einfacher implementieren. So lassen sich auf Grundlage eines Modells verschiedene Arten von Analysen, für verschiedene Fragestellungen, mit geringerem Aufwand realisieren, als wenn jede Analyse, wie in dieser Arbeit, seine eigene, gezielte Quellcodeverarbeitung beinhaltet.

Der Nachteil dieses Vorgehens ist, dass die Verarbeitung von Quelltext, der nicht der vom Parser erwarteten Syntax entspricht, in der Regel zum vollständigen Abbruch

des Parsing- und Analyseprozesses führt. So basiert beispielsweise die in KGEN (siehe Abschnitt 8.5.6) realisierte Codeanalyse auf einem AST, der von einem Parser erzeugt wird, der der Python-Bibliothek *F2PY* (Peterson, 2009; f2py.com) entnommen ist. Dieser Parser unterstützt jedoch nicht alle Konstrukte der Fortran-Versionen 2003 und aufwärts. Dies führte beispielsweise bei Versuchen, KGEN auf ICON anzuwenden, dazu, dass alle Versuche, einen Kernel zu isolieren, fehlschlugen, da der Prozess bereits beim Parsing abbrach. Andere Anwender berichteten ähnliches.

Ein derartiges Verhalten kann zu Frustration bei den BenutzerInnen führen und verhindern, dass ein Softwarewerkzeug überhaupt zum Einsatz kommt. Die in fcg implementierte Analyse, die dem Prinzip *Robustheit vor Vollständigkeit* folgt (Abschnitt 9.2), verarbeitet nur die Quellcodezeilen, die „auf den ersten Blick“ interessant erscheinen, und lässt alle anderen außer Acht. Nicht unterstützte Sprachkonstrukte führen so nicht zwangsläufig zum Totalabbruch.

Viele Werkzeuge zur statischen Codeanalyse basieren auf der *LLVM Compiler Infrastructure* bzw. der *LLVM Intermediate Representation* (Lattner und Adve, 2004), welche besondere Unterstützung für statische Analysen bietet (siehe z.B. Castro u. a., 2015; Droste, Kuhn und Ludwig, 2015; Grech u. a., 2015; Cassez u. a., 2017). Diese hat den Vorteil, dass damit relativ einfach verschiedene Arten der Analyse realisiert werden können und zum anderen, dass mehrere Programmiersprachen gleichzeitig unterstützt werden können, sofern für diese ein LLVM-kompatibler Compiler existiert. Entscheidender Nachteil hierbei ist jedoch, dass diese Werkzeuge von der Verfügbarkeit eines entsprechenden Compilers auf dem jeweiligen Zielsystem abhängig sind. Diese Festlegung auf eine bestimmte Technologie steht im Widerspruch zum anwendungsorientierten Anspruch. Zudem ist erst seit kurzem ein LLVM-kompatibler Fortran-Compiler verfügbar (siehe auch GitHub, Flang).

Zwar ist auch fcg von der Verfügbarkeit des GNU Fortran Compilers abhängig, jedoch gehört dieser zum einen zur Standardausstattung fast jedes Rechenzentrums und Entwickler-PCs und zum anderen bezieht sich diese Abhängigkeit nur auf die Erstellung des Aufrufgraphen. Mit relativ wenig Aufwand ließen sich hierfür auch Extraktoren für andere Formate erstellen. Die eigentlich Codeanalyse zum Auffinden verwendeter Variablen ist jedoch deutlich aufwendiger und lässt sich nicht ohne weiteres für andere Formate neu implementieren. Daher bietet sich hierfür eine technologieneutrale Analyse des Originalquellcodes am besten an.

9.5.3 Quellcodenormalisierung

Der erste Schritt der Quellcodeverarbeitung in fcg besteht aus der Normalisierung des Quellcodes, um die anschließende Analyse zu erleichtern. Hierzu werden folgende Schritte durchgeführt:

Kommentare Kommentare, welche in Fortran mit `!` eingeleitet werden, werden für die Analyse nicht benötigt und daher entfernt.

Präprozessordirektiven Präprozessordirektiven erkennt man am `#` am Zeilenanfang. Alle derartigen Zeilen werden entfernt.

Mehrzeilige Anweisungen In Fortran entspricht normalerweise eine Quelltextzeile einer Anweisung. Mehrzeilige Anweisungen sind jedoch möglich. Ein `&` am Ende einer Zeile signalisiert, dass die Anweisung in der Folgezeile fortgesetzt wird. Derartige mehrzeilige Anweisungen werden in Rahmen der Normalisierung zu einer Zeile zusammengefügt.

Stringliterale Da die Inhalte von Strings für die Analyse uninteressant sind, Stringliterale jedoch Text enthalten können, der ggf. wie ein Fortran-Sprachkonstrukt aussieht, werden sämtliche Stringliterale auf den leeren String (`''` bzw. `""`) verkürzt.

Leerzeichen Einige syntaktisch nicht notwendige Leerzeichen werden entfernt. Dies betrifft zum Beispiel Leerzeichen am Zeilenanfang oder -ende, Leerzeichen vor öffnenden oder schließenden Klammern, oder um Kommas oder Operatoren. Außerdem werden mehrfache Leerzeichen hintereinander zu einem reduziert.

Leerzeilen Auch leere Zeilen werden entfernt.

Beispiel 9.6: Quellcodenormalisierung

Originalquellcode

```
1  ! Beispielmodul
2  MODULE example
3
4    USE modulewithlongidentifiers, ONLY : parameterwithlongidentifier, &
5      functionwithlongidentifier, typewithlongidentifier
6  #ifdef __OTHER_ENABLED__
7    USE other
8  #endif
9
10 CONTAINS
11
12   SUBROUTINE   subr(a, result)
13     TYPE(typewithlongidentifier), INTENT(in) :: a ! In-Variable
14     INTEGER, INTENT(out) :: result ! Out-Variable
15
16     WRITE (*,*) 'This subroutine subr does not use a%component2'
17     result = functionwithlongidentifier ( parameterwithlongidentifier, a )
18
19   END SUBROUTINE
20
21 END MODULE example
```

Normalisierter Quellcode

```
22 MODULE example
23 USE modulewithlongidentifiers,ONLY:parameterwithlongidentifier,
       ↪ functionwithlongidentifier, typewithlongidentifier
24 USE other
25 CONTAINS
26 SUBROUTINE subr(a,result)
27 TYPE(typewithlongidentifier),INTENT(in)::a
28 INTEGER,INTENT(out)::result
29 WRITE(*,*) ''
30 result=functionwithlongidentifier(parameterwithlongidentifier,a)
31 END SUBROUTINE
32 END MODULE example
```

9.5.4 Präprozessordirektiven

Einige Klimamodelle enthalten in ihrem Code sehr viele Präprozessordirektiven (siehe auch Abschnitt 2.5.1; Méndez, Tinetti und Overbey, 2014), durch die der Quellcode vor dem Kompilieren verändert wird. fcg entfernt derartige Direktiven während der Quellcodenormalisierung, das bedeutet, dass der Quellcode so analysiert wird als seien diese nicht existent. In vielen Fällen beeinträchtigt dies die Analyse nicht gravierend, es kann hierdurch jedoch auch zu fehlerhaften Ergebnissen kommen oder die Analyse ganz verhindert werden, etwa wenn der Quellcode ohne Berücksichtigung der Präprozessordirektiven kein gültiger Fortran-Code ist.

Beispiel 9.7: Ungültiger Code ohne Präprozessor

```
1   y = x + &
2 #if __ENABLE_FOO__
3       & foo()
4 #else
5       & bar()
6 #endif
```

Abhängig davon, ob bei Aufruf des Präprozessors die Option `__ENABLE_FOO__` gesetzt wurde, soll hier entweder das Ergebnis der Funktion `foo` oder das Ergebnis der Funktion `bar` zu `x` addiert werden. Ignoriert man jedoch die Präprozessordirektiven ergibt sich (ohne weitere Normalisierung) folgender fehlerhafter Fortran-Code:

```
7 y = x + &
8     & foo()
9     & bar()
```

Hinzu kommt, dass der Assemblercode, der für die Erstellung des Aufrufgraphen verwendet wird, aus bereits vorverarbeitetem Code erzeugt wird. Hierdurch kann es zu Inkonsistenzen zwischen dem Aufrufgraphen und dem analysierten Quellcode kommen. Sind derartige Probleme zu erwarten, hat die BenutzerIn die Möglichkeit anstelle des Originalquellcodes einen vom Präprozessor bereits vorverarbeiteten Code analysieren zu lassen. Mit dem GNU Compiler lässt sich dieser beispielsweise mit der Option `-save-temps` erzeugen.

9.5.5 Kontextinformationen

Die eigentliche Analyse der Variablenverwendung benötigt einige Kontextinformationen, die zuvor in einem gesonderten Durchgang ermittelt werden. Neben dem Aufrufgraph werden Sammlungen der im Quellcode deklarierten INTERFACES und Verbunddatentypen erstellt. Die INTERFACE-Informationen werden zusammen mit dem Aufrufgraph benötigt, um die Zuordnung von Aufrufen überladener Prozeduren und tatsächlich aufgerufenen Prozeduren vornehmen zu können (siehe auch Abschnitt 2.4.9). Die Deklarationen der Verbunddatentypen werden ausgelesen, um die Typen der einzelnen KOMPONENTEN zu erfahren und über welche TYPGEBUNDENEN PROZEDUREN ein Verbunddatentyp verfügt (siehe auch Abschnitt 2.4.5). Um die Sammlungen der INTERFACES und Verbunddatentypen zu erstellen, werden alle Module, zu denen das Hauptmodul, welches die PUT enthält, eine direkte oder indirekte Abhängigkeit per USE-Anweisung hat, nach entsprechenden Deklarationen durchsucht.

9.5.6 Grundlegender Algorithmus

fcg unterscheidet zwischen drei Arten von Variablen, deren Verwendung analysiert werden kann:

- DUMMYARGUMENTE
- Ergebnisvariablen von FUNKTIONEN
- MODULVARIABLEN

Im Folgenden wird zunächst beschrieben, wie der Quellcode nach Verwendungen von DUMMYARGUMENTEN durchsucht wird. Für Ergebnisvariablen von FUNKTIONEN wird auf die gleiche Weise verfahren. Die Unterschiede in der Vorgehensweise bei MODULVARIABLEN werden im Abschnitt 9.5.7 beschrieben. Anstelle einer formalen Beschreibung wird das Vorgehen anhand eines Beispiels erläutert.

Beispiel 9.8: Suche nach verwendeten Variablen

Die zu analysierende PUT sei die SUBROUTINE `example` aus dem Modul `exmod`, dessen Quellcode sich in der Datei exmod.f90 befindet.

```
1  SUBROUTINE example(a,b,c,d)
2   ⋮
3  END SUBROUTINE example
```

Zunächst wird die SUBROUTINEN-Definition von `example` in der Datei exmod.f90 gesucht. Die Definition beginnt mit einer Zeile mit dem Schlüsselwort `SUBROUTINE` gefolgt von dem Namen der Prozedur und einer Argumentliste und endet mit einer Zeile mit den Schlüsselwörtern `END SUBROUTINE`.

```
4  SUBROUTINE example(aa,bb,cc,dd)
5  TYPE(A),INTENT(in)::aa
6  TYPE(B),INTENT(in)::bb(:)
7  REAL,INTENT(in)::cc(:,:)
8  INTEGER,INTENT(out)::dd
9  TYPE(A)::p
10 INTEGER::i
11  ⋮
```

Durch Parsen der Variablenspezifikationen in den ersten Zeilen der Prozedur wird eine Liste aller Variablen der Prozedur mit ihren Eigenschaften, insbesondere des Typs, erstellt. Durch Auslesen der Argumentliste im Prozedurkopf werden die Variablen identifiziert, bei denen es sich um DUMMYARGUMENTE handelt, hier also `aa`, `bb`, `cc` und `dd`. Allein durch das Schlüsselwort `INTENT` ließen sich ARGUMENTE nicht sicher identifizieren, da dieses optional ist.

Alle DUMMYARGUMENTE mit primitivem Datentyp, hier `cc` und `dd`, werden sofort der Menge der verwendeten Variablen hinzugefügt. Für jedes DUMMYARGUMENT mit Verbunddatentyp, hier `aa` und `bb`, wird der Quellcode der Prozedur nach Referenzierungen der Variable gesucht. Gesucht werden diese mit Hilfe eines regulären Ausdrucks, der den Namen der Variable enthält. Wird auf diese Weise die Referenzierung einer KOMPONENTE der Variable gefunden, wird mit Hilfe der im Vorfeld gesammelten Typinformationen überprüft, ob es sich um eine BASISKOMPONENTE handelt. Ist dies der Fall, wird diese in die Liste der verwendeten Variablen aufgenommen.

```
12 IF (aa%l) THEN
13 d=d+1
14 END IF
15 CALL other1(bb(d)%a%r)
```

Hier werden die KOMPONENTEN `aa%l` und `bb%a%r` als verwendet erkannt.

Wird auf diese Weise die Referenzierung einer Variable ohne Komponentenanhang gefunden oder die Referenzierung einer Objektkomponente, sind je nach Kontext weitere Maßnahmen nötig. Die wichtigsten werden im Folgenden erläutert.

Prozeduraufrufe

Wird ein gesuchtes Dummyargument oder eine seiner Objektkomponente als Argument für den Aufruf einer anderen Prozedur verwendet, wird das entsprechende Dummyargument der aufgerufenen Prozedur auf die gleiche Weise untersucht, wie hier beschrieben. Alle gefundenen Referenzierungen von Basiskomponenten des Dummyarguments werden als Referenzierung der entsprechenden Basiskomponente der als Argument übergebenen Variable gewertet.

Prozeduraufrufe werden an der Klammersyntax, d.h. Bezeichner gefolgt von runden Klammern erkannt, wobei nicht jedes dieser Konstrukte einen Prozeduraufruf in diesem Sinne darstellt. So kann es sich auch um Zugriffe auf Arrayvariablen handeln, die über die gleiche Syntax verfügen, oder auf Aufrufe von in Fortran eingebauten Prozeduren oder sonstigen, für die der Quellcode nicht vorliegt. Mit Hilfe des Aufrufgraphs wird daher überprüft, ob es sich bei dem gefundenen Konstrukt tatsächlich um eine zu analysierende Prozedur handelt. Ebenso wird beim Aufruf von überladenen Prozeduren mit Hilfe des Aufrufgraphs und den zuvor erstellten Interface-Informationen herausgefunden, welche Prozedur tatsächlich aufgerufen wird.

Beispiel 9.9: Prozeduraufruf

```
1 CALL other2(bb,d)
```

Das gesuchte Dummyargument `bb` wird als Argument an die Subroutine `other2` übergeben. Somit muss auch diese Prozedur analysiert werden, indem alle Referenzierungen des Dummyarguments `argB`, an welches `bb` gebunden wurde, gesucht werden.

```
2 SUBROUTINE other2(argB,argD)
3 TYPE(B),INTENT(in)::argB
4 INTEGER,INTENT(inout)::argD
5 argD=argD+argB%i
6 END SUBROUTINE other2
```

Da in `other2` die Basiskomponenten `argB%i` referenziert wird (Zeile 5), gilt `bb%i` als verwendet.

Zuweisungen

Steht ein gesuchtes Dummyargument oder eine seiner Objektkomponenten auf der rechten Seite einer Anweisung *(right hand side, RHS)*, werden die der Anweisung folgenden Quellcodezeilen nach Referenzierungen der Variable, die in der Anweisung auf der linken Seite steht *(left hand side, LHS)* durchsucht. Alle gefundenen Referenzierungen von Basiskomponenten der LHS-Variable werden als Referenzierung der entsprechenden Basiskomponente der RHS-Variable gewertet.

Zuweisungen werden am =-Operator erkannt.

Beispiel 9.10: Zuweisung

```
1 p=bb%a
2 cc(:)=p%r
```

Hier wird zunächst festgestellt, dass der Variable `p` der Wert der Objektkomponente `bb%a` zugewiesen wird. Daraufhin werden die nachfolgenden Zeilen nach Referenzierungen von `p` durchsucht und festgestellt, dass `p%r` verwendet wird. Damit gilt auch `bb%a%r` als verwendet.

Findet eine Zuweisung innerhalb einer Schleife statt, müsste die Suche nach der LHS-Variable streng genommen nicht in der auf die Anweisung folgenden Zeile beginnen, sondern am Beginn der (äußersten) Schleife, da grundsätzlich der gesamte Schleifenrumpf nach der Zuweisung erneut ausgeführt werden könnte. In der aktuellen Version von fcg ist dies jedoch noch nicht berücksichtigt.

Zuweisungen an Out-Dummyargumente oder Ergebnisvariablen

Findet innerhalb einer aufgerufenen Prozedur eine Zuweisung eines gesuchten Dummyarguments oder eines seiner Objektkomponenten zu einem Out-Argument statt, wird diese wie eine Zuweisung zum entsprechenden Argument an der Aufrufstelle behandelt.

Beispiel 9.11: Zuweisung zu Out-Dummyargument

```
1 CALL getA(bb, p)
2 cc(:)=p%r
```

Das gesuchte Argument `bb` wird an die Subroutine `getA` übergeben (Zeile 1), weshalb diese nach Referenzierungen des Dummyarguments `argIn` durchsucht wird.

```
3  SUBROUTINE getA(argIn, argOut)
4  TYPE(B),INTENT(in)::argIn
5  TYPE(A),INTENT(out)::argOut
6  argOut=argIn%a
7  END SUBROUTINE getA
```

Dabei wird festgestellt, dass die Objektkomponente `argIn%a` dem Out-Dummyargument `argOut` zugewiesen wird (Zeile 6). An der Aufrufstelle wird dies daraufhin wie eine Zuweisung zu `p=bb%a` behandelt und entsprechend in den Folgezeilen nach Referenzierungen von `p` gesucht. Dabei wird die Referenzierung der Komponente `p%r` gefunden (Zeile 2). Somit gilt `bb%a%r` als verwendet.

Handelt es sich bei der aufgerufenen Prozedur um eine Funktion, müssen auch Zuweisungen zur Ergebnisvariable auf die Weise behandelt werden.

Beispiel 9.12: Zuweisung zu Funktionsergebnis

```
1  p = getAOfB(bb)
2  cc(:)=p%r
```

```
3  FUNCTION getAOfB(argIn)
4  TYPE(B),INTENT(in)::argIn
5  TYPE(A)::getAOfB
6  getAOfB=argIn%a
7  END FUNCTION getAOfB
```

Wegen des Setzens von `argIn%a` als Funktionsergebnis von `getAOfB` (Zeile 6) wird hier die Zuweisung des Funktionsergebnisses an `p` (Zeile 1) wie eine Zuweisung von `bb%a` behandelt. Durch die Referenzierung von `p%r` (Zeile 2) gilt auch in diesem Beispiel `bb%a%r` als verwendet.

9.5.7 Modulvariablen

Die Suche nach verwendeten Modulvariablen funktioniert auf ähnliche Weise wie die Suche nach verwendeten Komponenten von Dummyargumenten mit Verbunddatentyp. Die Ermittlung der Variablen, nach denen zu suchen ist, gestaltet sich jedoch anders. Hierzu müssen zunächst sämtliche Module, die Prozeduren enthalten, die Teil des Aufrufgraphs sind, zum einen nach Deklarationen von Modulvariablen und zum anderen nach `USE`-Anweisungen, welche Variablen anderer Module

importieren, durchsucht werden. Für jede Prozedur des Aufrufgraphs entsteht so eine Liste von Modulvariablen, die potenziell benutzt werden könnten.

Nach diesen Variablen wird dann auf die gleiche Weise gesucht, wie oben beschrieben. Ein Unterschied ist dabei jedoch, dass die Suche nicht zwangsläufig in der PUT beginnt, sondern jeweils in der Prozedur, zu dessen Modul die Variable gehört bzw. das die jeweilige Variable importiert. Außerdem werden Modulvariablen mit primitivem Datentyp nicht automatisch als verwendet angesehen, stattdessen wird auch für diese nach Referenzierungen gesucht. Werden zudem Modulvariablen oder deren Objektkomponenten von einer Prozedur unterhalb der PUT per Out-Argument oder als Funktionsergebnis zurückgegeben, werden sämtliche Aufrufe dieser Prozedur im Aufrufgraph gesucht und von diesen aus nach Weiterverwendungen des Out-Arguments bzw. des Funktionsergebnisses gesucht.

9.5.8 Komplexität

Grundsätzlich wird bei der oben beschriebenen Vorgehensweise für jede zu suchende Variable jede Quelltextzeile der Prozeduren des Aufrufgraphs einmal durchsucht. Zwar wird nicht für jede Variable der gesamte Aufrufgraph traversiert, auch beginnt nicht für jede Modulvariable die Suche bei der Wurzel des Aufrufgraphs, dies kann für die Betrachtung der Komplexität jedoch vernachlässigt werden. Zusätzlich werden für jede Zuweisung noch einmal sämtliche Quellcodezeilen unterhalb der Zuweisung durchsucht, ggf. einschließlich der Prozeduren, an denen die LHS-Variable der Zuweisung übergeben wird.

Sei

- P die Menge der Prozeduren im Aufrufgraph und $|P|$ deren Anzahl,
- $s(p)$ die Größe einer Prozedur $p \in P$, wobei die Größe der Anzahl der Quellcodezeilen entspricht,
- $S = \sum_{p \in P}^{p} s(p)$ die Gesamtgröße der Prozeduren des Aufrufgraphs,
- A die Menge der Dummyargumente der PUT mit Verbunddatentyp und $|A|$ deren Anzahl,
- M die Menge der Modulvariablen, die von den Prozeduren des Aufrufgraphs potenziell benutzt werden könnten, und $|M|$ deren Anzahl,
- Z die Menge der Zuweisungen zu zu suchenden Variablen oder deren Objektkomponenten und $|Z|$ deren Anzahl.

Für die Komplexität f_V des Algorithmus zur Suche von verwendeten Variablen und Komponenten ergibt sich:

$$f_V = O((|A| + |M| + |Z|) \cdot S) = O(|A| \cdot S) + O(|M| \cdot S) + O(|Z| \cdot S) \tag{9.3}$$

Bei der Anzahl der Dummyargumente der PUT kann man davon ausgehen, dass diese nicht unbegrenzt wächst und somit bei der Betrachtung der Komplexität vernachlässigt werden kann. Die Anzahl der Modulvariablen sowie die Anzahl der Zuweisungen dürfte jedoch proportional zu S sein. Für die Komplexität ergibt sich somit eine quadratische Abhängigkeit zu S:

$$f_V = O(S) + O(S^2) + O(S^2) = O(S^2) \tag{9.4}$$

Dies ist nicht ideal. Besser wäre es und vermutlich auch möglich, wenn jede Quellcodezeile nur einmal gelesen werden müsste, jedoch zeigt auch das hier beschriebene Verfahren in der Praxis akzeptable Laufzeiten (siehe auch Abschnitt 11.2.2).

9.6 FortranTestGenerator

Der FortranTestGenerator (FTG) ist ein Programm, welches mit Hilfe der von fcg gelieferten Informationen Capture-, Replay- und Exportcode für eine gegebene PUT erzeugen kann. Es wird über eine Kommandozeilenschnittstelle aufgerufen. Grundlegende Einstellungen bezüglich der Zielanwendung werden wie bei fcg in einer Konfigurationsdatei festgelegt. So lässt sich u.a. auch festlegen, ob Capture- und Exportcode in vom Präprozessor vorverarbeiteten Code oder in die originalen Quellcodedateien eingefügt werden sollen. FTG verwendet Templates, mit denen der generierte Code an die jeweilige Umgebung angepasst werden kann. Details zu allen Konfigurationsoptionen und dem vollständigen CLI sind in Anhang D zu finden.

Beispiel 9.13: FortranTestGenerator-CLI

```
$> ./FortranTestGenerator -c example_module example_procedure
```

Der Aufruf von ftg mit der Option `-c` bewirkt die Erzeugung von Capture- und Exportcode für die Prozedur `example_procedure` in dem Modul `example_module`.

```
$> ./FortranTestGenerator -r example_module example_procedure
```

Mit der Option `-r` wird Replaycode, d.h. ein Testprogramm, erzeugt, sowie der notwendige Exportcode.

```
$> ./FortranTestGenerator -cr example_module example_procedure
```

Verschiedene Optionen lassen sich auch kombinieren. Mit `-cr` werden Capture-, Replay- und Exportcode erzeugt.

```
$> ./FortranTestGenerator -a
```

Die Option `-a` sorgt dafür, dass sämtlicher Capturecode aus der Anwendung wieder entfernt wird.

9.6.1 Cheetah

Der Kern von FTG ist das Templatesystem, mit dessen Hilfe Capture-, Replay- und Exportcode generiert wird. Zum Einsatz kommt hier die Python-basierte Template-Engine *Cheetah* (Tavis Rudd und Bicking, 2002)[19]. Templates sind textuelle Vorlagen, die die Grundstruktur eines Dokuments vorgeben, in der mit Hilfe von Platzhaltervariablen und Kontrollstrukturen bestimmte Teile dynamisch generiert werden. Ein typischer, aber nicht alleiniger Einsatzzweck von Templates ist die Erzeugung dynamischer Webseiten.

Beispiel 9.14: Cheetah-Template (Webseite)

veranstaltunga.tmpl

```
1  <html>
2  <body>
3  <h1>Teilnehmer</h1>
4  #if $teilnehmer
5  <ul>
6  #for $name in $teilnehmer
7    <li>$name</li>
8  #end for
9  </ul>
10 #end if
11 </body>
12 </html>
```

[19] Leider wurde während der Arbeit an FTG die Weiterentwicklung des ursprünglichen Cheetah eingestellt, jedoch ist inzwischen mit Cheetah3 (GitHub, Cheetah3) ein Fork entstanden, der weitergepflegt wird.

Das Beispiel zeigt ein Cheetah-Template einer Webseite, auf der die Namen von Teilnehmer einer Veranstaltung ausgegeben werden. Cheetah-Schlüsselworte wie hier `#if` oder `#for` beginnen immer mit dem #-Symbol, Platzhalter mit `$`. Das Template verwendet die Platzhaltervariable `$teilnehmer`, in der eine Liste von Namen erwartet wird. Ist diese gesetzt (Prüfung in Zeile 4), werden in einer Schleife alle Elemente von `$teilnehmer` ausgegeben (Zeilen 6–8).

Sei dieses Template in einer Datei namens veranstaltunga.tmpl gespeichert, dann ließe es sich in Python beispielsweise folgendermaßen verwenden:

```
13 from Cheetah.Template import Template
14 nameSpace = {'teilnehmer':['Katrin', 'Christian', 'Mattis', 'Hanna']}
15 t = Template(file='veranstaltunga.tmpl', searchList=[nameSpace])
16 print t
```

In diesem Fall erhielte man die folgende Ausgabe:

```
17 <html>
18 <body>
19 <h1>Teilnehmer</h1>
20 <ul>
21   <li>Katrin</li>
22   <li>Christian</li>
23   <li>Mattis</li>
24   <li>Hanna</li>
25 </ul>
26 </body>
27 </html>
```

Tatsächlich erzeugt Cheetah aus dem Template eine Python-Klasse, welches für die gewünschte Ausgabe sorgt. Ein Cheetah-Template kann daher auch als eine Art Python-Klasse, formuliert in anderer Syntax, angesehen werden. So lassen sich mit dem Schlüsselwort `#def` auch Methoden definieren, die an anderer Stelle im Template aufgerufen werden.

Beispiel 9.15: Cheetah-Template mit Methoden

veranstaltungb.tmpl

```
1 #def listNames($names)
2 <ul>
3 #for $name in $names
4   <li>$name</li>
5 #end for
6 </ul>
7 #end def
8 <html>
9 <body>
```

```
10 <h1>Teilnehmer</h1>
11 #if $teilnehmer
12 $listNames($teilnehmer)
13 #end if
14 </body>
15 </html>
```

In diesem Beispiel wird der Teil, der die Namen der Teilnehmer listet, als Methode `listNames` definiert (Zeilen 1–7). Wie eine Platzhaltervariable wird diese mit vorangestelltem `$` im Hauptteil des Templates eingesetzt (Zeile 12).

Mit dem Schlüsselwort `#extends` kann ein Template auch von einem anderen erben und Methoden überschreiben.

Beispiel 9.16: Cheetah-Template mit Vererbung

veranstaltungc.tmpl

```
1  #extends veranstaltungb
2  #def listNames($names)
3  <table>
4  #for $name in $names
5    <tr>
6      <td>$name</td>
7    </tr>
8  #end for
9  </table>
10 #end def
```

Dieses Template erbt von dem in Beispiel 9.15 definierten Template veranstaltungb.tmpl (Zeile 1). Es erzeugt daher eine ähnliche Webseite wie die vorherigen. Da die Methode `listNames` überschrieben wird (Zeilen 2–10), wird die Liste der Teilnehmer jedoch nicht in einer Liste mit Aufzählungszeichen (`<ul>...</ul>`) dargestellt, sondern in einer Tabelle (`<table>...</table>`).

9.6.2 Templates

Ein Template für die Codegenerierung mit FTG besteht aus acht Teilen (*Template Parts*), welche als Vorlage für verschiedene Codeabschnitte dienen. Hierzu definiert FTG eine Templateoberklasse `FTGTemplate`, die acht Methoden vorgibt, von denen jede einem Codeabschnitt entspricht. Alle Methoden in `FTGTemplate` sind leer, d.h. sie enthalten noch keinen Templatecode. Ein echtes Template für die Codegenerierung muss von dieser Templateoberklasse erben und die acht Methoden implementieren. Anstelle direkt von `FTGTemplate` zu erben, kann eine Templateklasse auch von

einer anderen Unterklasse von `FTGTemplate` erben und nur einen Teil der Methoden überschreiben. FTG enthält bereits verschiedene Standardtemplates, die direkt benutzt oder beerbt werden können.

Die acht Methoden/Codeabschnitte sind:

captureAfterUse Erster Abschnitt des Capturecodes. Wird in das Modul der PUT unterhalb der letzten `USE`-Anweisung eingefügt. Dient zum Einfügen weiterer USE-Anweisung zum Import von Variablen und Prozeduren, die für das Capture benötigt werden.

captureBeforeContains Zweiter Abschnitt des Capturecodes. Wird in das Modul der PUT oberhalb des `CONTAINS`-Schlüsselworts eingefügt. Dient der Deklaration von Modulvariablen, die für das Capture benötigt werden.

captureAfterLastSpecification Dritter Abschnitt des Capturecodes. Wird in die PUT zwischen der letzten Variablen-Deklaration und der ersten Anweisung eingefügt. Ist für die Deklaration weiterer lokaler Variablen, die für das Capture gebraucht werden, vorgesehen, sowie für den Code für das Aufzeichnen der Eingabevariablen.

captureBeforeEnd Vierter Abschnitt des Capturecodes. Wird in die PUT zwischen der letzten Anweisung und dem `END`-Schlüsselwort eingefügt. Ist für den Code für das Aufzeichnen der Ausgabevariablen vorgesehen.

captureAfterSubroutine Fünfter Abschnitt des Capturecodes. Wird in das Modul der PUT direkt nach der PUT eingefügt. Hier können zusätzliche Prozeduren definiert werden, die in den anderen Abschnitten aufgerufen werden, etwa zum Aufzeichnen der Daten.

exportAfterUse Erster Abschnitt des Exportcodes. Dieser wird in alle Module, aus denen private Modulvariablen benötigt werden, unterhalb der letzten `USE`-Anweisung eingefügt. Dient zum Einfügen weiterer USE-Anweisung, die in den verwendeten Modulen ggf. benötigt werden.

exportBeforeContains Zweiter Abschnitt des Exportcodes. Wird in alle Module, aus denen private Modulvariablen benötigt werden, oberhalb des `CONTAINS`-Schlüsselworts eingefügt. Dient u.a. der Ausgabe einer `PUBLIC`-Anweisung, durch die bisher private Modulvariablen exportiert werden.

replay Replaycode. Hieraus wird das Testtreiberprogramm oder -modul erzeugt.

Innerhalb des Templatecodes kann auf die *FTG Template API* zugegriffen werden. Diese enthält Platzhaltervariablen und -methoden, mit denen auf die von fcg gelieferten Informationen zugegriffen werden kann. Hierzu gehören beispielweise die Listen der Ein- und Ausgabevariablen oder Name und Dummyargumente der PUT.

Der Codegenerierungsprozess beginnt zunächst mit dem Aufruf der fcg-API zur Analyse der PUT. Aus den gesammelten Daten wird ein Namespace-Objekt, welches die Template API realisiert, erzeugt, ähnlich der `nameSpace`-Variable in Beispiel 9.14, nur deutlich komplexer. Mit diesem Namespace-Objekt und den Templates werden nach und nach die verschiedenen Codeabschnitte erzeugt und in die jeweiligen Module bzw. im Fall des Replaycodes in eine neue Datei eingefügt.

Abbildung 9.2 zeigt die Einfügepunkte der Capture-Template-Parts innerhalb des PUT-Moduls. Der Aufbau der einzelnen Template Parts wird in Abschnitt 9.6.5 und Anhang D.3 anhand eines Beispiel-Templates näher betrachtet.

9.6.3 Standardtemplates

In FTG sind bereits verschiedene Standardtemplates verfügbar, die sich in einigen Details unterscheiden. So gibt es generische Templates oder Templates, die speziell für die Verwendung mit bestimmten Klimamodellen (ICON, NICAM) vorgesehen sind. Zudem gibt es Templates mit und ohne MPI-Unterstützung. Templates mit MPI-Unterstützung erzeugen in den Testprogrammen die obligatorischen Aufrufe von `MPI_Init` und `MPI_Finalize`. Zudem wird sichergestellt, dass jeder Prozess seine eigenen Ein- und Ausgabedaten schreibt und liest.

Teil des anwendungsorientierten Ansatzes ist es, dass der generierte Code von der BenutzerIn bearbeitet werden kann. Daher sind die Standardtemplates dahingehend entworfen, möglichst lesbaren und verständlichen Code zu generieren. So enthält dieser so wenig Verschachtelungen wie möglich, Ein- und Ausgabevariablen sind alphabetisch sortiert, einzelne Teilaufgaben sind in Prozeduren gekapselt und die mit Serialisierung und Dateisystemoperationen verbundene Komplexität ist in einer Bibliothek verborgen (siehe auch nächster Abschnitt).

9.6.4 Datenaufzeichnung

Für die Aufzeichnung der Ein- und Ausgabedaten gibt es verschiedene Möglichkeiten, wie diese abgespeichert werden können. So ließen sich die einzelnen Variablen beispielsweise in einer relationalen Datenbank ablegen, in einem Objektspeicher oder in Form von Dateien. Als Speicherorte kämen etwa das lokale Dateisystem in Frage, ein Netzwerkspeicher oder auch eine Cloud-Infrastruktur. Das Templatekonzept ermöglicht den BenutzerInnen, frei zu wählen, welche Speichertechnologie zum Einsatz kommt. Da einzelne Schreiboperationen in Fortran ggf. aus sehr vielen Zeilen Code bestehen können, bietet es sich an, Bibliotheken zu verwenden, mit denen ein Großteil der Komplexität aus dem Capture- und Replaycode herausgehalten werden kann.

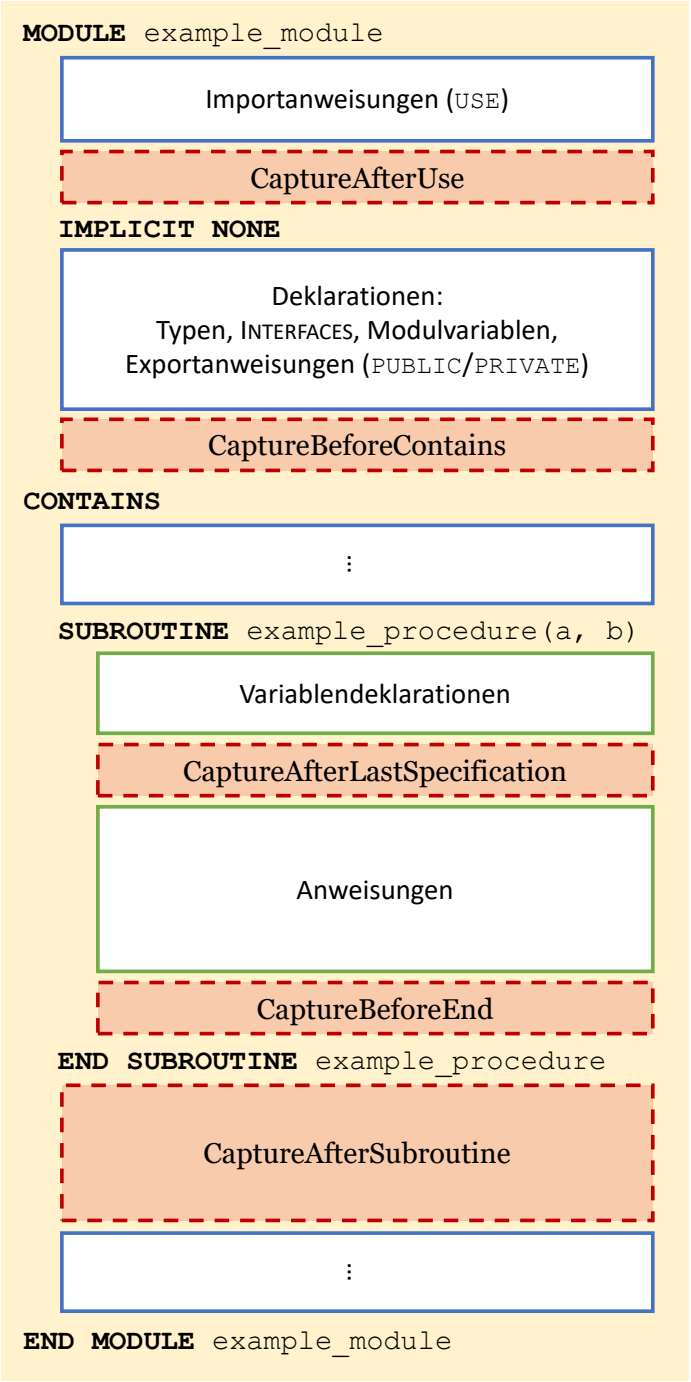


Abbildung 9.2: Einfügepunkte des Capturecodes

In den von FTG mitgelieferten Standardtemplates kommt hierfür die Bibliothek *Serialbox2* (GitHub, SB2) zum Einsatz. *Serialbox2* ist eine Serialisierungs- und I/O-Bibliothek, mit deren Hilfe die Werte einzelner Variablen, angereichert mit Metadaten, im Dateisystem abgespeichert werden können. Beispiele für Metadaten sind u.a. der Variablenname, Arraygrößen oder Datentyp. Zusätzlich lassen sich beliebige selbstdefinierte Metadaten mit abspeichern. Serialbox2 verfügt über Schnittstellen für C++, C, Python und Fortran. Als Dateiformate stehen ein eigenes Binärformat sowie NetCDF zur Verfügung. Serialbox2 wurde am schweizerischen CSCS gezielt für den Einsatz in der Wetter- und Klimamodellierung entwickelt. Um die Verwendung durch FTG etwas komfortabler zu gestalten, wurde im Rahmen dieser Arbeit die Fortran-Schnittstelle durch ein spezielles *FTG-Interface* erweitert, das nun Teil der Serialbox2-Bibliothek ist (GitHub, SB2-PR134). Dieses enthält u.a. Prozeduren, um Arrayvariablen auf Basis der abgespeicherten Größen zu alloziieren oder Prozeduren, um die aktuellen Werte von Variablen mit zuvor abgespeicherten Werten zu vergleichen.

9.6.5 Beispieltest

Im Folgenden soll anhand einer Beispiel-Subroutine und dem Standardtemplate *BaseCompare* die grundsätzliche Funktionsweise der Codegenerierung für das Capture erläutert werden. Das Template BaseCompare ist Teil von FTG, wurde jedoch speziell für die Veranschaulichung in dieser Arbeit erstellt. Alle Template Parts von BaseCompare werden in Anhang D.3 erläutert. Das folgende Beispiel beschränkt sich auf die Codeabschnitte captureAfterLastSpecification und captureBeforeEnd sowie das erzeugte Testprogramm.

Beispiel 9.17: BaseCompare

Original-PUT

```
1   SUBROUTINE example_procedure(a, b)
2
3     INTEGER, INTENT(in) :: a(:,:,:)
4     INTEGER, INTENT(inout) :: b(:,:)
5     INTEGER :: i
6
7     DO i = 1, data_count
8       b(:,:) = b(:,:) + a(i,:,:)
9     END DO
10
11  END SUBROUTINE example_procedure
```

Für die Subroutine `example_procedure` soll Capturecode erzeugt werden. fcg erkennt hier sowohl die beiden Dummyargumente mit primitivem Typ `a` und `b` als auch die Modulvariable `data_count` als verwendet.

Template Part captureAfterLastSpecification

```
12 $prologue
13
14 ftg_${subroutine.name}_round = ftg_${subroutine.name}_round + 1
15 IF (ftg_${subroutine.name}_capture_active()) THEN
16   CALL ftg_${subroutine.name}_capture_data($commaList("'input'", $args))
17 END IF
18
19 $epilogue
```

Der Codeabschnitt captureAfterLastSpecification erzeugt Code, der in die PUT direkt nach der letzten Variablendeklaration, d.h. vor der ersten Anweisung, eingefügt wird. Er dient der Aufzeichnung der Eingabevariablen der PUT. Die Platzhaltervariablen `$prologue` und `$epilogue` sind in der Templatehauptdatei definiert und enthalten Kommentarzeilen, die der optischen Abgrenzung des von FTG generierten Codes dienen.

Zunächst wird die in captureBeforeContains deklarierte Modulvariable `ftg_${subroutine.name}_round` inkrementiert (Zeile 14). Auf die Weise wird die Anzahl der Ausführungen der PUT gezählt. `${subroutine.name}` ist ein Platzhalter für den Namen der PUT.

Anschließend wird mit Hilfe der Funktion `ftg_${subroutine.name}_capture_active()` geprüft, ob die aktuelle Ausführung der PUT aufgezeichnet werden soll (Zeile 15). `ftg_${subroutine.name}_capture_active()` wird in captureAfterSubroutine definiert und prüft standardmäßig, ob der Wert von `ftg_${subroutine.name}_round` 1 ist. Diese Funktion kann von der BenutzerIn im Template oder im generierten Code beliebig geändert werden, um eine andere Capturebedingung zu definieren.

Liefert `ftg_${subroutine.name}_capture_active()` true, wird die Subroutine für das Aufzeichnen der Eingabedaten, welche ebenfalls in captureAfterSubroutine definiert ist, aufgerufen (Zeile 16). Die zur FTG Template API gehörende Funktion `$commaList` erzeugt eine kommaseparierte Ausgabe aller Parameter, in diesem Fall aus der Zeichenkette `'input'` und allen PUT-Dummyargumenten, welche in der API-Variable `$args` enthalten sind. Modulvariablen müssen nicht per Argument übergeben werden, da auf diese im gesamten Modul direkt zugegriffen werden kann.

Template Part captureBeforeEnd

```
20 $prologue
21
22 IF (ftg_${subroutine.name}_capture_active()) THEN
23   CALL ftg_${subroutine.name}_capture_data($commaList("'output'", $args,
       ↪ $result))
24 END IF
25
26 $epilogue
```

Der Template Part captureBeforeEnd von BaseCompare ist sehr ähnlich zu captureAfterLastSpecification. `ftg_${subroutine.name}_round` wird nicht noch einmal inkrementiert wird und an `ftg_${subroutine.name}_capture_data` wird anstelle von `'input' 'output'` übergeben. Handelt es sich bei der PUT um eine FUNKTION wird zusätzlich die Ergebnisvariable an `ftg_${subroutine.name}_capture_data` übergeben. Der Name der Ergebnisvariable ist in der API-Variable `$result` enthalten. Handelt es nicht um eine FUNKTION, ist diese Variable leer und wird von `$commaList` ignoriert.

Instrumentierte PUT

```
27 SUBROUTINE example_procedure(a, b)
28
29   INTEGER, INTENT(in) :: a(:,:,:)
30   INTEGER, INTENT(inout) :: b(:,:)
31   INTEGER :: i
32
33 ! =========== BEGIN FORTRAN TEST GENERATOR (FTG) ===========
34
35 ftg_example_procedure_round = ftg_example_procedure_round + 1
36 IF (ftg_example_procedure_capture_active()) THEN
37   CALL ftg_example_procedure_capture_data('input', a, b)
38 END IF
39
40 ! =========== END FORTRAN TEST GENERATOR (FTG) =============
41
42   DO i = 1, data_count
43     b(:,:) = b(:,:) + a(i,:,:)
44   END DO
45
46 ! =========== BEGIN FORTRAN TEST GENERATOR (FTG) ===========
47
48 IF (ftg_example_procedure_capture_active()) THEN
49   CALL ftg_example_procedure_capture_data('output', a, b)
50 END IF
51
52 ! =========== END FORTRAN TEST GENERATOR (FTG) =============
53
54 END SUBROUTINE example_procedure
```

Die instrumentierte PUT enthält den auf Basis der Template Parts captureAfterLastSpecification (Zeilen 33–40) und captureBeforeEnd (Zeilen 46–52) erzeugten Code. Die Platzhalter `$prologue` und `$epilogue` wurden durch die langen BEGIN- und END-Kommentarzeilen ersetzt,

`${subroutine.name}` durch `example_procedure` und der Aufruf von `ftg_example_procedure_capture_data` enthält nun die Liste der DUMMYARGUMENTE.

Durch das Einfügen des Capturecodes unmittelbar vor der ersten Anweisung ist sichergestellt, dass alle Eingabedaten unverändert aufgezeichnet werden. Das Aufzeichnen der Ausgabedaten wird dagegen direkt nach der letzten Anweisung der Prozedur durchgeführt.

Testprogramm

```
55  PROGRAM ftg_example_procedure_test
56
57    USE m_ser_ftg
58    USE m_ser_ftg_cmp
59    USE example_module, ONLY: example_procedure, data_count
60
61    IMPLICIT NONE
62
63    INTEGER :: failure_count
64    INTEGER, DIMENSION(:,:,:), ALLOCATABLE :: a
65    INTEGER, DIMENSION(:,:), ALLOCATABLE :: b
66
67    CALL ftg_example_procedure_replay_input(a, b)
68    CALL example_procedure(a, b)
69    CALL ftg_example_procedure_compare_output(b, failure_count)
70
71    IF (failure_count > 0) THEN
72      WRITE (*,*) '*** TEST FAILED ***'
73    ELSE
74      WRITE (*,*) '*** TEST PASSED ***'
75    END IF
76
77  CONTAINS
78
79    ⋮
80
81    SUBROUTINE ftg_example_procedure_replay_input(a, b)
82
83      INTEGER, DIMENSION(:,:,:), ALLOCATABLE, INTENT(inout) :: a
84      INTEGER, DIMENSION(:,:), ALLOCATABLE, INTENT(inout) :: b
85
86      INTEGER, DIMENSION(8) :: ftg_bounds
87      INTEGER :: ftg_d1, ftg_d2, ftg_d3, ftg_d4
88      CHARACTER(len=256) :: ftg_name
89
90      CALL ftg_example_procedure_init_serializer('input')
91      CALL ftg_allocate_and_read_allocatable("a", a)
92      CALL ftg_allocate_and_read_allocatable("b", b)
93      CALL ftg_read("data_count", data_count)
94      CALL ftg_example_procedure_close_serializer()
95
96    END SUBROUTINE ftg_example_procedure_replay_input
97
98    SUBROUTINE ftg_example_procedure_compare_output(b, failure_count)
99
100     INTEGER, DIMENSION(:,:), INTENT(in) :: b
101     INTEGER, INTENT(out) :: failure_count
102
103     LOGICAL :: result
```

```
104     INTEGER :: ftg_d1, ftg_d2, ftg_d3, ftg_d4
105     CHARACTER(len=256) :: ftg_name
106
107     failure_count = 0
108
109     CALL ftg_example_procedure_init_serializer('output')
110     CALL ftg_compare("b", b, result, failure_count, LBOUND(b), UBOUND(b))
111     CALL ftg_compare("data_count", data_count, result, failure_count)
112     CALL ftg_example_procedure_close_serializer()
113
114   END SUBROUTINE ftg_example_procedure_compare_output
115
116 END PROGRAM ftg_example_procedure_test
```

Aufgrund seiner Länge wird das Template für das Testprogramm hier nicht abgedruckt. Es ist ebenfalls in Anhang D.3 zu finden. Gezeigt wird hier der für `example_procedure` generierte Test. Zur besseren Veranschaulichung wurde der MPI-Code für die Ausführung mit mehreren Prozessen entfernt, da `example_procedure` ohnehin keine Interprozesskommunikation enthält. Außerdem wurden unnötige Leerzeilen und Kommentare entfernt.

Das Testprogramm importiert zunächst die Module des FTG-Interface von Serialbox2 (Zeilen 57+58) sowie die PUT `example_procedure` und die benötigte Modulvariable `data_count` aus dem PUT-Modul `example_module` (Zeile 59).

Das Hauptprogramm besteht aus vier Schritten:

1. Laden der Eingabedaten (Zeile 67)
2. Ausführen der PUT (Zeile 68)
3. Vergleich der Ausgabedaten mit zuvor aufgezeichneten Referenzdaten (Zeile 69)
4. Ausgabe des Testergebnisses (Zeilen 71–75)

Schritt 1 und 3 sind jeweils in separaten Subroutinen realisiert. `ftg_example_procedure_replay_input` (Zeilen 81–96) enthält den Code zum Laden der Eingabedaten. Die in den Zeilen 90 und 94 aufgerufenen Subroutinen dienen dem Initialisieren bzw. Aufräumen von Serialbox2 und sind hier aus Platzgründen nicht abgedruckt. In den Zeilen 91–93 werden für die `Dummyargumente a` und `b` sowie für die Modulvariable `data_count` nacheinander die entsprechenden Serialbox-2-Subroutinen zum Einlesen der aufgezeichneten Daten aufgerufen. Arrayvariablen wie `a` und `b` werden in mit BaseCompare erstellten Tests grundsätzlich als `ALLOCATABLE` deklariert (siehe Abschnitt 2.4.3) und von der I/O-Bibliothek entsprechend der abgespeicherten Größe angelegt (`ftg_allocate_and_read_allocatable`),

während für die skalare Variablen wie `data_count` einfach nur der abgespeicherte Wert gesetzt werden muss (`ftg_read`).

`ftg_example_procedure_compare_output` (Zeilen 98–114) ist für das Vergleichen der Ausgabedaten von `example_procedure` mit den aufgezeichneten Referenzdaten zuständig. Hierzu bietet das FTG-Interface von Serialbox2 die SUBROUTINE `ftg_compare` an, welche hier für das INOUT-ARGUMENT `b` sowie für die MODULVARIABLE `data_count` aufgerufen wird (Zeilen 110+111). Bei Abweichungen wird die Variable `failure_count` von `ftg_compare` inkrementiert und die boolesche Variable `result` auf false gesetzt. Letztere wird hier im BaseCompare-erzeugten Test nicht verwendet, stattdessen wird der Erfolg des Tests anhand von `failure_count` bestimmt.

9.7 Arbeitsablauf

Ein typischer Arbeitsablauf für die Erstellung eines Tests mit dem FortranTestGenerator könnte beispielsweise folgendermaßen aussehen:

Beispiel 9.18: Arbeitsablauf FortranTestGenerator

1. Erstellung des Assemblercodes: Kompilieren der Originalanwendung mit dem GNU Fortran Compiler und den Optionen `-S -g -O0` oder `-save-temps -g -O0` (vgl. Abschnitt 9.4.1).
2. Instrumentieren der Originalanwendung mit dem Capturecode:

   ```
   $> ./FortranTestGenerator -c <modul> <prozedur>
   ```

   Der Exportcode wird automatisch mitgeneriert.
3. Kompilieren der instrumentierten Originalanwendung
4. Capture: Ausführen der Originalanwendung, zum Beispiel durch Ausführen eines Ende-zu-Ende-Tests in einer geeigneten Konfiguration, durch die es zur Ausführung der PUT kommt.
5. Entfernen des Capturecodes aus der Originalanwendung:

   ```
   $> ./FortranTestGenerator -a
   ```

6. Erstellung des Testprogramms:

```
$> ./FortranTestGenerator -r <modul> <prozedur>
```

7. Kompilieren des Testprogramms
8. Replay: Ausführen des Testprogramms
9. Manuelle Erweiterung des Testprogramms
   (anschließend fortfahren mit Schritt 7)

Mit Hilfe eines Skripts lässt sich dieser Prozess bis auf den letzten Schritt sehr gut automatisieren. Dabei muss er nicht in dieser Reihenfolge durchgeführt werden. So ließe sich beispielsweise das Testprogramm auch zusammen mit der Instrumentierung der Originalanwendung erstellen. Das Capture muss natürlich vor dem Replay stattfinden. Der Capturecode muss hingegen nicht unbedingt entfernt werden. Je nach Bedarf kann er auch (zunächst) in der Originalanwendung verbleiben. Durch Verwendung geeigneter Konfigurationsvariablen ist dabei sicherzustellen, dass das Capture in diesem Fall nur durchgeführt wird, wenn es benötigt wird.

9.8 Anpassbarkeit

Die BenutzerInnen des FortranTestGenerators können auf drei verschiedene Arten in den Testerstellungsprozess eingreifen: Erstens können sie die Reihenfolge der Arbeitsschritte wie zuvor beschrieben selbst bestimmen, soweit die Logik des Prozesses dies zulässt. Zweitens können sie den von FTG generierten Code bearbeiten. Drittens können sie das Werkzeug selbst an die jeweilige Einsatzumgebung anpassen.

Die in Abbildung 9.3 dargestellt Sicht auf den FortranTestGenerator zeigt verschiedene Komponenten und Schichten des Werkzeugs, die auf unterschiedliche Weise veränderbar sind. Im Inneren ist FortranCallGraph dargestellt, welches den Aufrufgraph erzeugt und die Codeanalyse durchführt. fcg kann auch als eigenständiges Werkzeug verwendet werden, um Eigenschaften einer Fortran-Prozedur abzufragen. Zusammen mit der Template API und den definierten Einfügepunkten der Template Parts bildet es dazu den Kern von FTG. Dieser Kern stellt ein relativ generisches Werkzeug zur Codegenerierung dar, das lediglich durch die Logik und Benennung der Einfügepunkte den Einsatzzweck Capture & Replay nahelegt.

Erst durch die mitgelieferten Standardtemplates wird die Funktionalität der Ein- und Ausgabeaufzeichnung und Testerzeugung realisiert. Auf dieser Ebene können die BenutzerInnen eingreifen. So können die erzeugten Tests angepasst werden oder

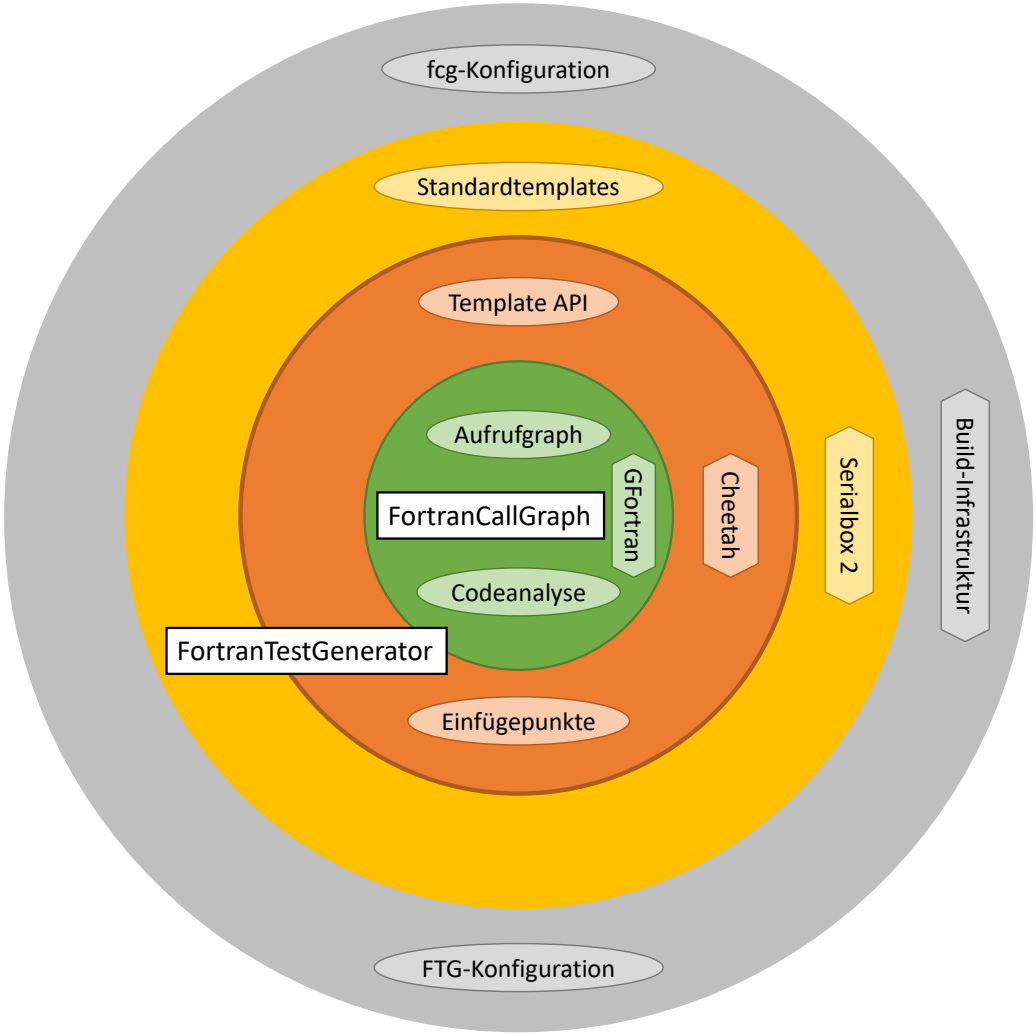


Abbildung 9.3: Komponenten und Schichten des FortranTestGenerators

der Code zur Aufzeichnung der Daten. Es kann auf Basis der von fcg gelieferten Informationen auch Code für völlig andere Zwecke erzeugt werden. Dabei muss es sich nicht einmal um Code handeln, auch wenn die vorgegebenen Einfügepunkte dies nahelegen. Für Anpassungen auf dieser Ebene ist ein gewisses Maß an Einarbeitung nötig, in die Cheetah-Syntax, die Template API und ggf. in die Zielsprache. Sie sind daher nicht obligatorisch. In vielen Fällen dürften auch die Standardtemplates ihren Zweck erfüllen. Sollten Anpassungen notwendig sein, müssen diese nicht von jeder BenutzerIn durchgeführt werden. Es reicht, wenn diese für eine Zielanwendung einmal durchgeführt werden, ggf. von technisch versierteren EntwicklerInnen und/oder mit externer Hilfe, und dann allen EntwicklerInnen zur Verfügung gestellt werden.

Notwendig ist jedoch eine Anpassung der auf der äußersten Ebene dargestellten Konfigurationen in den entsprechenden Konfigurationsdateien von fcg und FTG. Hier müssen grundlegende Angaben wie etwa Dateipfade usw. gemacht werden. Auch diese können im Entwicklungsteam einer Zielanwendung zentral bereitgestellt werden, so dass einzelne EntwicklerInnen nur noch minimale Anpassungen vornehmen müssen.

9.9 Externe Softwarepakete

In Abbildung 9.3 sind in den sechseckigen Kästchen jeweils externe Programme und Bibliotheken dargestellt, die auf den jeweiligen Ebenen benötigt werden. fcg benötigt die vom GNU Fortran Compiler (GFortran) erzeugten Assemblerdateien. GFortran muss daher zur Verfügung stehen, auch wenn es das Ziel ist, in Zukunft weitere Formate zu unterstützen. Der Codegenerierungsprozess von FTG basiert auf der Cheetah Template-Engine, auch diese ist daher obligatorisch. Serialbox2 wird von den mitgelieferten Standardtemplates verwendet und ist somit grundsätzlich austauschbar. Nicht vorgegeben, aber notwendig ist eine Buildinfrastruktur, mit der die erzeugten Tests und die instrumentierte Originalanwendung kompiliert werden können. Für Letztere dürfte i.d.R. die vorhandene Infrastruktur verwendet werden können. Um auch die Tests (effizient) kompilieren zu können, muss diese jedoch ggf. erweitert werden.

Nicht abgebildet, aber notwendig, um sowohl fcg als FTG ausführen zu können, ist ein Python-Interpreter. Unterstützt werden sowohl Python 2.7 als auch Python 3.x. Ebenso wie GFortran gehört Python i.d.R. zur Standardausstattung von Hochleistungsrechnern, während Cheetah und Serialbox2 normalerweise von der BenutzerIn selbst installiert werden müssen. Für die Verwendung beider Pakete sind einfache Benutzerrechte ebenso ausreichend wie für fcg und FTG insgesamt.

9.10 Zusammenfassung

In diesem Kapitel wurde der FortranTestGenerator (FTG) vorgestellt, ein Softwarewerkzeug, das die Testerzeugung mittels Capture & Replay unterstützt. Das System besteht aus zwei Komponenten: FortranCallGraph (fcg) ein Programm zur statischen Analyse von Fortran-Code, mit dem die Ein- und Ausgabevariablen einer zu testenden Prozedur (PUT) ermittelt werden, und dem eigentlichen Testgenerator. Beide Programme sind in Python geschrieben und unter einer Open-Source-Lizenz frei verfügbar.

fcg extrahiert einen Aufrufgraph aus vom Compiler erzeugten Assemblerdateien und analysiert den Originalquellcode, indem vor allem mittels regulärer Ausdrücke Verwendungen einzelner Variablen gesucht werden. Dieses Verfahren wurde insbesondere aufgrund seiner Robustheit und der Unabhängigkeit von externen Softwarekomponenten oder -werkzeugen gewählt. Auf Grundlage der von fcg bereitgestellten Informationen erzeugt FTG Capture- und Exportcode für die Instrumentierung der Originalanwendung sowie ein Testprogramm für die PUT (Replaycode). Der generierte Code kann mit Hilfe anpassbarer Vorlagen, sog. Templates von der BenutzerIn beliebig an die jeweilige Zielumgebung angepasst werden. Die Codegenerierung basiert auf der frei verfügbaren Cheetah Template Engine, definierten Einfügepunkten für den Instrumentierungscode sowie einer Template API, mit der innerhalb der Codevorlagen auf die von fcg gelieferten Informationen zugegriffen werden. FTG enthält bereits verschiedene Standardtemplates für Capture-, Export- und Replaycode. Beim Capture werden neben den Eingabedaten der PUT auch ihre Ausgabedaten während der Ausführung der Originalanwendung aufgezeichnet. Diese dienen im erzeugten Test als Referenzdaten, mit denen die Ausgabedaten, die die PUT im Test liefert, automatisch verglichen werden. Zum Aufzeichnen und Laden der Ein- und Ausgabedaten kommt in den Standardtemplates die Serialisierungs- und I/O-Bibliothek Serialbox2 zum Einsatz.

Die BenutzerIn steuert selbst die einzelnen Capture-&-Replay-Schritte wie das Erzeugen von Capture- und Replaycode, das Kompilieren und das Ausführen von instrumentierter Originalanwendung und Test. Dies ermöglicht ihr, in den Prozess einzugreifen, etwa indem sie den generierten Code manuell bearbeitet. Zu diesem Zweck wurden die von FTG mitgelieferten Standardtemplates dahingehend optimiert, dass sie möglichst verständlichen Code erzeugen.

Kapitel 10

Grenzen

Capture & Replay (C&R) ist ein vielversprechender, anwendungsorientierter Ansatz, um Unittests für einzelne Prozeduren von Klimamodellen zu erstellen, insbesondere für bereits existierende Prozeduren. Der Ansatz verfügt jedoch auch über einige grundlegende Grenzen. Und auch die Umsetzung in Form des FortranTestGenerators (FTG) stellt kein lückenloses System dar, das alle Sonderfälle bei der Testerzeugung berücksichtigt. Dieses Kapitel beschäftigt sich mit diesen grundlegenden Grenzen des C&R (Abschnitt 10.1) sowie den Beschränkungen der Umsetzung (Abschnitt 10.2). Zudem wird der Einsatz im parallelen Kontext diskutiert, wenn die in Abschnitt 8.2.5 getroffenen Annahmen nicht gelten.

10.1 Grundlegende Grenzen des C&R-Ansatzes

10.1.1 Beschränktheit der Testfälle

Durch die Gewinnung der Testdaten aus der Originalanwendung ist die Menge der möglichen Testfälle begrenzt. Es können nur solche Daten gewonnen werden, die durch Ausführung der Originalanwendung erzeugt werden können. Ein systematisches Testen wie bei der manuellen Testdatenerstellung, bei der gezielt Grenzwerte und Vertreter von Äquivalenzklassen als Eingabedaten verwendet werden können, ist so nicht möglich. Auch eine hohe Codeabdeckung, die Ziel einiger der in Kapitel 7 beschriebenen Ansätze ist, ist so unter Umständen nicht immer zu erreichen. Durch die Ableitung von Testfällen durch Modifikation der gewonnenen Daten lassen sich zwar gezielt bestimmte Grenzfälle oder Ausführungspfade testen, dies ist jedoch nur in dem Rahmen möglich, dass die Konsistenz der Testdaten durch die manuellen Veränderungen nicht zerstört wird.

10.1.2 Schnittstellenänderungen

Beim Regressionstesten besteht die Möglichkeit, dass bei Änderungen an der PUT die mit einer Vorversion ermittelten und aufgezeichneten Eingabedaten nicht mehr passen. So könnten beispielsweise neu benötigte Eingabedaten fehlen oder sich die Struktur einzelner Daten verändert haben. In dem Fall müssen die Eingabedaten neu beschafft werden, indem der C&R-Prozess erneut durchgeführt wird. Soll dies vor Änderung der PUT geschehen, müssen neu benötigte Daten entweder manuell zur Liste der Eingabedaten hinzugefügt werden oder der PUT müssen zumindest solche Anweisungen hinzugefügt werden, dass die neu benötigten Daten durch die statische Analyse erkannt werden. Handelt es sich um ergebnisneutrale Änderungen, können die Ausgabedaten der Vorversion weiter als Referenz verwendet werden, ggf. müssen auch diese durch erneute Aufzeichnungen ergänzt werden. Ähnliche Probleme bestehen jedoch auch bei anderen Formen der Erstellung von Regressionstests.

10.1.3 Datenmenge

Abhängig von der PUT und der für das Capture gewählten Modellkonfiguration kann die Menge der aufgezeichneten Eingabedaten sehr groß sein. Dies schränkt zum einen die Verständlichkeit des Tests ein und zum anderen verlangsamt dies die Testausführung, da die Eingabedaten zunächst aus dem Speichersystem geladen werden müssen. Dies steht im Widerspruch mit der für Unittests geforderten Schnelligkeit. Dennoch verspricht dieser Ansatz, Tests zu ermöglichen, die schneller sind als die ansonsten üblichen Ende-zu-Ende-Tests. Im Rahmen der Evaluation (Kapitel 11) wird daher zu untersuchen sein, ob sich hier tatsächlich ein Geschwindigkeitsvorteil ergibt und wie hoch dieser ggf. ist.

10.1.4 Technische Beschränkungen

Weitere Einschränkungen ergeben sich unter Umständen aus den technischen Rahmenbedingungen. So hängt die Tauglichkeit des Ansatzes entscheidend davon ab, dass die Eingabedaten einer PUT vollständig identifiziert werden können. Wie in Abschnitt 8.3.3 dargestellt, haben mögliche Analysestrategien jeweils eigene, spezifische Einschränkungen. Neben der Identifizierbarkeit muss zudem auch die Aufzeichenbarkeit gegeben sein, das heißt, dass sämtliche Daten mit den gewählten Mitteln erfassbar sein müssen. Insbesondere bei Daten, die von externen Bibliotheken oder Drittsystemen verwalteten werden, könnten sich unter Umständen Einschränkungen ergeben. Sowohl bei der Identifizierung als auch der Aufzeichnung der Daten ist es

jedoch denkbar, dass die EntwicklerIn unterstützend eingreift, um Unzulänglichkeiten der C&R-Software auszugleichen. Dadurch erhöht sich jedoch der Aufwand der Testerstellung.

10.1.5 Nichtdeterminismus

Per C&R aufgezeichnete Ausgabedaten können nur dann als Testorakel verwendet werden, wenn sich die zu testende Prozedur deterministisch verhält. Ist dies nicht der Fall, müssen andere Testorakel verwendet werden. Dies ist jedoch keine Besonderheit des C&R-Ansatzes, sondern betrifft jede Art von Test. Der exakte Vergleich mit konkreten Erwartungswerten ist in jedem Fall nur bei deterministischen Testobjekten möglich. Nichtdeterminismus schränkt jedoch nicht die Verwendung von per C&R gewonnenen Eingabedaten ein.

10.2 Beschränkungen der Umsetzung

Die Entwicklung des FortranTestGenerators dient im Wesentlichen der Demonstrationen der grundsätzlichen Machbarkeit und Tauglichkeit des C&R-Ansatzes. Aufgrund der zeitlichen Beschränkung dieser Arbeit und der Herangehensweise (Abschnitt 9.2) verfügt er über Unzulänglichkeiten, die in Ausnahmefällen die Funktion einschränken. Einige davon sind grundsätzlicher Natur, andere lediglich der Priorisierung der umzusetzenden Funktionalität zum Opfer gefallen.

10.2.1 Nutzdatenfokussierung

Die Funktionsweise von FortranCallGraph und des FortranTestGenerators basiert auf der Annahme, dass das Verhalten der PUT durch die Nutzdaten der übergebenen Datenstrukturen bestimmt wird, d.h. durch die Variablen und Komponenten mit primitiven Datentypen. Nur diese werden als „verwendete“ Variablen angesehen und gespeichert. Strukturdaten, d.h. Datenstrukturen mit Verbunddatentypen werden nur insofern beachtet, als sie zu Analyse und Speicherung ihrer Basiskomponenten benötigt werden. Sämtliches Verhalten, das auf Strukturdaten basiert, welche aufgrund dieses Vorgehens nicht gespeichert werden, wird daher ggf. nicht korrekt reproduziert. Zudem werden rekursive Datenstrukturen von fcg/FTG nicht unterstützt.

10.2.2 Dynamische Datenstrukturen

Der von FTG generierte Code basiert auf Informationen, die mithilfe einer statischen Codeanalyse ermittelt werden. Dynamische Strukturen, etwa bedingt durch objektorientierte Vererbung (siehe auch Abschnitt 2.4.6), können auf diese Weise nicht erfasst werden. In einigen Fällen können diese mit Hilfe der BenutzerIn aufgelöst werden, eine allgemeine Lösung existiert hierfür jedoch nicht.

So können BenutzerInnen etwa über die Konfiguration einem abstrakten Typ einen konkreten zuweisen (siehe auch Anhang C.1). fcg behandelt dann das Vorkommen des abstrakten Typs so als würde der gewählte konkrete Typ benutzt werden, um so etwa TYPGEBUNDENE PROZEDUREN aufzulösen. Die Standardtemplates von FTG sind zudem in der Lage den entsprechenden Code zu generieren, um die BASISKOMPONENTEN des ausgewählten Typs aufzuzeichnen. Dies ist vor allem dann nützlich, wenn zur Laufzeit jeweils nur ein konkreter Typ verwendet wird, etwa abhängig von der Modellkonfiguration. Wenn jedoch während einer Simulation tatsächlich mehrere konkrete Typen für einen abstrakten Typ eingesetzt werden, stößt dies an seine Grenzen.

Grundsätzlich wäre es auch möglich, fcg anstatt nur einer Zuordnung von konkretem zu abstraktem Typ mehrere Varianten analysieren zu lassen und FTG den passenden Code für mehrere mögliche Typen generieren zu lassen. Diese Möglichkeit ist jedoch noch nicht implementiert. Zudem würde hierdurch der generierte Code u.U. deutlich komplexer werden.

10.2.3 SAVE-Variablen

Das Capture findet in FTG zwar durch Instrumentierung der PUT statt, so dass Ein- und Ausgabedaten innerhalb der PUT aufgezeichnet werden. Beim Replay werden diese Daten der PUT jedoch von außen übergeben, d.h. entweder in Form von ARGUMENTEN oder globalen Variablen. Die PUT benötigt hierfür keine Instrumentierung mehr. Dadurch ist jedoch nicht möglich, die Werte von Variablen mit dem SAVE-Attribut zu setzen, d.h. von lokalen Variablen, deren Werte von einem Prozeduraufruf zum nächsten erhalten bleiben (siehe auch Abschnitte 2.4.1 und 8.2.5) Dies ist insofern problematisch, als dass die in diesen Variablen gespeicherten Werte maßgeblich das Verhalten einer Prozedur beeinflussen können. Ein Injizieren von aufgezeichneten Daten in solche Variablen wäre nur durch eine replaybezogene Instrumentierung der PUT möglich. Dies ist in FTG jedoch nicht vorgesehen.

10.2.4 RETURN-Anweisungen

FTG definiert feste Einfügepunkte für den Capturecode. Dazu gehört u.a., dass der Code für das Aufzeichnen der Ausgabedaten nach der letzten Anweisung in die PUT eingefügt wird (siehe auch Abschnitt 9.6.2). Enthält eine Prozedur jedoch eine oder mehrere RETURN-Anweisungen, die dafür sorgen, dass die Prozedur unter Umständen frühzeitig verlassen wird, wird der am Ende eingefügte Code ggf. nicht ausgeführt. Hier muss die BenutzerIn manuell dafür sorgen, dass der Capturecode vor die vorhandenen RETURN-Anweisungen verschoben oder kopiert wird.

10.2.5 Instrumentierung rein additiv

FTG kann im Rahmen der Instrumentierung der Originalanwendung ausschließlich Code hinzufügen, jedoch keinen bestehenden Code löschen oder auskommentieren. Dies schränkt vor allem den Export privater Modulelemente ein. Der von FTG erzeugte Exportcode kann lediglich mit Hilfe einer PUBLIC-Deklaration Elemente eines privaten Moduls, die nicht individuell als PRIVATE oder PROTECTED gekennzeichnet sind, exportieren (siehe auch Abschnitt 2.4.2).

Beispiel 10.1: Exportcode

```
1  MODULE private_module
2    IMPLICIT NONE
3    PRIVATE
4
5    INTEGER :: modvar
6
7  ! =========== BEGIN FORTRAN TEST GENERATOR (FTG) ===========
8
9  #ifdef __FTG__
10 PUBLIC :: modvar
11 #endif
12
13 ! =========== END FORTRAN TEST GENERATOR (FTG) =============
14
15   CONTAINS
16
17   ⋮
18
19 END MODULE private_module
```

Durch das Schlüsselwort PRIVATE (Zeile 3) sind alle Elemente des Moduls private_module standardmäßig privat. Durch die von FTG erzeugte PUBLIC-Deklaration (Zeile 10) wird die Modulvariable modvar exportiert.

```
20 MODULE private_module
21   IMPLICIT NONE
22
23   INTEGER, PRIVATE :: modvar
24
25   CONTAINS
26
27   ⋮
28
29 END MODULE private_module
```

Da die Modulvariable `modvar` hier individuell als `PRIVATE` deklariert ist (Zeile 23), kann sie von FTG nicht exportiert werden. Da Capture- und Replaycode jedoch auf `modvar` zugreifen wollen, wird der Compiler eine entsprechende Fehlermeldung produzieren und die BenutzerIn so auf diesen Umstand aufmerksam machen. Sie muss dann das Schlüsselwort `PRIVATE` manuell entfernen. Damit die Kapselung nicht verloren geht, sollte dabei mit einer Präprozessoranweisung eine entsprechende Fallunterscheidung eingefügt werden, so dass der Export nur für Capture- und Testmodus gilt.

10.2.6 Beschränkungen von Serialbox2

Die Standardtemplates von FTG verwenden die Bibliothek Serialbox2 zur Serialisierung und zum Schreiben und Lesen der Ein- und Ausgabedaten. Diese Bibliothek unterliegt einigen Beschränkungen. So unterstützt sie maximal vierdimensionale Arrays und nur die gängigsten Datentypen. Strings werden zudem nur eingeschränkt unterstützt. Für die meisten Variablen in Klimamodellen ist dies jedoch ausreichend.

10.2.7 Aliase

Die Standardtemplates von FTG erzeugen im Testprogramm für jede Ein- und Ausgabevariable eine eigene Variable, auch für Zeigervariablen. Beim Capture kann es jedoch vorkommen, dass mehrere Zeigervariablen auf dieselbe Speicherstelle verweisen. Dies kann dazu führen, dass das Verhalten der Prozedur beim Replay nicht korrekt reproduziert wird. Insbesondere für Variablen, die zu den Ausgabedaten gehören, ist dies problematisch, wenn sich deren Werte beim Capture aufgrund von Aliasen verändern, beim Replay jedoch nicht. Die Standardtemplates speichern jedoch die Speicheradressen der Variablen in den Serialbox-2-Metadaten. So können BenutzerInnen manuell oder mit geeigneten Skripten Aliasbeziehungen erkennen und das Testprogramm entsprechend verändern, so dass diese Beziehungen wiederhergestellt werden. Eine automatische Wiederherstellung von Aliasbeziehungen, etwa innerhalb

des FTG-Interfaces von Serialbox2, wäre grundsätzlich möglich, ist jedoch noch nicht implementiert.

10.2.8 Assumed Size Arrays

Damit Variablen zu Beginn einer Prozedur korrekt aufgezeichnet werden können, müssen ihre Größe und Struktur bekannt sein. Da Fortran eine streng typisierte Programmiersprache ist, ist dies in der Regel der Fall. Daher enthält auch die Serialbox2-Bibliothek spezielle Schreibroutinen für unterschiedliche Datentypen. Zudem liefern Arrays ab Fortran 90 Informationen über ihre Größe und Indexbereiche. Dies ermöglicht etwa Prozeduren, Arrays unterschiedlicher Größe als Argument entgegen zu nehmen und diese korrekt zu verarbeiten. Zuvor musste stattdessen mit sog. Assumed Size Arrays gearbeitet werden. Auch diese ermöglichen es, dass Prozeduren Arrays unterschiedlicher Größe verarbeiten können, da diese ihre Größe und Indexbereiche jedoch nicht selbst mitteilen, werden zusätzliche Mittel benötigt, um sicherzugehen, dass keine Zugriffe außerhalb der Indexgrenzen versucht werden. Ein typisches Vorgehen ist dabei, ähnlich wie in C, die Größe eines Arrays in einem zweiten Argument mitzuliefern.

In einigen Klimamodellen finden sich noch ältere Prozeduren mit Assumed Size Arrays. Sind diese so aufgebaut, dass deren Indexbereiche zusätzlich als Argument übergeben werden, ist das Aufzeichnen dieser Arrays zu Beginn der Prozeduren möglich. FTG ist zwar nicht in der Lage den passenden Code zu generieren, da die Semantik der Argumente unbekannt ist, jedoch kann hier manuell nachgeholfen werden. Werden die Indexbereiche eines Arrays erst im Laufe einer Prozedur aufwendig berechnet, kann es jedoch sein, dass eine Aufzeichnung zu deren Beginn unmöglich ist.

10.2.9 Uninitialisierte und hängende Zeigervariablen

Unverknüpfte Zeigervariablen (siehe auch Abschnitt 2.4.4) enthalten keine Daten und können somit beim Capture auch nicht aufgezeichnet werden. Beim Replay bleiben diese Variablen entsprechend auch unverknüpft. Zu diesem Zweck enthalten die Schreibeprozeduren des FTG-Interface von Serialbox2 Abfragen des Status von Zeigervariablen mit Hilfe der Fortran-Funktion `ASSOCIATED`. Diese Funktion liefert jedoch nur dann ein brauchbares Ergebnis, wenn die Zeigervariable einen definierten Status (verknüpft oder unverknüpft) hat. Bei hängenden Zeigern oder Zeigervariablen, die nie initialisiert wurden, liefert `ASSOCIATED` ein mehr oder weniger zufälliges Ergebnis. Liefert `ASSOCIATED` in diesem Fall false, tritt kein Problem auf; dann

wird die Variable so behandelt werden als wäre sie unverknüpft, was i.d.R. zu korrektem Verhalten führen dürfte. Liefert `ASSOCIATED` jedoch true, wird fälschlicherweise versucht, den Inhalt der Variable aufzuzeichnen. Im besten Fall kommt es dabei zu einem Speicherzugriffsfehler. Es kann jedoch auch passieren, dass unbemerkt falsche Daten aufgezeichnet werden.

In manchen Fällen lassen sich durch die hieraus folgenden Probleme hängende Zeiger im zu testenden Code aufdecken, in den meisten dürften diese jedoch eher lästig sein und das Funktionieren der Tests behindern. Dies trifft insbesondere im Zusammenhang mit nicht initialisierte Zeigervariablen zu, da diese deutlich häufiger auftreten. Die einzige Möglichkeit, die geschilderten Probleme weitestgehend zu verhindern, ist es, keine uninitialisierten Zeigervariablen zuzulassen. Hierzu können diese bereits bei der Deklaration mit `NULL()` verknüpft werden. Um FTG erfolgreich einsetzen zu können, müssen derartige Initialisierung ggf. an allen relevanten Stellen im Anwendungscode eingefügt werden.

Beispiel 10.2: Zeigerinitialisierung

```
1 TYPE typeWithPointerComponents
2   INTEGER, POINTER :: intPtrA => NULL(), intPtrB => NULL()
3   REAL, POINTER :: realPtrC => NULL()
4 END TYPE typeWithPointerComponents
```

10.2.10 Weitere nicht implementierte Funktionen

fcg und FTG wurden im Verlauf dieser Arbeit stetig weiterentwickelt, jedoch konnten bis Fertigstellung dieser Dissertation nicht alle gewünschten Funktionen umgesetzt werden. So wurde etwa die Unterstützung von `Common Blocks` aufgrund deren seltener Verwendung zurückgestellt. Zudem werden etwa überladene Operatoren noch nicht unterstützt, was dazu führt, dass per Operator aufgerufene Prozeduren bei der Analyse nicht berücksichtigt werden. Subroutinen, die sich außerhalb von Modulen befinden, können von fcg noch nicht gefunden und analysiert werden. Außerdem fehlt es noch an einer ausführlichen Dokumentation. Weitere fehlende Funktionen werden zudem auf den GitHub-Seiten von fcg und FTG gelistet (GitHub, FTG-Issues; GitHub, fcg-Issues).

10.3 C&R in parallelen Anwendungen

Der in dieser Arbeit beschriebene C&R-Ansatz sowie der FortranTestGenerator funktionieren grundsätzlich auch mit MPI-basierten parallelen Anwendungen. Hierzu müssen jedoch zwei Voraussetzungen gegeben sein (vgl. Abschnitt 8.2.5):

1. Capture und Replay müssen mit derselben Anzahl Prozesse durchgeführt werden.
2. Die zu reproduzierende Ausführung der PUT darf nur intern mit sich selbst kommunizieren.

Im Folgenden werden diese beide Voraussetzungen näher erläutert und diskutiert sowie eine Lösung vorgeschlagen, den C&R-Ansatz auf Anwendungsfälle zu erweitern, in denen diese nicht gegeben sind.

10.3.1 Prozessanzahl und -konfiguration

Wenn die Originalanwendung mit dem von FTG generierten Capturecode mit mehreren MPI-Prozessen ausgeführt wird, werden die Eingabevariablen pro Prozess aufgezeichnet. Bei Verwendung der Standardtemplates führt dies dazu, dass die Serialbox-2-Bibliothek im Dateisystem eine Datei pro Variable und Prozess anlegt. Die laufende Nummer des Prozesses ist dabei Teil des Dateinamens. Beim Replay lädt jeder Prozess die ihm zugehörigen Daten.

Eine Ausführung des Replays mit mehr Prozessen als beim Capture ist auf diese Weise ohne manuellen Eingriff nicht möglich, da für die hinzukommenden Prozesse keine Eingabedaten zur Verfügung stehen. Ein Test mit weniger Prozessen wäre grundsätzlich möglich, jedoch kann in vielen Fällen davon ausgegangen werden, dass dieser nicht dieselben Ergebnisse liefert wie die Ausführung mit der ursprünglichen Anzahl an Prozessen. Wenn die anderen Prozesse als Kommunikationspartner entfallen und somit deren Eingabedaten nicht mit in die Berechnung einfließen, ergeben einige Berechnungen unter Umständen andere Werte. Dies ist zumindest dann der Fall, wenn sich die Ein- und/oder Ausgabedaten der Prozesse unterscheiden. Denkbar ist auch, dass eine PUT in allen Prozessen mit den gleichen Eingabedaten startet und auch die Ausgabedaten vor Ende der PUT synchronisiert werden, wobei lediglich einzelne Berechnungen auf die verfügbare Anzahl Prozesse verteilt werden. In so einem Fall könnte die PUT unabhängig von Anzahl der Prozesse stets das gleiche Ergebnis liefern. Dann wäre eine Reproduktion auch mit weniger Prozessen möglich – oder sogar mit mehr, wenn händisch die Dateien der Eingabedaten vervielfältigt werden.

Allgemein ist jedoch davon auszugeben, dass der Replay mit der identischen Anzahl an Prozessen stattfinden muss. Neben der Anzahl der Prozesse müssen auch weitere

Konfigurationen übereinstimmen. So lassen sich in MPI etwa Prozessuntermengen in Gruppen oder sogenannten Kommunikatoren zusammenfassen und anordnen. Auch diese müssen im Test identisch angeordnet sein wie beim Capture, um eine korrekte Reproduktion zu erreichen. Auch kann es einen Unterschied machen, auf wie viele Knoten oder Prozessoren die Prozesse verteilt sind.

In den meisten Fällen dürfte diese Anforderung beim Testen von Klimamodellen unproblematisch sein. Übliche Ende-zu-Ende-Tests, die für das Regressionstesten eingesetzt werden, lassen sich auch mit einer niedrigen Anzahl von Prozessen (< 10) ausführen. Werden diese als Vorlage für das Capture verwendet, bleibt der Ressourcenverbrauch des Tests überschaubar. MPI-Konfigurationen lassen sich auf mit dem FortranTestGenerator erstellte Tests übertragen, in dem die notwendigen Anweisungen in den generierten Test oder bereits in das Template eingefügt werden. Sollen dagegen Situationen reproduziert werden, die nur bei Verwendung sehr vieler Prozesse (z.B. > 1.000) auftreten, stößt der C&R-Ansatz an seine Grenzen, da sich die produzierte Datenmenge entsprechend der Anzahl der Prozesse erhöht. Zudem ist es in der Regel nicht möglich, den Einfluss unterschiedlicher Prozessanzahlen oder MPI-Konfigurationen auf das Ergebnis bei Verwendung derselben Eingabedaten zu testen.

10.3.2 Kommunikationsmuster

Kommunizieren mehrere MPI-Prozesse im Laufe einer PUT miteinander, muss diese Kommunikation für eine korrekte Reproduktion der PUT-Ausführung ebenfalls korrekt reproduziert werden. Der hier beschriebene und mit dem FortranTestGenerator umgesetzte C&R-Ansatz geht davon aus, dass dies der Fall ist, wenn die PUT mit derselben Prozessanzahl und MPI-Konfiguration ausgeführt wird, wobei jeder Prozess seine spezifischen Eingabedaten verwendet. Voraussetzung hierfür ist, dass eine PUT nur mit sich selbst kommuniziert, das bedeutet, dass korrespondierende Sende- und Empfangsoperationen oder Kollektivoperationen sich jeweils in der PUT bzw. in einer ihrer Unterprozeduren befinden. Zudem ist es von Vorteil, wenn eine zu reproduzierende Ausführung x der PUT nur mit jeweils korrespondierenden Ausführungen in anderen Prozessen kommuniziert, wie in Abbildung 10.1a dargestellt. Das bedeutet z.B., dass die erste Ausführung der PUT in Prozess 0 nur mit der ersten Ausführung in Prozess 4 kommunizieren sollte, jedoch nicht mit der dritten Ausführung. Beim Replay wird die PUT in allen Prozessen in der Regel einmal ausgeführt, d.h. hier kann die Kommunikation nur zwischen dieser ersten Ausführung stattfinden (Abbildung 10.1b).

Sollte es nicht der Fall sein, dass korrespondierende Ausführungen der PUT miteinander kommunizieren, sollte zumindest vorhersehbar sein, welche Ausführungen miteinander kommunizieren. Ließe sich in so einem Fall die Capturebedingung als

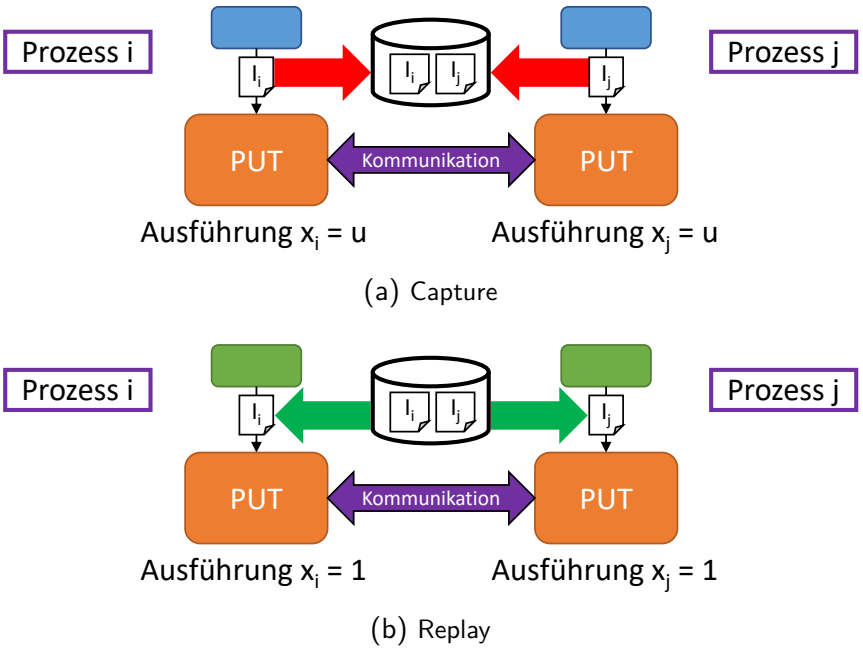
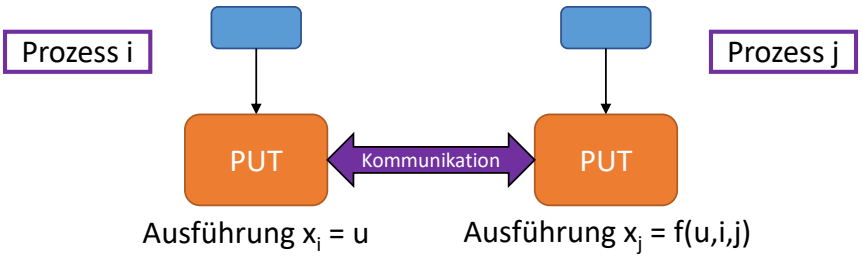


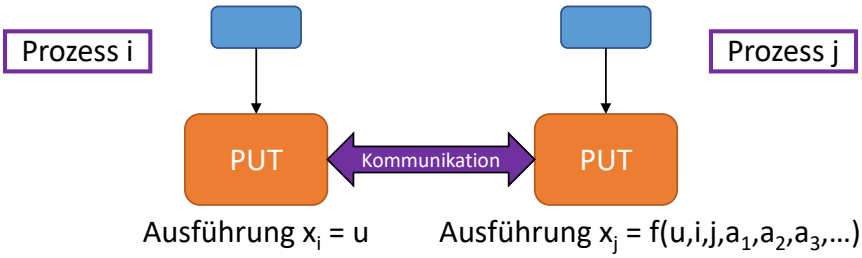
Abbildung 10.1: Capture & Replay bei interner MPI-Kommunikation

einfache Funktion der Prozessnummer formulieren, wäre es möglich, im Test genau diese kommunizierenden Ausführungen in den einzelnen Prozessen zu reproduzieren (Abbildung 10.2a). Hängt es jedoch von vielen unbekannten oder schwer zu bestimmenden Faktoren ab, welche Ausführungen miteinander kommunizieren (Abbildung 10.2b), stößt der Ansatz an seine Grenzen. Ebenfalls schwierig ist es, wenn eine Ausführung der PUT in einem Prozess mit mehreren Ausführungen in einem anderen Prozess kommuniziert (Abbildung 10.2c) oder wenn die PUT mit einer anderen Prozedur kommuniziert (Abbildung 10.2d). Grundsätzlich ließen sich auch solche Situationen mit dem C&R-Ansatz reproduzieren, jedoch wäre dies mit dem Fortran-TestGenerator nur mit manuellen Eingriffen möglich.

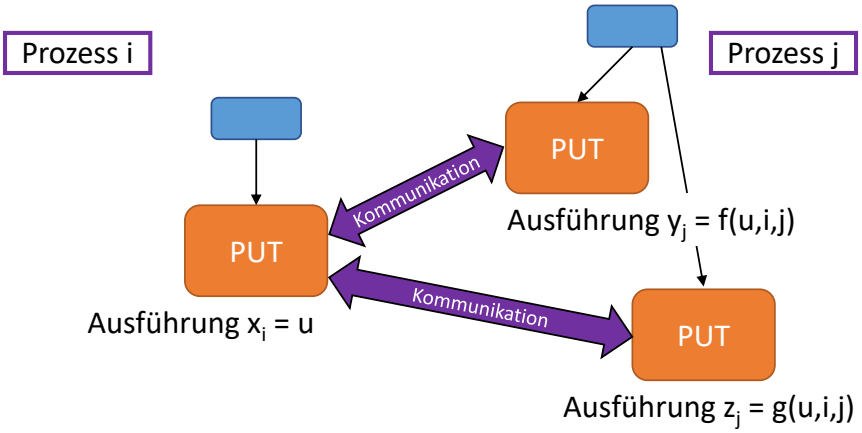
Die beschriebenen Fälle sind eher theoretischer Natur und in Klimamodellen kaum anzutreffen. Die Parallelisierung von Klimamodellen ist im Wesentlichen datenbasiert und beruht auf *Gebietszerlegung (domain decomposition)*, d.h. jeder Rechenprozess berechnet dieselben Gleichungen für einen ihm zugewiesenen Teil des Gitters (siehe auch Abschnitt 2.3.4). Einzelne Algorithmen werden dabei meist innerhalb einer Prozedur realisiert, so dass die Kommunikation zwischen den beteiligten Prozessen ausschließlich intern innerhalb der Prozedur stattfindet. Zwar gibt es auch Formen von funktions- bzw. aufgabenbasierter Parallelisierung *(task parallelisation)*, doch auch in diesen Fällen ist es in der Regel so, dass einzelne fachliche Prozeduren nicht miteinander kommunizieren, sondern dass der Datenaustausch zwischen den Prozessen intern in übergeordneten Prozeduren stattfindet.



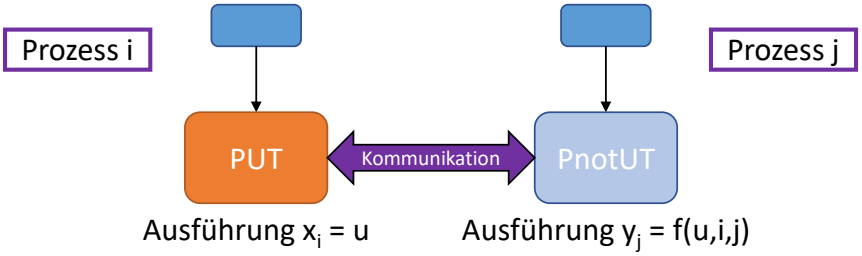
(a) Vorhersehbare Kommunikation



(b) Unvorhersehbare Kommunikation

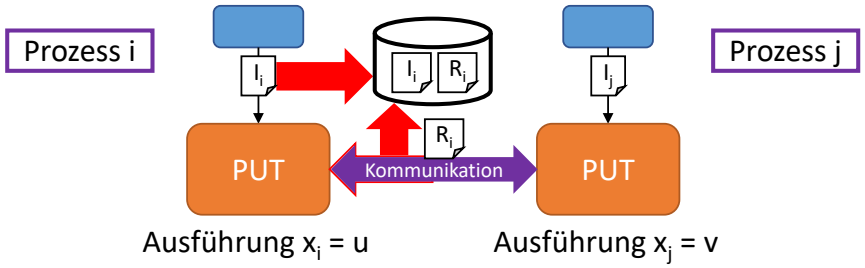


(c) Mehrfache Kommunikation

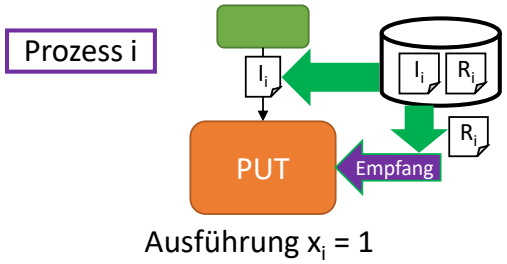


(d) Kommunikation mit anderer Prozedur

Abbildung 10.2: Für C&R schwierige Kommunikationsmuster



(a) Capture der per MPI empfangenen Daten



(b) Replay der per MPI empfangenen Daten

Abbildung 10.3: C&R der MPI-Kommunikation

10.3.3 C&R der MPI-Kommunikation

Sollte es entgegen dem vorangehend geschilderten Regelfall doch einmal nicht möglich sein, das Replay mit derselben Anzahl Prozesse und der gleichen MPI-Konfiguration durchzuführen wie das Capture, wäre eine mögliche Lösung, das Capture & Replay auf die per MPI ausgetauschten Daten zu erweitern. Denkbar wäre dies in Situationen, in denen gezielt nur ein Prozess (bzw. eine definierte Untermenge von Prozessen) untersucht werden soll. Wie in Abbildung 10.3 dargestellt müsste hierzu ein Mechanismus geschaffen werden, der beim Capture neben den übrigen Eingabedaten auch die per MPI empfangenen Daten aufgezeichnet. Die aufgezeichneten Daten (R_i in Abbildung 10.3) müssten beim Replay erneut „gesendet“ werden, mit dem Unterschied, dass sie nicht von einem anderen Prozess, auf dem die gleiche Anwendung läuft, gesendet werden, sondern vom C&R-System. Um eine Reproduktion der PUT in dem zu untersuchenden Prozess zu erreichen, müssen nur die empfangenen Daten aufgezeichnet werden, da nur diese das weitere Verhalten beeinflussen können. Zur Validierung könnte es dennoch nützlich sein, auch die gesendeten Daten aufzuzeichnen, um mit diesen die im Test gesendeten Daten zu vergleichen.

10.3.4 Masterarbeit Tareq Kellyeh

Tareq Kellyeh hat das C&R von MPI-Kommunikation im Rahmen einer durch Julian Kunkel und mich betreuten Masterarbeit erprobt (Kellyeh, 2018). Dabei sind zwei Bibliotheken entstanden. Zum einen eine MPI-Aufzeichnungsbibliothek (*MPI Tracing Library*), die beim Capture in die Originalanwendung eingebunden wird. Diese Bibliothek implementiert Wrapperprozeduren für ausgewählte MPI-Operationen, in denen die übertragenen Daten zunächst im Dateisystem gespeichert werden, bevor sie an die Prozeduren der eigentlichen MPI-Bibliothek übergeben werden (Abbildung 10.4). Der MPI-Standard sieht die Erstellung derartiger Wrapperbibliotheken explizit vor und bietet dafür das *MPI Profiling Interface* an (MPI, 2015, S. 561ff). Zum anderen hat Kellyeh eine MPI-Dummy-Bibliothek erstellt, die beim Aufruf der MPI-Empfangsoperationen die aufgezeichneten Daten lädt und an den Testprozess überträgt.

Die Aufzeichnungsbibliothek enthält zusätzlich zu den MPI-Wrapperprozeduren Steuerprozeduren, die in den Quelltext der Originalanwendung eingefügt werden, um die Datenaufzeichnung zu aktvieren und zu beenden. Dadurch kann der zu testenden Codeabschnitt, innerhalb dem die MPI-Kommunikation aufgezeichnet wird, festgelegt werden. Zudem können weitere Eingabedaten durch manuelles Hinzufügen entsprechender Prozeduraufrufe aufgezeichnet werden, um eine Isolierung des zu testenden Codeabschnitts zu ermöglichen. Als Dateiformat hat Kellyeh den für wissenschaftliche Daten gebräuchlichen HDF5-Standard (Folk u. a., 2011) gewählt, weshalb sein System den Namen *HDF5 MPI Capture-Replay (H5MR)* bekam. Die Bibliotheken wurden in C geschrieben, eine Fortran-Schnittstelle existiert noch nicht.

10.3.5 Projektarbeit Marcel Heing-Becker

Einen ähnlichen Ansatz wie Kellyeh hat auch Marcel Heing-Becker in einer ebenfalls von mir und Julian Kunkel betreuten Projektarbeit umgesetzt. Heing-Becker hat dies in Form einer Wrapperbibliothek umgesetzt, die sowohl über einen Capture- als auch einen Replaymodus verfügt, welche zur Laufzeit aktiviert werden können. Auch diese wurde in C geschrieben, aber zusätzlich mit einer Fortran-Schnittstelle versehen. Dadurch war es möglich, das MPI Capture & Replay in den FortranTestGenerator zu integrieren. Zu diesem Zweck hat Heing-Becker ein FTG-Template erstellt, in dem die entsprechenden Aufrufe der Steuerprozeduren der Bibliothek, die das Capture bzw. Replay aktivieren, eingefügt wurden. Erprobt wurde dieser Ansatz an einer ausgewählten Prozedur aus dem Klimamodell ICON. Beim Capture wurde das Modell mit vier Prozessen ausgeführt, anschließend war ein erfolgreiches Replay mit einem Prozess möglich.

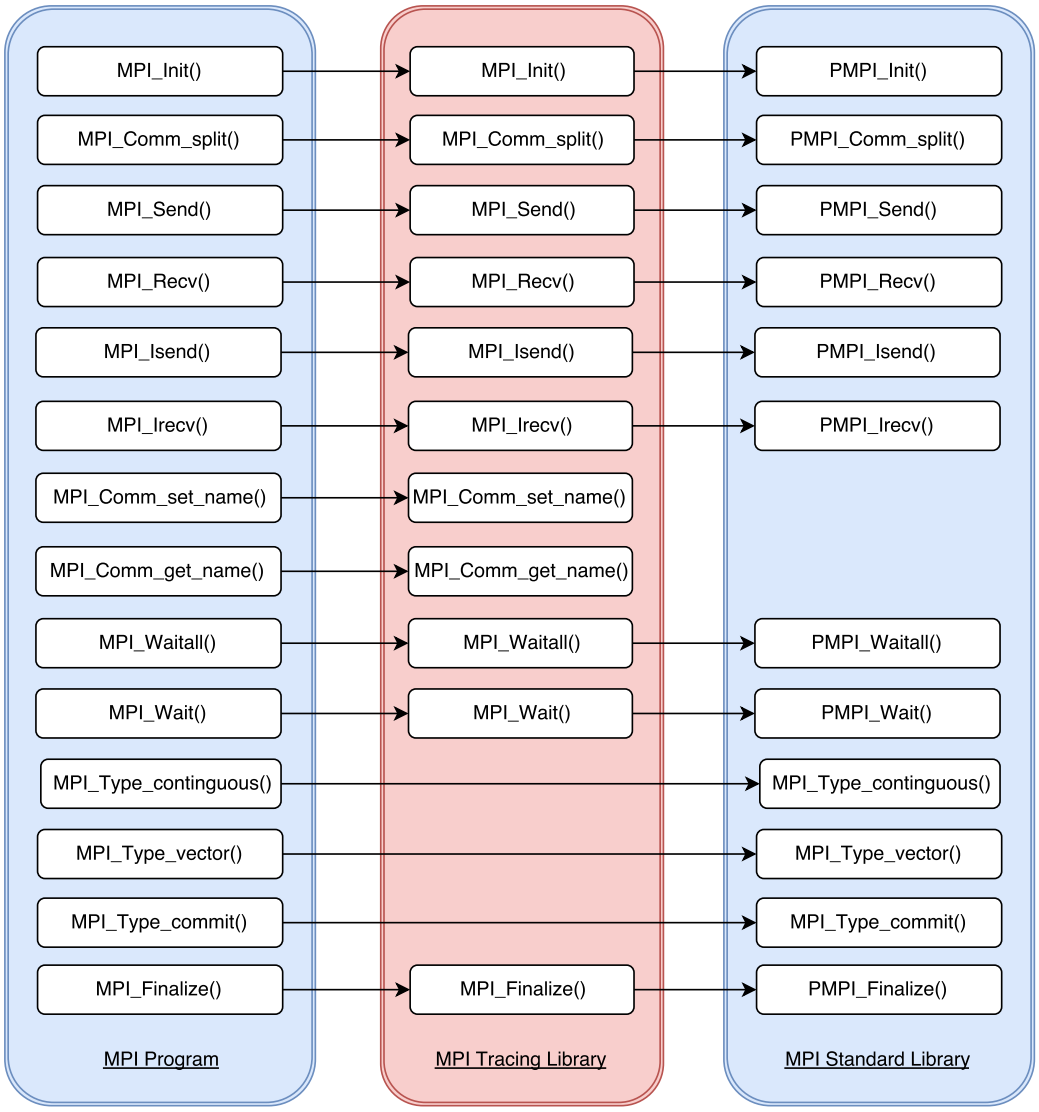


Abbildung 10.4: MPI Tracing Library (Kellyeh, 2018)

10.4 Zusammenfassung

In diesem Kapitel wurden grundlegende Grenzen des C&R-Ansatzes sowie Beschränkungen des FortranTestGenerators diskutiert. Die wichtigste grundlegende Einschränkung ist die Beschränkung auf Testfälle, die sich aus der Originalanwendung heraus aufzeichnen lassen bzw. aus aufgezeichneten Daten ableiten lassen. Zentrale Einschränkungen des FortranTestGenerators sind u.a. die Fokussierung auf Variablen und KOMPONENTEN mit primitiven Datentypen, die fehlende bzw. eingeschränkte Unterstützung von rekursiven und anderer dynamischen Datenstrukturen, die nicht vorhandene Unterstützung von SAVE-Variablen als Eingabedaten sowie die rein additive Instrumentierung.

Neben diesen Einschränkungen wurden auch Probleme diskutiert, die in MPI-parallelen Anwendungen beim C&R auftreten können. Als Lösungsweg wurde die Ausweitung des C&R auf Daten, die per MPI übertragen werden, vorgeschlagen sowie zwei Studentenprojekte vorgestellt, die diesen Ansatz prototypisch umgesetzt haben.

Kapitel 11

Erprobung

Neben der Konstruktion eines Artefakts ist dessen Erprobung im Einsatzkontext essenzieller Bestandteil der Design-Science-Methode (vgl. Wieringa, 2014, S. 3). Die in dieser Arbeit konstruierten Artefakte sind zum einen die in Kapitel 8 beschriebene Methode zur unterstützten Erzeugung von Unittests für Klimamodelle auf Grundlage des Capture-&-Replay-Ansatzes und zum anderen die in Kapitel 9 beschriebene Umsetzung dieses Konzepts in Form des Softwarewerkzeugs FortranTestGenerator (FTG). Da die vorgeschlagene Methode nur in Form ihrer Umsetzung auf den Einsatzkontext wirken kann, konzentriert sich die in diesem Kapitel beschriebene Erprobung auf den FortranTestGenerator. Einige der dabei gewonnenen Erkenntnisse lassen sich durch Generalisierung auf die beschriebene Methode übertragen, andere sind spezifisch für das vorliegende Softwarewerkzeug.

Roel J. Wieringa (2014, S. 31) unterscheidet zwischen *Validierung* und *Evaluation*. Validierung definiert er als Erprobung eines Artefakts unter kontrollierten Laborbedingungen, um Vorhersagen treffen zu können, wie es sich im Einsatzkontext verhalten würde. Evaluation ist dagegen die Beobachtung des Einsatzes eines Artefakts im realen Kontext.

> *„Validation and evaluation are different research goals that require different research approaches. The goal of validation is to predict how an artifact will interact with its context, without actually observing an implemented artifact in a real-world context. Validation research is experimental and is usually done in the laboratory.*
>
> [...]
>
> *The goal of evaluation research, by contrast, is to investigate how implemented artifacts interact with their real-world context. Evaluation research is field research of the properties of implemented artifacts.“*

Zur Validierung im Sinne Wieringas wurden mit Hilfe des FortranTestGenerators Tests für Prozeduren von vier verschiedenen Klimamodellen erzeugt. Zum einen wird

so die Funktionsfähigkeit des Werkzeugs demonstriert, zum anderen wurden dabei verschiedene quantitative Daten erhoben, um so die Tauglichkeit des Werkzeugs bewerten zu können.

Zur Evaluation wurde eine Benutzerstudie mit Entwicklern des *ENIAC-Projekts* durchgeführt, die den FortranTestGenerator verwendet haben, um Unittests für das Klimamodell ICON zu erstellen. Das ENIAC-Projekt ist ein reales Entwicklungsprojekt, in dem Teile des ICON-Modells zur Ausführung auf GPUs portiert werden. Zum Zwecke der Studie wurden die Entwickler des Projekts zur Tauglichkeit und Benutzbarkeit des FortranTestGenerators befragt.

Die in diesem Kapitel beschriebene Erprobung dient somit in erster Linie den FortranTestGenerator hinsichtlich der ersten zwei Anforderungen an anwendungsorientierte Software zu überprüfen: aufgabenorientierte Funktionalität und benutzergerechte Handhabung. Die Erfüllung der dritten Anforderung, Anpassbarkeit, lässt sich im Rahmen dieser Arbeit nur ansatzweise bewerten. Zwar mussten die Entwickler der Benutzerstudie auch einige Anpassungen vornehmen, da der FortranTestGenerator jedoch bereits in vielen Aspekten auf die Anforderungen der teilnehmenden Entwickler ausgerichtet ist, waren die Anpassungen sehr begrenzt. Auch für die Validierung waren Anpassungen notwendig, um den FortranTestGenerator an verschiedenen Klimamodellen zu erproben, da diese jedoch durch mich selbst durchgeführt wurden, lässt sich auch hier nur eingeschränkt beurteilen, wie problemlos diese für andere BenutzerInnen wären.

In Abschnitt 11.1 werden die zur Validierung durchgeführten Experimente beschrieben und die allgemeine Funktionsfähigkeit des FortranTestGenerators untersucht. In Abschnitt 11.2 erfolgt eine quantitative Auswertung dieser Experimente. Die durchgeführte Benutzerstudie steht im Mittelpunkt von Abschnitt 11.3. Wenn in diesem Kapitel von *Unterprozeduren* die Rede ist, schließt dies jeweils die PUT als unechte Unterprozedur von sich selbst mit ein.

11.1 Funktionsfähigkeit

Um die grundsätzliche Funktionsfähigkeit des FortranTestGenerators zu erproben, wurden mit diesem Capture- und Replaycode für einzelne Prozeduren verschiedener Klimamodelle erstellt, Capture und Replay ausgeführt und überprüft, ob die *Reproduktion* der Prozedurausführung erfolgreich war. Als Reproduktion wird hier die Ausführung der Prozedur im Testprogramm mit aus der Originalauswendung aufgezeichneten Eingabedaten bezeichnet. Die Reproduktion gilt als erfolgreich, wenn sie im Test dieselben Ergebnisse erzeugt wie in der Originalanwendung, d.h. wenn sämtliche ermittelten Ausgabevariablen nach Ausführung der Prozedur die gleichen Werte

Modell	PC	Mistral
ICON	✔	✔
MOM6	✔	
NICAM		✔
CESM2	✔	

Tabelle 11.1: Verwendete Modelle und Rechnerplattformen

haben. Hierzu werden bei der Ausführung der Prozedur in der Originalanwendung ebenfalls die Ausgabedaten der Prozedur aufgezeichnet. Innerhalb des mit dem FortranTestGenerators erzeugten Testprogramms werden die Ergebnisse der Prozedur mit den aufgezeichneten Ausgabedaten verglichen. Werden dabei keine Abweichungen festgestellt, gilt der Test und damit die Reproduktion der Prozedurausführung als erfolgreich.

Als Testmodelle wurden das integrierte Atmosphären- und Ozeanmodell *ICON* (Zängl u. a., 2015), das Ozeanmodell *MOM6* (GitHub, MOM6), das Atmosphärenmodell *NICAM* (Satoh, Tomita u. a., 2014) und das Erdsystemmodell *CESM2* (Hurrell u. a., 2013; Lauritzen, Nair u. a., 2018) verwendet. Als Rechnerplattformen wurde zum einen ein handelsüblicher PC mit Ubuntu-Linux sowie der Hochleistungsrechner Mistral am Deutschen Klimarechenzentrum verwendet. Die Spezifikationen der Rechner sind in Anhang E.1 aufgeführt. In Tabelle 11.1 ist dargestellt, welche Modelle auf welcher Plattform verwendet wurden.

Bei der Auswahl der Modelle spielte die Verfügbarkeit des Quellcodes eine wesentliche Rolle. Zu den Modellen ICON und NICAM hatte ich durch die im Rahmen dieser Arbeit entstandenen Kontakte Zugang. MOM6 und CESM2 sind hingegen als Open-Source-Modelle öffentlich verfügbar. Die Anzahl der Modelle erscheint ausreichend, um die grundsätzliche Funktionsfähigkeit im angestrebten Einsatzkontext Klimamodell nachzuweisen. Allerdings ist nicht auszuschließen, dass der Einsatz bei anderen Modellen mit spezifischen Problemen verbunden ist, die bei den hier getesteten nicht auftreten. Aufgrund des Entwicklungsstands der Werkzeuge traten bei jedem Modell Fehler innerhalb von fcg und FTG auf, die zuvor unentdeckt waren und zunächst behoben werden mussten, bevor Capture- und Replaycode erfolgreich erzeugt werden konnten.

Auch die Wahl der Rechnerplattform war durch ihre Verfügbarkeit bestimmt, wobei die gewählten Plattformen gute Repräsentanten für die allgemeinen Zielplattformen Entwickler-PC und Hochleistungsrechner darstellen. Das Kompilieren und Ausführen eines in Quelltextform vorliegenden Klimamodells ist nicht einfach. Die Buildprozesse sind häufig kompliziert und die mitgelieferten Skripte enthalten häufig spezifische Annahmen über die technische Umgebung wie die Rechnerplattform, Zielpfade, vor-

handene Bibliotheken etc., welche an die eigene Umgebung angepasst werden müssen. Daher wurden drei der vier Modelle nur auf jeweils einer Plattform verwendet, wobei die Einfachheit der notwendigen Anpassungen ausschlaggebend war. Insgesamt wurden so jedoch alle vier Modelle und beide Plattformen erprobt.

Folgende Softwareversionen wurden für die hier dokumentierten Versuche verwendet:

FortranCallGraph v1.8.3

FortranTestGenerator v1.11.3

Cheetah3 3.2.1a0

Serialbox2 v2.5.3

Analysiert wurde jeweils der vom Präprozessor vorverarbeitete Code. Instrumentiert wurde der Originalquellcode. Die Auswahl der Prozeduren erfolgte nach dem Gesichtspunkt, dass unterschiedliche Ebenen innerhalb des Aufrufgraphs vertreten sind. Zudem mussten passende E2E-Tests verfügbar sein, die die entsprechenden Prozeduren aufrufen. Diese wurden jeweils den mit den Modellen ausgelieferten Testsammlungen entnommen. Die Anzahl der verwendeten MPI-Prozesse richtete sich nach den Standardeinstellungen der jeweiligen E2E-Tests. Aufgezeichnet wurde jeweils die erste Ausführung der zu testenden Prozedur.

Auch wenn diese Arbeit vornehmlich auf Prozeduren, die auf niedriger bis mittlerer Ebene im Aufrufgraphen einer Anwendung stehen, abzielen, da hier der Vorteil von Unittests gegenüber Ende-zu-Ende-Tests besonders zum Tragen kommt, wurden im Rahmen dieser Erprobung auch Tests für hochrangige Prozeduren erstellt. Der Grund hierfür ist, dass Prozeduren höherer Ebene mehr Abhängigkeiten haben und mehr Eingabedaten benötigen, die Reproduktion einer Ausführung somit schwieriger bzw. fehleranfälliger ist. Gelingt die Reproduktion dennoch, lässt sich daraus noch stärker die allgemeine Funktionsfähigkeit von Methode und Werkzeug ableiten als durch die Reproduktion von Prozeduren mittlerer und niedriger Ebene.

11.1.1 ICON

Versuchsaufbau

- Modellversion: *icon-2.4.0* aus dem nicht öffentlichen git-Repository (siehe auch MPI-M, Obtain ICON)
- Rechnerplattformen: PC, Mistral (Die Versuche wurden zunächst auf dem PC durchgeführt und bei Erfolg auf der Mistral wiederholt.)

- Compiler: GNU Fortran 7 (PC); GNU Fortran 6 (Mistral)
- fcg/FTG-Konfigurationsdateien: siehe Anhang E.2.1
- Templates: IconStandalone, IconJsbachMock
- Prozeduren: 8, siehe Tabelle 11.2
- MPI-Prozesse: 2 (PC); 2 und 24 (Mistral)

Ergebnisse

Prozedur	Modul	Template	E2E-Test
`compute_airmass`	`mo_nh_dtp_interface`	IconStandalone	atm_amip_noforcing_notransport_test
`integrate_nh`	`mo_nh_stepping`	IconJsbachMock	atm_amip_noforcing_notransport_test
`interface_full`	`mo_jsb_interface`	IconStandalone	atm_amip_noforcing_notransport_test
`solve_nh`	`mo_solve_nonhydro`	IconStandalone	atm_amip_noforcing_notransport_test
`step_advection`	`mo_advection_stepping`	IconStandalone	atm_amip_test
`update_diag_state`	`mo_ha_diag_util`	IconStandalone	atm_jww_hs_test
`update_ho_params`	`mo_ocean_physics`	IconStandalone	test_ocean_omip_10days
`update_surface`	`mo_surface`	IconJsbachMock	atm_amip_test

Tabelle 11.2: Getestete Prozeduren aus ICON
grün: erfolgreiche Reproduktion, gelb: quasi-erfolgreiche Reproduktion, rot: fehlgeschlagene Reproduktion

- Vier Prozeduren konnten mit kleineren Anpassungen reproduziert werden: `compute_airmass`, `solve_nh`, `update_diag_state` und `update_ho_params`.
- Die Reproduktion von `step_advection` war quasi-erfolgreich. Bei Ausführung mit zwei Prozessen traten keinerlei Abweichungen auf, bei der Ausführung mit 24 Prozessen lediglich in einer Variable. Diese Abweichungen ließen sich jedoch auch zwischen zwei E2E-Test-Ausführungen beobachten. Hier scheint also allgemein ein nichtdeterministisches Verhalten vorzuliegen, mutmaßlich wegen einer unvollständigen Initialisierung.
- Der Test mit `interface_full` war nicht erfolgreich. Diese SUBROUTINE gehört zu dem in ICON integrierten Landmodell *JSBACH*. Dieses verfügt über eine eher objektorientierte Architektur und macht ausgiebig von Vererbung Gebrauch, was zu dynamischen Daten- und Programmstrukturen führt. Diese können von fcg/FTG (noch) nicht analysiert und reproduziert werden. Der Test endete mit einer Fehlermeldung einer eingebauten Prüfung, dass eine Variable einen falschen Typ hätte. Zudem war ersichtlich, dass der ermittelte Aufrufgraph sowie die ermittelte Menge der verwendeten Variablen unvollständig waren.

- Die zwei SUBROUTINEN `integrate_nh` und `update_surface` rufen jeweils `interface_full` auf (unter dem INTERFACE-Namen `jsbach_interface`). Diese konnten erfolgreich reproduziert werden, nachdem manuell ein Testdouble für JSBACH erstellt wurde. Insbesondere für die SUBROUTINE `integrate_nh` ist die erfolgreiche Reproduktion bemerkenswert. Sie ist eine der Hauptprozeduren innerhalb Zeitschleife des ICON-Atmosphärenmodells und befindet sich sehr weit oben im Aufrufgraphen des Modells. Sie ruft selbst über 1.000 andere Prozeduren direkt oder indirekt auf und benötigt über 4.000 von fcg ermittelte Eingabevariablen (siehe auch Tabelle 11.7 auf Seite 252).

Notwendige Anpassungen

Folgende allgemeine Anpassungen wurden an dem aus dem git-Repository entnommen Code vorgenommen, bevor einzelne Prozeduren isoliert wurden:

1. Anpassung der Buildskripte, um das Kompilieren und Ausführen auf dem PC zu ermöglichen und Serialbox2 einzubinden.
2. Aus den Quellcodedateien mo_init_vgrid.f90 und mo_albedo.f90 wurde jeweils ein unsichtbares Steuerzeichen entfernt, das Python beim Einlesen der Datei zum Absturz brachten.
3. In den Modulen `mo_nwp_land_types`, `mo_communication_types` und `mo_communication_types_orig` wurden `NULL()`-Initialisierungen von Zeigervariablen ergänzt.
4. Aus den häufig verwendeten Typen `t_comm_pattern_orig`, `t_comm_pattern_collection_orig`, `vector` und `vector_ref` wurde das `PRIVATE`-Schlüsselwort entfernt.
5. Aus dem häufig verwendeten Modul `mo_master_config` wurde eine `PROTECTED`-Deklaration entfernt.

Folgende besondere Konfigurationen von fcg waren zudem notwendig:

1. Module externer Bibliotheken sowie diverse Module nicht benötigter Modellteile wurden als zu ignorieren deklariert (siehe auch Anhang E.2.1).
2. Dem abstrakten Typ `t_comm_pattern` wurde der konkrete Untertyp `t_comm_pattern_orig` zugewiesen, d.h. dass fcg alle Vorkommen von `t_comm_pattern` so behandelt als würde tatsächlich `t_comm_pattern_orig` verwendet werden (siehe auch Abschnitt 10.2.2). ICON enthält zwei verschiedene Infrastrukturmodule als MPI-Abstraktionsschicht: ein älteres und

ein neueres. Über die Modellkonfiguration lässt sich die zu verwendende Variante auswählen, was im Code u.a. mit Hilfe einer polymorphen Struktur bzgl. des Typs `t_comm_pattern` realisiert ist, d.h. abhängig von der Modellkonfiguration ist ein `t_comm_pattern` zur Laufzeit entweder ein `t_comm_pattern_orig` (alte Version) oder ein `t_comm_pattern_yaxt` (neue Version). Da die für das Capture verwendeten E2E-Tests alle mit der alten Version arbeiten, wurde fcg entsprechend konfiguriert.

3. Analog wurde dem abstrakten Typ `t_comm_pattern_collection` der konkrete Untertyp `t_comm_pattern_collection_orig` zugewiesen.
4. Für die Typen `datetime` und `timedelta` wurde festgelegt, dass von diesen immer alle KOMPONENTEN als verwendet gelten. Diese Typen gehören zu der Fortran-Schnittstelle der in C implementierten Infrastrukturbibliothek für Zeitberechnungen innerhalb des Modells. Da der Zugriff auf ihre KOMPONENTEN innerhalb der C-Prozeduren nicht analysiert werden kann, werden sie immer vollständig aufgezeichnet.

Das verwendete Template *IconStandalone* enthält die folgenden Besonderheiten:

1. Für die Ausgabe von Meldungen und Fehlerbehandlung werden die entsprechende ICON-Infrastrukturprozeduren verwendet.
2. Alle MPI-Operationen sind durch entsprechende ICON-Infrastrukturprozeduren ersetzt worden.
3. Einige Variablen der Infrastruktur halten von MPI zufällig erzeugte Identifikationsnummern (*handles*) von Kommunikatoren. Beim Replay werden für diese Variablen anstelle der aufgezeichneten Daten entsprechende aktuelle IDs gesetzt. Diese Variablen werden zudem bei der Ergebnisvalidierung ausgelassen.
4. Variablen, deren Werte abhängig von der aktuellen Uhrzeit sind, werden bei der Ergebnisvalidierung ausgelassen.
5. Eine Konfigurationsvariable der externen Bibliothek `mtime` wird beim Capture ausgelesen und beim Replay gesetzt.
6. Die Vorlage für das Testprogramm enthält einige ICON-spezifische Imports.

Zusätzlich zu diesen enthält das Template *IconJsbachMock* weitere Besonderheiten, die nachfolgend im Zusammenhang mit `integrate_nh` erläutert werden.

Folgende weitere Anpassungen waren für die einzelnen (erfolgreichen) Prozeduren notwendig:

`compute_airmass` *keine*

integrate_nh 1. `integrate_nh` ruft indirekt zwei Subroutinen namens `omp_loop_cell_prog` und `omp_loop_cell_diag` auf. Diese Subroutinen nehmen jeweils eine andere Subroutine als Argument entgegen und rufen diese auf. Da solche Prozedurargumente von fcg (noch) nicht analysiert werden können, wurden manuell die Prozeduren gesucht, die `omp_loop_cell_prog` und `omp_loop_cell_diag` als Argumente übergeben werden und vorübergehend explizite Aufrufe zu diesen Prozeduren in den Code von `omp_loop_cell_prog` und `omp_loop_cell_diag` eingefügt. Dadurch konnte ein korrekter Aufrufgraph erzeugt und eine Analyse der verwendeten Prozeduren durchgeführt werden.

2. Das Modul `src_turbdiff` wird auch in einem anderen Modell verwendet, hat dort aber einen anderen Namen. Die Unterscheidung erfolgt per Präprozessor-Fallunterscheidung. Dies ist für die Codeanalyse kein Problem, da hier der vom Präprozessor vorverarbeitete Code verwendet wurde. Jedoch konnte so der richtige Einfügepunkt für den Exportcode im Originalcode des Moduls nicht gefunden werden. Dies wurde behoben, indem die Zeile mit dem alternativen Namen auskommentiert wurde.

3. Ein ähnliches Problem bestand für das Modul `mo_interface_echam_ocean`. Hier befanden sich sogar zwei vollständige alternative Module in der entsprechenden Quellcodedatei. Die nicht benötigte Version wurde ebenfalls auskommentiert.

4. In den Modulen `mo_communication_orig`, `mo_ext_data_types` und `mo_model_domain` wurden zusätzliche `NULL()`-Initialisierungen von Zeigervariablen ergänzt.

5. Für die Subroutine `interface_full` bzw. das Interface `jsbach_interface` wurde ein Testdouble erstellt. Abbildung 11.1 zeigt dessen ebenfalls auf Capture & Replay basierende Funktionsweise. Es besteht aus einem Modul `mo_jsbach_interface_mock`, welches ebenfalls das Interface `jsbach_interface` und eine Subroutine `interface_full` enthält. Zudem enthält es Subroutinen, mit denen es sich in einen Capture- oder in einen Replaymodus versetzen lässt. Im Capturemodus wird beim Aufruf von `interface_full` die Original-Subroutine aus dem Modul `mo_jsb_interface` aufgerufen und anschließend dessen Ausgabedaten aufgezeichnet und als eigene Ausgabedaten verwendet. Im Replaymodus wird das Original nicht aufgerufen, stattdessen werden die zuvor aufgezeichneten Ausgabedaten geladen. Das Mockmodul ist auch auf GitHub erhältlich (GitHub, jsbach-mock)[20].

[20] Meszaros (2006, S. 523ff) definiert einige unterschiedliche Formen von Testdoubles, die auch von Fowler (2007) aufgegriffen werden. Nach diesen Definitionen handelt es sich bei dem hier

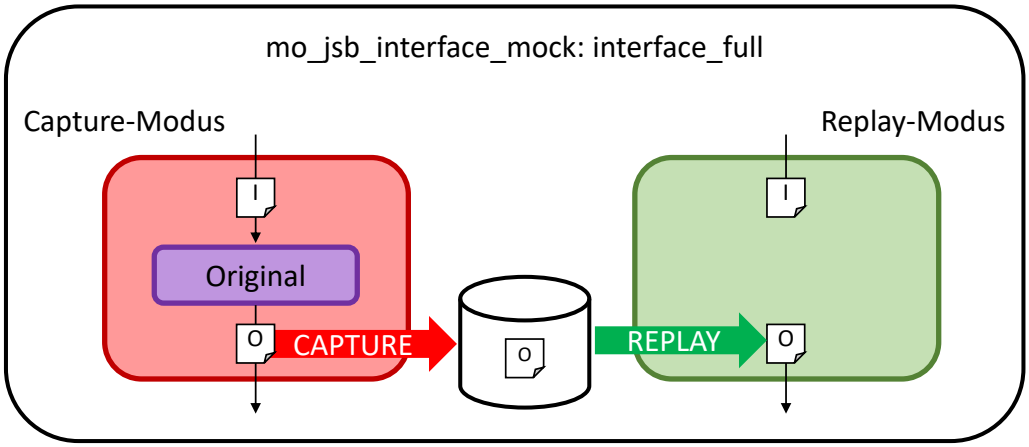


Abbildung 11.1: JSBACH-Testdouble
I: Eingabedaten, O: Ausgabedaten

Zusätzlich wurde das Template IconJsbachMock erstellt. Dieses enthält die entsprechenden Anweisungen, um den Capture- bzw. Replaymodus von `mo_jsb_interface_mock` zu aktivieren. Außerdem sorgen die Template Parts captureAfterUse und exportAfterUse dafür, dass in jedem Modul, in dem die Prozedur `jsbach_interface` aufgerufen wird, diese aus `mo_jsb_interface_mock` importiert wird. Da hierzu nur eine zusätzliche `USE`-Anweisung hinzugefügt, der ursprüngliche Import von `jsbach_interface` aus `mo_jsb_interface` jedoch nicht entfernt werden kann, führt dies zunächst zu einem Compilerfehler aufgrund des doppelten Imports. Die ursprünglichen `USE`-Anweisungen müssen manuell entfernt bzw. auskommentiert oder ggf. eine Präprozessor-Fallunterscheidung eingebaut werden.

`solve_nh` *keine*

`step_advection` In dieser SUBROUTINE werden zwei auf Modulebene deklarierte Zeigervariablen jeweils einer lokalen Variable zugewiesen, was zu hängenden Zeigern führt. Dies wurde korrigiert. Zudem wurde dieser Programmierfehler zusammen mit dem Korrekturvorschlag im ICON-Ticketsystem gemeldet.

`update_diag_state` Im Modul `mo_icoham_dyn_types` wurden fehlende Zeigerinitialisierungen ergänzt.

`update_ho_params` Der Export von MODULVARIABLEN im Modul `mo_ocean_physics` führte im Modul `mo_ocean_tracer_transport_vert` zu Be-

verwendeten „JSBACH-Mock“ nicht um ein *Mock* im eigentlichen Sinne, da es selbst keine Verifikation der übergebenen Daten vornimmt, sondern eher um ein *Stub*. Dies wurde bei der Namensgebung nicht berücksichtigt.

zeichnerkollisionen, da hier `mo_ocean_physics` vollständig importiert wird. Die wurde behoben, indem der Import mit Hilfe des `ONLY`-Schlüsselwortes auf die tatsächlich benötigten Elemente eingeschränkt wurde.

`update_surface` Auch hier wurde mit Hilfe des Templates IconJsbachMock das JSBACH-Testdouble eingebunden.

11.1.2 MOM6

Versuchsaufbau

- Modellversion: Commit *78c1690e78c9fc4b676b66055b5809ba42a64573* aus dem öffentlichen GitHub-Repository *MOM6-examples* (GitHub, MOM6-examples)
- Rechnerplattform: PC
- Compiler: GNU Fortran 7
- fcg/FTG-Konfigurationsdateien: siehe Anhang E.2.2
- Templates: Standalone
- Prozeduren: 2, siehe Tabelle 11.3
- MPI-Prozesse: jeweils 1

Ergebnisse

Prozedur	Modul	Template	E2E-Test
`calculate_diagnostic_fields`	`mom_diagnostics`	Standalone	double_gyre
`int_density_dz_linear`	`mom_eos_linear`	Standalone	external_gwave

Tabelle 11.3: Getestete Prozeduren aus MOM6
grün: erfolgreiche Reproduktion

Beide Prozeduren konnten mit kleineren Anpassungen erfolgreich reproduziert werden.

Notwendige Anpassungen

Folgende allgemeine Anpassungen wurden an dem aus dem GitHub-Repository entnommen Code vorgenommen, bevor einzelne Prozeduren isoliert wurden:

1. Anpassung der Buildskripte, um das Kompilieren und Ausführen auf dem PC zu ermöglichen und Serialbox2 einzubinden.

Folgende besondere Konfigurationen von fcg waren zudem notwendig:

1. Module externer Bibliotheken sowie diverse Module nicht benötigter Modellteile wurden als zu ignorieren deklariert (siehe auch Anhang E.2.2).

Das verwendete Template *Standalone* ist ein generisches, das keine modellspezifischen Besonderheiten enthält.

Folgende weitere Anpassungen waren für die einzelnen Prozeduren notwendig:

`calculate_diagnostic_fields`

1. Einige PARAMETER wurden aufgrund veralteter Fortran-77-Syntax von fcg für Variablen gehalten. Die daraufhin fehlerhaft erzeugten Lese- und Schreib- und Vergleichsanweisungen für diese PARAMETER mussten manuell entfernt werden.
2. Aufgrund von nicht erkannten Aliasen wurden von einigen wenigen Variablen die Typen nicht richtig erkannt. Dadurch wurden leicht fehlerhafte Schreib- und Vergleichsanweisungen erzeugt, die manuell korrigiert werden mussten.
3. In den Modulen `fms_mod`, `fms_io_mod`, `memutils_mod`, `MOM_diagnostics`, `MOM_EOS`, `MOM_remapping`, `MOM_wave_speed`, `mpp_mod`, `mpp_domains_mod`, `mpp_io_mod`, `time_manager_mod` wurden von einigen Typen und Variablen die individuellen `PRIVATE`-Deklarationen entfernt.

`int_density_dz_linear` *keine*

11.1.3 CESM2

Versuchsaufbau

- Modellversion: *cesm2.1.0* aus dem öffentlichen GitHub-Repository (GitHub, CESM)
- Rechnerplattform: PC
- Compiler: GNU Fortran 7

- fcg/FTG-Konfigurationsdateien: siehe Anhang E.2.3
- Templates: Standalone, CesmStandalone
- Prozeduren: 3, siehe Tabelle 11.4
- MPI-Prozesse: jeweils 2

Ergebnisse

Prozedur	Modul	Template	E2E-Test
`dyn_run`	`dyn_comp`	CesmStandalone	fkessler
`d2a3dikj`	`d2a3dikj_mod`	CesmStandalone	fkessler
`mapn_ppm_tracer`	`mapz_module`	CesmStandalone	fkessler

Tabelle 11.4: Getestete Prozeduren aus CESM2
grün: erfolgreiche Reproduktion, rot: fehlgeschlagene Reproduktion; für den Test „fkessler“ siehe auch Lauritzen und Goldhaber, 2017

- Zwei Prozeduren konnten mit kleineren Anpassungen erfolgreich reproduziert werden: `d2a3dikj` und `mapn_ppm_tracer`
- Der Test mit `dyn_run` war nicht erfolgreich. Er konnte ausgeführt werden, jedoch wichen bei vier von insgesamt 255 aufgezeichneten Ausgangsvariablen die Werte von den aufgezeichneten Referenzdaten ab. Die Abweichungen traten jeweils in beiden MPI-Prozessen auf. Die Gründe für die Abweichungen konnten in angemessener Zeit nicht ermittelt werden.

Notwendige Anpassungen

Folgende allgemeine Anpassungen wurden an dem aus dem GitHub-Repository entnommen Code vorgenommen, bevor einzelne Prozeduren isoliert wurden:

1. Anpassung der Buildskripte, um das Kompilieren und Ausführen auf dem PC zu ermöglichen und Serialbox2 einzubinden.

Folgende besondere Konfigurationen von fcg waren zudem notwendig:

1. Module externer Bibliotheken wurden als zu ignorieren deklariert (siehe auch Anhang E.2.3).

Das verwendete Template *CesmStandalone* enthält die folgende Besonderheit:

1. Variablen, in denen die Handles von MPI-Kommunikatoren gespeichert sind, werden gesondert behandelt. Für die jeweiligen Kommunikatoren werden die Mitgliedsprozesse der zugehörigen Gruppe sowie der Rang des aktuellen Prozesses in dieser Gruppe gespeichert. Diese Informationen werden beim Replay verwendet, um die jeweiligen Kommunikatoren wiederherzustellen. Während etwa in ICON alle Kommunikatoren in einer einzelnen speziellen Initialisierungsprozedur erstellt werden, deren Aufruf Teil des Testtemplates ist, ist die Kommunikator-Erzeugung in CESM über die gesamte Modellinitialisierung verteilt. Da es dem Sinn des Unittests widersprechen würde, die gesamte Modellinitialisierung zu durchlaufen, ist stattdessen dieser Weg für die Wiederherstellung der MPI-Kommunikatoren gewählt worden. Die Variablen, die Kommunikator-Handles tragen werden zudem bei der Ergebnisvalidierung ausgelassen.
2. Die Vorlage für das Testprogramm enthält einige CESM-spezifische Imports.

Folgende weitere Anpassungen waren für die einzelnen Prozeduren notwendig:

`d2a3dikj`

1. In den Modulen `perf_mod` und `perf_utils` wurden von einigen Typen und Variablen individuelle `PRIVATE`-Deklarationen entfernt.
2. Eine (unnötige) `RETURN`-Anweisung am Ende der Prozedur wurde manuell hinter den Capturecode für die Ausgangsdaten verschoben (siehe auch Abschnitt 10.2.4).
3. Für die Konstante `r8` aus dem Modul `shr_kind_mod` musste manuell ein Import in den Capturecode eingefügt werden.
4. Für einige Variablen der MPI- sowie der GPTL-Bibliothek, die per Include-Anweisung vom Präprozessor in andere Module eingefügt worden, wurden manuell die Schreib-, Lese- und Vergleichsanweisungen entfernt.

`dyn_run`

1. In CESM existieren mehrere Subroutinen außerhalb von Modulen. Diese können von fcg noch nicht gefunden und analysiert werden (siehe auch Abschnitt 10.2.10). Daher wurden einige von Ihnen, deren Analyse zwingend notwendig erschien (`cd_core`, `geopk`, `par_xsum`, `trac2d`), jeweils mit einem Modul umschlossen und in die benutzenden Module entsprechende USE-Anweisungen eingefügt.
2. In den Modulen `perf_mod`, `perf_utils` und `physconst` wurden von einigen Variablen individuelle `PRIVATE`- und `PROTECTED`-Deklarationen entfernt.
3. Für einige Variablen der MPI- sowie der GPTL-Bibliothek, die per Include-Anweisung vom Präprozessor in andere Module eingefügt worden, wurden manuell die Schreib-, Lese- und Vergleichsanweisungen entfernt.

mapn_ppm_tracer

1. Eine (unnötige) `RETURN`-Anweisung am Ende der Prozedur wurde manuell hinter den Capturecode für die Ausgangsdaten verschoben (siehe auch Abschnitt 10.2.4).
2. Für die Konstante `r8` aus dem Modul `shr_kind_mod` musste manuell ein Import in das Testprogramm eingefügt werden.

11.1.4 NICAM

Versuchsaufbau

- Modellversion: *NICAM.16.6* aus dem nicht öffentlichen git-Repository (siehe auch NICAM, Collaborations)
- Rechnerplattform: Mistral
- Compiler: GNU Fortran 6
- fcg/FTG-Konfigurationsdateien: siehe Anhang E.2.4
- Templates: NicamStandalone
- Prozeduren: 4, siehe Tabelle 11.5
- MPI-Prozesse: jeweils 40

Ergebnisse

Prozedur	Modul	Template	E2E-Test
`dynstep`	`mod_dynstep`	NicamStandalone	GL05RL01Az78
`mp_lsc_wcdiag`	`mod_mp_lsc`	NicamStandalone	GL05RL01Az78
`vi_rhow_update_matrix`	`mod_vi`	NicamStandalone	GL05RL01Az78
`vi_small_step`	`mod_vi`	NicamStandalone	GL05RL01Az78

Tabelle 11.5: Getestete Prozeduren aus CESM2
grün: erfolgreiche Reproduktion, gelb: quasi-erfolgreiche Reproduktion

- Die Prozeduren `mp_lsc_wcdiag` und `vi_rhow_update_matrix` konnten erfolgreich reproduziert werden.
- Die Reproduktionen der Prozeduren `vi_small_step` und `dynstep` waren quasi-erfolgreich. Zwar enthielten jeweils zwei Variablen Abweichungen zu den Referenzendaten, jedoch lieferte für diese Variablen auch der E2E-Test keine einheitlichen Werte. In Bezug auf diese Variablen ist somit von einem nichtdeterministischen Verhalten auszugehen.

Notwendige Anpassungen

Folgende allgemeine Anpassungen wurden an dem aus dem GitHub-Repository entnommen Code vorgenommen, bevor einzelne Prozeduren isoliert wurden:

1. Anpassung der Buildskripte, um das Kompilieren mit dem GNU Compiler zu ermöglichen und Serialbox2 einzubinden.
2. Einige Prozeduren in NICAM enthalten lokale Variablen mit dem `SAVE`-Attribut. Deren Werte können in den Tests nicht wiederhergestellt werden (siehe auch Abschnitt 10.2.3). In den Modulen `mod_dynstep`, `mod_src` und `mod_vi` wurden daher `SAVE`-Variablen in MODULVARIABLEN umgewandelt.
3. In der SUBROUTINE `src_flux_convergence` aus dem Modul `mod_src` sowie in `vi_rhow_update_matrix` wurden Initialisierungen für die Elemente frisch alloziierter Arrays hinzugefügt, da die fehlenden Initialisierungen zu nichtdeterministischem Verhalten führten.
4. NICAM verwendet ausgiebig individuelle `PRIVATE`-Deklarationen, auch in Modulen, die bereits global als privat gekennzeichnet wurden. In diversen Modulen mussten daher individuelle `PRIVATE`-Deklarationen entfernt werden.

Folgende besondere Konfigurationen von fcg waren zudem notwendig:

1. Das Modul der MPI-Bibliothek wurde als zu ignorieren deklariert (siehe auch Anhang E.2.4).

Das verwendete Template *NicamStandalone* enthält die folgenden Besonderheiten:

1. Unterstützung für Strings
2. Unterstützung für 5-dimensionale Arrays

Folgende weitere Anpassungen waren für die einzelnen Prozeduren notwendig:

`dynstep` Eine (unnötige) `RETURN`-Anweisung am Fuß der Prozedur wurde manuell hinter den eingefügten Capturecode verschoben.

`mp_lsc_wcdiag` Eine (unnötige) `RETURN`-Anweisung am Fuß der Prozedur wurde manuell hinter den eingefügten Capturecode verschoben.

`vi_rhow_update_matrix` Eine (unnötige) `RETURN`-Anweisung am Fuß der Prozedur wurde manuell hinter den eingefügten Capturecode verschoben.

`vi_small_step` 1. Eine (unnötige) `RETURN`-Anweisung am Fuß der Prozedur wurde manuell hinter den eingefügten Capturecode verschoben.

2. Die Deklaration einer sog. Anweisungsfunktion (altes FORTRAN-77-Konstrukt), welche am Anfang des Anweisungsblocks einer Prozedur stehen muss, wurde manuell vor den eingefügten Capturecode verschoben.

11.1.5 Diskussion

Die Erprobung des FortranTestGenerators mit vier verschiedenen Klimamodellen hat gezeigt, dass das Werkzeug in der Lage ist, funktionierende Tests für Prozeduren realer Anwendungen zu erzeugen. Nur in wenigen Fällen lieferten die isolierten Prozeduren für einzelne Variablen nicht dieselben Werte wie bei der Ausführung innerhalb der Originalanwendung. Insbesondere die Versuche mit sehr hochrangigen Prozeduren mit hunderten bis zu über eintausend Unterprozeduren, mehreren hunderttausend zu analysierenden Codezeilen und mehreren tausend Eingabevariablen, wie etwa `intergrate_nh` (ICON) oder `calculate_diagnostic_fields` (MOM6) demonstrieren die allgemeine Funktionsfähigkeit des Werkzeugs. Die Anzahl der Unterprozeduren und die zu analysierenden Codezeilen sind neben weiteren Metriken in Tabelle 11.7 angegeben.

Für fast alle Prozeduren waren kleinere bis größere Anpassungen des Originalquellcodes, des Templates und/oder des generierten Codes notwendig, um zu einem funktionierenden Test zu kommen. Einige notwendige Anpassungen, wie etwa das Entfernen individueller `PRIVATE`-Deklarationen, waren sehr einfach zu ermitteln und umzusetzen. Andere waren mit erheblichem Aufwand verbunden. So etwa das Erstellen und Einbinden des JSBACH-Testdoubles (ICON), das Wiederherstellen der MPI-Kommunikatoren oder das Einbetten modulloser Subroutinen in Module (jeweils CESM2).

Einige Anpassungen waren notwendig aufgrund von Unzulänglichkeiten des FortranTestGenerators, die nicht grundsätzlicher Natur sind, sondern im Entwicklungsstadium des Werkzeugs begründet sind, wie etwa die Nichtunterstützung von Subroutinen außerhalb von Modulen. Andere Anpassungen waren aufgrund der grundsätzlichen Funktionsweise des Werkzeugs notwendig, wie etwa die Umwandlung von `SAVE`-Variablen in Modulvariablen. Weitere Anpassungen standen zudem im Zusammenhang mit grundsätzlich problematischen bzw. fehlerhaften Konstrukten im Originalquellcode, die im Zuge der Testerstellung aufgedeckt wurden, wie z.B. nicht initialisierte Variablen oder hängende Zeiger.

Lediglich das Problem der individuellen `PRIVATE`- und `PROTECTED`-Deklarationen trat bei allen Modellen auf. Alle weiteren für den FortranTestGenerator problematischen Konstrukte mussten nur vereinzelt behandelt werden (Tabelle 11.6), entweder durch besondere Anpassung des Originalcodes, der Templates oder des generierten Codes.

Konstrukt	Modelle
Individuelle PRIVATE-/PROTECTED-Deklarationen	ICON, MOM6, CESM2, NICAM
Nicht initialisierte oder hängende Zeiger	ICON
Vererbung	ICON
C-Bibliotheken	ICON
MPI-Kommunikator-Handles	ICON, CESM2
PROZEDURARGUMENTE	ICON
Nicht auffindbare Module (wg. Präprozessorfallunterscheidungen)	ICON
Namenskollisionen aufgrund vollständigen Modulimports	ICON
Nicht unterstützte FORTRAN-77-Syntax	MOM6, CESM2, NICAM
Nicht erkannte Import-Aliase	MOM6
Prozeduren außerhalb von Modulen	CESM2
RETURN-Anweisungen	CESM2, NICAM
Lokale SAVE-Variablen	NICAM
Nicht initialisierte Variablen	NICAM
Stringarrays (von Serialbox2 nicht unterstützt)	NICAM
5-dimensionale Arrays (von Serialbox2 nicht unterstützt)	NICAM

Tabelle 11.6: Auftreten problematischer Konstrukte

11.2 Quantitative Auswertung

Dieser Abschnitt befasst sich mit verschiedenen quantitativen Aspekten der im vorangegangenen Abschnitt beschriebenen Experimente. Um die Tauglichkeit der im FortranTestGenerator implementierten Verfahren im Hinblick auf ihre Skalierbarkeit zu bewerten, wird in Abschnitt 11.2.2 die Dauer der Testerstellung ausgewertet. Die Angemessenheit der benötigten Zeit ergibt sich u.a. aus der Relation zu der generierten Codemenge unter der Annahme, dass bei einer manuellen Testerstellung Code in vergleichbarer Größenordnung geschrieben werden müsste. Die generierten Codemengen werden hierzu eingangs in Abschnitt 11.2.1 erörtert.

Abschnitt 11.2.3 beschäftigt sich mit den Datenmengen, die beim Capture der einzelnen Unittests entstehen. Das Einlesen dieser Daten beim Replay beeinflusst entscheidend die Laufzeiten der Tests. Diese werden in Abschnitt 11.2.4 diskutiert. Um die Nützlichkeit der erstellten Unittests zu bewerten, wird zudem deren Codeabdeckung ermittelt und mit den entsprechenden Werten der E2E-Tests verglichen (Abschnitt 11.2.5).

Die Prozeduren `integrate_nh` (ICON), `calculate_diagnostic_fields` (MOM6), `dyn_run` (CESM2) und `dynstep` (NICAM) mit jeweils mehr als 100.000 normalisierter Codezeilen in allen Unterprozeduren werden bei der Auswertung von Datenmenge, Testlaufzeit und Codeabdeckung nicht berücksichtigt. Diese dienten vorangehend zur Demonstration der Funktionsfähigkeit des FortranTestGenerators. Tatsächliche Unittests für derartig hochrangige Prozeduren ohne einen umfangreichen Einsatz von Testdoubles erscheinen jedoch nicht sinnvoll. Daher hätten die für diese Prozeduren ermittelten Zahlen keine nennenswerte Aussagekraft.

Prozedur	Originalcode [Anzahl...] Unterprozeduren	Originalcode [Anzahl...] Normalisierte Codezeilen	Originalcode [Anzahl...] Modulabhängigkeiten	Originalcode [Anzahl...] Eingabevariablen	Generierter Code [Anzahl Zeilen] Capture	Generierter Code [Anzahl Zeilen] Export	Generierter Code [Anzahl Zeilen] Replay	Generierter Code [Anzahl Zeilen] gesamt	Generierter Code [Anzahl Zeilen] pro Variable	fcg/FTG-Laufzeiten [Sekunden] Aufrufgraph erzeugen	fcg/FTG-Laufzeiten [Sekunden] INTERFACES und Typen suchen	fcg/FTG-Laufzeiten [Sekunden] Verwendete Variablen ermitteln	fcg/FTG-Laufzeiten [Sekunden] Code generieren	fcg/FTG-Laufzeiten [Sekunden] gesamt
ICON (PC)														
`compute_airmass`	2	82	89	15	123	0	131	254	16,93	0,0	2,2	0,1	0,2	2,5
`integrate_nh`	1.165	372.800	569	4.214	12.683	149	16.499	29.331	6,96	85,3	16,1	161,8	43,8	307,1
`solve_nh`	91	90.636	132	424	1.366	8	1.343	2.717	6,41	4,6	3,5	8,1	2,7	19,0
`step_advection`	101	26.765	97	278	908	3	844	1.755	6,31	5,1	2,6	16,1	1,9	25,7
`update_diag_state`	80	1.600	91	308	1.003	7	980	1.990	6,46	3,4	2,1	1,5	1,8	8,8
`update_ho_params`	70	1.820	115	275	971	12	787	1.770	6,44	3,9	3,2	5,5	2,1	14,6
`update_surface`	31	12.059	147	136	489	3	613	1.105	8,13	0,9	3,6	0,6	1,2	6,2
ICON (Mistral)														
`compute_airmass`	2	82	89	15	123	0	131	254	16,93	0,2	3,8	0,1	1,2	5,3
`integrate_nh`	1.074	343.680	569	3.906	11.642	144	15.326	27.112	6,94	73,5	27,4	288,5	109,7	499,1
`solve_nh`	91	90.636	132	424	1.366	8	1.343	2.717	6,41	7,4	5,2	10,6	8,7	31,9
`step_advection`	101	26.765	97	278	908	3	844	1.755	6,31	8,5	4,2	24,6	8,2	45,4
`update_diag_state`	80	1.600	91	308	1.003	7	980	1.990	6,46	5,7	3,5	2,2	6,9	18,4
`update_ho_params`	70	1.820	115	275	971	12	787	1.770	6,44	6,5	4,9	7,8	6,1	25,3
`update_surface`	31	12.059	147	136	489	3	613	1.105	8,13	1,8	5,7	0,9	5,9	14,2
MOM6 (PC)														
`calculate_diagnostic_fields`	372	169.632	93	2.389	9.517	42	13.390	22.949	9,61	80,1	6,7	90,2	14,8	191,8
`int_density_dz_linear`	1	131	34	41	148	0	182	330	8,05	0,0	4,4	0,0	0,2	4,7
CESM2 (PC)														
`d2a3dikj`	31	6.541	21	82	220	9	306	535	6,52	0,8	0,8	1,1	0,7	3,5
`dyn_run`	91	152.243	374	434	1.242	15	1.722	2.979	6,86	3,0	11,5	4,0	2,7	21,1
`mapn_ppm_tracer`	4	292	21	15	118	0	167	285	19,00	0,1	0,8	0,0	0,2	1,1
NICAM (Mistral)														
`dynstep`	87	107.706	43	502	1.694	42	2.195	3.931	7,83	19,7	3,7	4,6	8,2	36,2
`mp_lsc_wcdiag`	1	34	13	12	87	1	129	217	18,08	0,2	0,9	0,1	1,2	2,3
`vi_rhow_update_matrix`	1	103	25	47	187	3	217	407	8,66	0,2	2,6	0,1	1,7	4,5
`vi_small_step`	26	15.314	25	244	809	15	1.048	1.872	7,67	5,9	2,2	1,0	4,9	14,1

Tabelle 11.7: Größenmetriken und Werkzeuglaufzeiten für alle getesteten Prozeduren
Die Größenmetriken für `integrate_nh` auf PC und Mistral unterscheiden sich aufgrund unterschiedlicher Präprozessoreinstellungen.

11.2.1 Codemengen

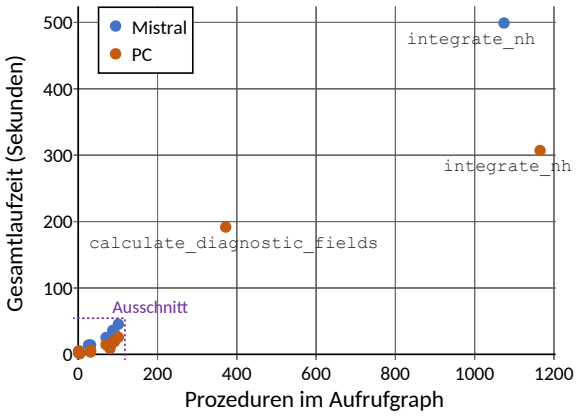
Tabelle 11.7 zeigt u.a. die Mengen des vom FortranTestGenertor generierten Codes. Die Zahlen stellen die Anzahl der generierten Codezeilen *(lines of code, LOC)* ohne Leerzeilen und Kommentare dar und wurden mit dem Softwarewerkzeug *cloc* (GitHub, cloc) ermittelt. Für Capture- und Exportcode wurden sie bestimmt, in dem die Differenz der Codezeilenanzahl vor und nach der Instrumentierung gebildet wurde. Bei den meisten Prozeduren liegt die Anzahl der generierten Codezeilen bei etwa 6–9 Zeilen pro ermittelter Eingabevariable (ohne Leerzeilen und Kommentare). Die Prozeduren mit sehr wenigen Eingabevariablen fallen heraus, da hier der Anteil der fixen gegenüber den variablen Codezeilen größer ist. Auffällig ist zudem die Prozedur `calculate_diagnostic_fields` (MOM6) mit 9,61 LOC/Eingabevariable bei 2.389 Eingabevariablen. Schuld sind hier jeweils komplexe Typstrukturen, für deren Verarbeitung mehr Code notwendig ist.

11.2.2 Werkzeuglaufzeit

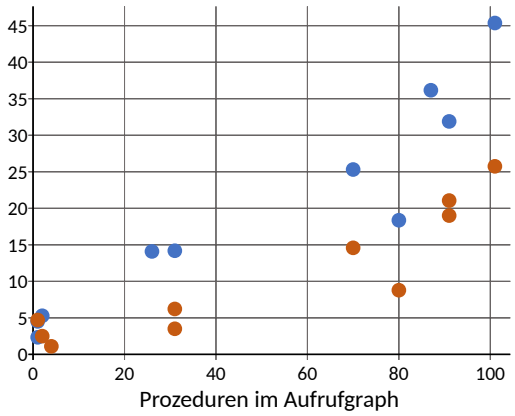
Tabelle 11.7 sowie die Abbildung 11.3 zeigen den zeitlichen Aufwand des FortranTestGenerators. Die Zahlen sollen einen ungefähren Eindruck der Größenordnung des Aufwands vermitteln und sind nicht allgemeingültig. Sie wurden einmalig auf den jeweiligen Plattformen ermittelt, indem FTG für die jeweilige Prozedur zehnmal hintereinander ausgeführt wurde und der Durchschnitt der dabei gemessenen Zeiten ermittelt wurde. Die Laufzeiten sind stark von der Rechner- und I/O-Plattform und deren momentanen Auslastung abhängig. Letzteres gilt insbesondere für Mehrbenutzersysteme. So wurde auf der Mistral FTG auf einem Login-Knoten ausgeführt, der i.d.R. gleichzeitig von mehreren Benutzern verwendet wird. So sind hier die ermittelten Laufzeiten auch höher als auf dem PC.

Intuitiv betrachtet liegen die Laufzeiten durchgängig im akzeptablen Bereich. Lediglich für die hochrangigen Prozeduren `integrate_nh` und `calculate_diagnostic_fields` waren mehrere Minuten nötig, um Analyse, Instrumentierung und Testerstellung durchzuführen. Berücksichtigt man, dass die zu testenden Prozeduren im realen Einsatz normalerweise nicht derartig hochrangig sein dürften, der FortranTestGenerator pro zu testender Prozedur in der Regel nur einmal ausgeführt wird und dass dabei Mengen an Code generiert werden (siehe Tabelle 11.7), deren manuelle Erstellung mehrere Stunden oder Tage dauern dürfte, erscheinen auch diese Laufzeiten von bis zu 8,5 Minuten für ein derartiges Werkzeug unproblematisch.

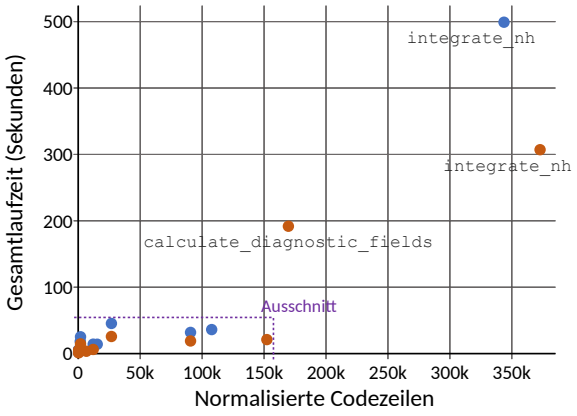
Die FTG-Gesamtlaufzeit wird in den Abbildungen 11.2a und 11.2b mit der Anzahl der Unterprozeduren in Beziehung gesetzt und in den Abbildungen 11.2c und 11.2d mit der Anzahl der normalisierten Codezeilen in allen Unterprozeduren. Dabei zeigen die Abbildungen 11.2a und 11.2c jeweils die Werte aller getesteter Prozeduren,



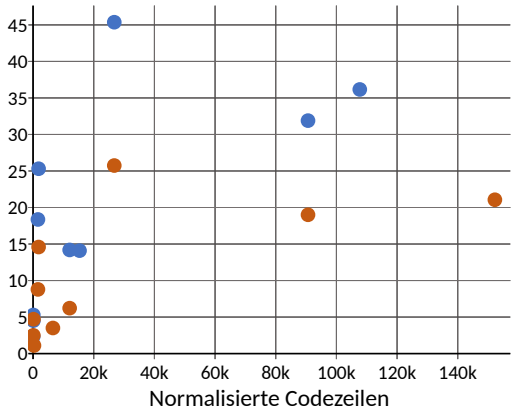
(a) alle Versuche



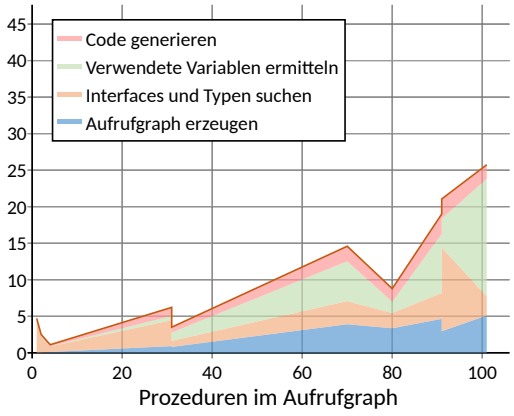
(b) Ausschnitt



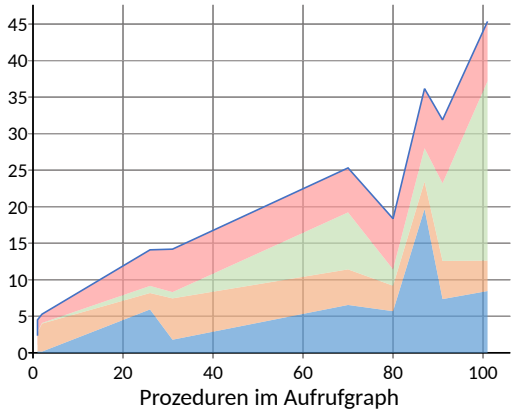
(c) alle Versuche



(d) Ausschnitt



(e) PC: Auschnitt



(f) Mistral: Ausschnitt

Abbildung 11.2: Laufzeiten FortranTestGenerator

während die Abbildungen 11.2b und 11.2d jeweils die in 11.2a und 11.2b markierten Ausschnitte darstellen. Dabei ist eher ein Zusammenhang zwischen der Anzahl der Unterprozeduren und der Gesamtlaufzeit zu erkennen als zwischen der Anzahl der Codezeilen und der Gesamtlaufzeit. Aufgrund der starken Streuung und der kleinen Stichprobe lassen sich valide statistische Aussagen jedoch nicht treffen.

Die Abbildungen 11.2e und 11.2f zeigen getrennt für PC und Mistral die Anteile der einzelnen Analyseschritte und der Codegenerierung am zeitlichen Gesamtaufwand des FortranTestGenerators. Dargestellt ist jeweils der in Abbildung 11.2a markierte Ausschnitt. Zu erkennen ist, dass zwischen den einzelnen Analyseschritten (Aufrufgraph erzeugen, INTERFACES und Typen suchen, verwendete Variablen ermitteln) keine eindeutige Verteilung existiert. Je nach zu testender Prozedur überwiegen unterschiedliche Analyseschritte. Lediglich der Anteil der Codegenerierung ist einigermaßen einheitlich, wobei deutlich wird, dass insbesondere hierbei der PC im Vorteil ist, was an dem lokalen und exklusiven Zugriff auf das Dateisystem liegen dürfte. Hierbei dürfte auch das Vorhandensein eines *Flashspeicher-Laufwerks (solid state drive)* im Testrechner von Vorteil sein (siehe auch Anhang E.1.1).

11.2.3 Datenmengen

Ein wesentliches Merkmal der auf Capture & Replay basierenden Unittests ist, dass sämtliche Eingabedaten der zu testenden Prozedur aufgezeichnet und bei jedem Testlauf geladen werden müssen. Hinzu kommen ggf. Ausgabedaten zur Validierung. Da das Einlesen von Daten in der Regel ein teurer Vorgang in Bezug auf den zeitlichen Aufwand ist, hat der Umfang dieser Daten entscheidenden Einfluss auf die Laufzeit der Tests. Der Umfang der bei Verwendung des FortranTestGenerators erzeugten Daten hängt wiederum von drei Faktoren ab: die Anzahl und Größe der Ein- und Ausgabevariablen, die Anzahl der beim Testen verwendeten Prozesse sowie die verwendete I/O-Technologie. Die in den FTG-Standardtemplates verwendete I/O-Bibliothek Serialbox2 arbeitet nach sehr einfachen Prinzipien: für jede Variable wird eine eigene Datei angelegt, redundante Daten werden nicht zusammengefasst, es findet keine Kompression statt.

Tabelle 11.8 zeigt die Datenmengen, die im Rahmen der in Abschnitt 11.1 beschriebenen Experimenten aufgezeichnet wurden. Die mit 236 KB kleinste Gesamtdatenmenge benötigt der für einen Prozess erzeugte Test für die SUBROUTINE `int_density_dz_linear` (MOM6). Der größte Datensatz mit 2,4 GB wurde auf 40 Prozessen für `vi_small_step` (NICAM) aufgezeichnet. Für die Datenmenge pro Prozess ergibt sich eine Bandbreite von 236 KB bis zu 557 MB.

Die Menge der Eingabedaten pro Prozess und Eingabevariable liegt zwischen 3 KB und 2,5 MB. Beim letzteren Wert (`mapn_ppm_tracer`) handelt es sich jedoch um

Prozedur	Unterprozeduren	Normalisierte Codezeilen	Eingabevariablen	Eingabedaten	Eingabedaten pro Prozess	Eingabedaten pro Prozess und Variable	Ausgabedaten	Ausgabedaten pro Prozess	gesamt	gesamt pro Prozess
ICON (PC, 2 Prozesse)										
compute_airmass	2	82	15	30,6 MB	15,3 MB	1,0 MB	15,3 MB	7,6 MB	45,9 MB	22,9 MB
solve_nh	91	90.636	424	576 MB	288 MB	695 KB	527 MB	263 MB	1103 MB	551 MB
step_advection	101	26.765	278	278 MB	139 MB	511 KB	159 MB	79,3 MB	436 MB	218 MB
update_diag_state	80	1.600	308	171 MB	85,5 MB	284 KB	144 MB	71,8 MB	315 MB	157 MB
update_ho_params	70	1.820	275	187 MB	93,7 MB	349 KB	140 MB	69,8 MB	327 MB	163 MB
update_surface	31	12.059	136	722 KB	361 KB	3 KB	519 KB	260 KB	1,2 MB	621 KB
ICON (Mistral, 24 Prozesse)										
compute_airmass	2	82	15	37,0 MB	1,5 MB	105 KB	18,4 MB	784 KB	55,3 MB	2,3 MB
solve_nh	91	90.636	424	702 MB	29,3 MB	71 KB	642 MB	26,8 MB	1344 MB	56,0 MB
step_advection	101	26.765	278	344 MB	14,3 MB	53 KB	201 MB	8,4 MB	545 MB	22,7 MB
update_diag_state	80	1.600	308	213 MB	8,9 MB	29 KB	179 MB	7,5 MB	392 MB	16,3 MB
update_ho_params	70	1.820	275	235 MB	9,8 MB	36 KB	172 MB	7,2 MB	407 MB	16,9 MB
update_surface	31	12.059	136	10,8 MB	460 KB	3 KB	8,3 MB	354 KB	19,1 MB	813 KB
MOM6 (PC, 1 Prozess)										
int_density_dz_linear	1	131	41	177 KB	177 KB	4 KB	59 KB	59 KB	236 KB	236 KB
CESM2 (PC, 2 Prozesse)										
d2a3dikj	31	6.541	82	54,1 MB	27,0 MB	338 KB	28,7 MB	14,4 MB	82,8 MB	41,4 MB
mapn_ppm_tracer	4	292	15	76,3 MB	38,1 MB	2,5 MB	75,9 MB	38,0 MB	152 MB	76,1 MB
NICAM (Mistral, 40 Prozesse)										
mp_lsc_wcdiag	1	34	12	21,9 MB	561 KB	47 KB	14,7 MB	377 KB	36,6 MB	938 KB
vi_rhow_update_matrix	1	103	47	51,2 MB	1,3 MB	28 KB	92,7 MB	2,3 MB	144 MB	3,6 MB
vi_small_step	26	15.314	244	1,2 GB	30,7 MB	129 KB	1,2 GB	29,5 MB	2,4 GB	60,2 MB

Tabelle 11.8: Aufgezeichnete Datenmengen

Die Zahlen wurden mit Hilfe des Linuxkommandos `du --apparent-size -c` ermittelt.
Die Datenmengen für ICON mit zwei Prozessen auf der Mistral entsprechen in etwa denen auf dem PC.

einen einzelnen Ausreißer. Alle anderen Werte liegen bei max. 1,0 MB und der Mittelwert bei 352 KB.

Der Vergleich zwischen ICON mit 2 und 24 Prozessen zeigt, dass die Datenmenge nicht unbedingt proportional mit der Prozessanzahl wächst. Stattdessen liegen bei fünf von sechs Prozeduren die Gesamtdatenmengen bei beiden Versuchsreihen in ähnlichen Größenordnungen. Dies liegt daran, dass aufgrund der Domain Decomposition die Modelldaten zwischen den Prozessen aufgeteilt werden und bei mehr Prozessen jeder Prozess entsprechend kleinere Arrays verwaltet. Eine Ausnahme bildet der Test für die SUBROUTINE `update_surface`. Dies zeigt, dass dieser Zusammenhang nicht in allen Fällen zutrifft.

Anders als die Datenmenge erhöht sich aufgrund der Funktionsweise von Serialbox2 jedoch die Anzahl der Dateien proportional mit der Anzahl der Prozesse und damit die Zugriffe auf das Dateisystem beim Einlesen der Daten. Da in der Regel alle Prozesse auf das gleiche Dateisystem zugreifen ergibt sich auch hieraus ein erhöhter Aufwand.

11.2.4 Testlaufzeit

Ein wesentliches Ziel dieser Arbeit ist es, die Erstellung von Unittests zu unterstützen, die signifant schneller sind als die alternativen E2E-Tests. Mit „signifikant schneller“ ist gemeint, dass die Ausführung eines Unittests weniger als ein Zehntel der Zeit benötigt im Vergleich zur Ausführung des E2E-Tests, von dem er abgeleitet wurde (siehe auch Abschnitt 6.2). Da die Unittests, die im Rahmen der in Abschnitt 11.1 beschriebenen Experimente erzeugt wurden, jeweils nur einen einzigen Testfall enthalten, d.h. die zu testenden Prozedur nur einmal ausführen, ist als Vergleichsgröße der jeweilige E2E-Tests beschränkt auf einen einzigen Zeitschritt heranzuziehen, da dies in den meisten Fällen ausreichen dürfte, um die zu testenden Prozedur einmal auszuführen.

Tabelle 11.9 zeigt die gemessenen Laufzeiten der generierten Unittests sowie die Laufzeiten der E2E-Tests, aus denen sie jeweils abgeleitet wurden. Für die E2E-Tests sind jeweils die Zeiten für die vollständige Ausführung angegeben sowie die Zeiten, die sich durch Reduzierung auf einen Zeitschritt ergeben. Die Simulationsdauern der vollen Ausführungen sind jeweils die, die in den jeweiligen Konfigurationen, entnommen aus den Testsammlungen der Modelle, vorgegeben waren. Zu berücksichtigen ist auch hier, dass die ermittelten Laufzeiten keine exakten Werte darstellen, da diese stark von der Auslastung des Rechners und des I/O-Systems abhängen. Die letzten zwei Spalten der Tabelle zeigen die Geschwindigkeitsfaktoren, die sich zwischen Unittest

Prozedur	E2E-Test vollständig	E2E-Test 1 Zeitschritt	Unittest	$\frac{\text{E2E-Test (vollständig)}}{\text{Unittest}}$	$\frac{\text{E2E-Test (1 Zeitschritt)}}{\text{Unittest}}$
ICON (PC, 2 Prozesse)					
exp.atm_amip_noforcing_notransport_test	47,7 s	22,8 s			
compute_airmass			90 ms	540	260
solve_nh			3,8 s	13	6,0
exp.atm_amip_test	5,1 min	2,4 min			
step_advection			2,9 s	100	49
update_surface			80 ms	3600	1700
exp.atm_jww_hs_test	15,7 s	5,9 s			
update_diag_state			400 ms	39	15
exp.test_ocean_omip_10days	9,2 min	8,6 s			
update_ho_params			540 ms	1000	16
ICON (Mistral, 2 Prozesse)					
exp.atm_amip_noforcing_notransport_test	1,0 min	34,1 s			
compute_airmass			2,2 s	29	16
solve_nh			8,5 s	7,3	4,0
exp.atm_amip_test	5,9 min	2,8 min			
step_advection			6,7 s	53	26
update_surface			2,4 s	150	72
exp.atm_jww_hs_test	25,6 s	13,5 s			
update_diag_state			3,6 s	7,1	3,8
exp.test_ocean_omip_10days	10,4 min	15,6 s			
update_ho_params			11,3 s	55,0	1,4
ICON (Mistral, 24 Prozesse)					
exp.atm_amip_noforcing_notransport_test	28,7 s	24,1 s			
compute_airmass			5,8 s	4,9	4,1
solve_nh			12,2 s	2,3	2,0
exp.atm_amip_test	1,1 min	46,4 s			
step_advection			10,6 s	6,4	4,4
update_surface			7,7 s	8,8	6,1
exp.atm_jww_hs_test	22,6 s	19,9 s			
update_diag_state			9,7 s	2,3	2,0
exp.test_ocean_omip_10days	1,5 min	22,3 s			
update_ho_params			17,9 s	5,0	1,2
MOM6 (PC, 1 Prozess)					
external_gwave	3,8 s	1,3 s			
int_density_dz_linear			20 ms	160	53
CESM2 (PC, 2 Prozesse)					
fkessler	2,0 h	1,7 min			
d2a3dikj			120 ms	62000	910
mapn_ppm_tracer			180 ms	42000	600
NICAM (Mistral, 40 Prozesse)					
GL05RL01Az78	2,3 min	10,4 s			
mp_lsc_wcdiag			2,5 s	54	4,2
vi_rhow_update_matrix			6,0 s	23	1,7
vi_small_step			15,0 s	9,0	0,7

Tabelle 11.9: Testlaufzeiten

Die Stunden-, Minuten- und Sekundenangaben wurden jeweils auf eine Nachkommastelle gerundet, Millisekundenangaben auf 10 Millisekunden. Die Faktoren in den beiden rechten Spalten wurden nach Division der ursprünglich gemessenen Millisekunden auf jeweils zwei signifikante Stellen gerundet.

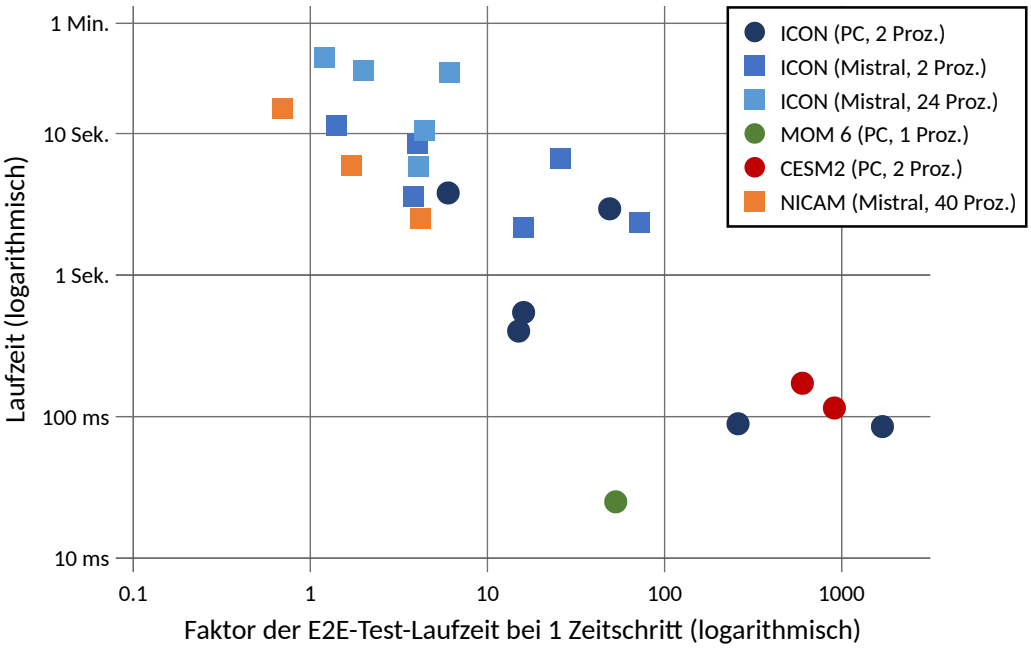
und vollständigem E2E-Test bzw. zwischen Unittest und E2E-Test mit einem Zeitschritt ergibt. Der jeweilige Faktor ergibt sich aus dem Quotienten aus der Laufzeit des E2E-Tests und der Laufzeit des Unittests.

Ermittelt wurden die Zahlen, indem die Tests jeweils mehrfach hintereinander ausgeführt wurden und aus den dabei gemessenen Zeiten der Durchschnitt gebildet wurde. Bei der Anzahl der Wiederholungen wurde an die erwartete Gesamtdauer angepasst. So wurden die E2E-Tests jeweils zehnmal wiederholt, die Unittests auf dem PC einhundertmal und die Unittests auf der Mistral dreißigmal.

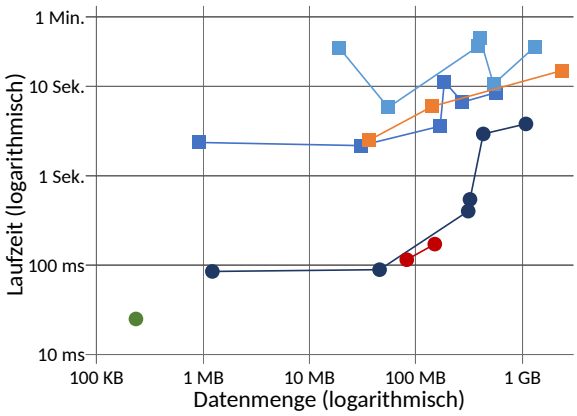
Abbildung 11.3a veranschaulicht die Verteilung von Testlaufzeiten und Geschwindigkeitsfaktoren zu den einschrittigen E2E-Tests. Zu sehen ist, dass nur ein Teil der Tests das Ziel Faktor 10 erreicht. In einem Fall ist der Unittest sogar langsamer als der entsprechende E2E-Test mit einem Zeitschritt. Zudem wird deutlich, dass insbesondere auf der Mistral der Geschwindigkeitsgewinn gering ist, während auf dem PC bis zu vierstellige Faktoren und Testlaufzeiten von zum Teil deutlich unter einer Sekunde erreicht werden. Nur in einem Fall verfehlt der Unittest auf dem PC die Faktor-10-Marke.

Ein weiterer erkennbarer Zusammenhang besteht darin, dass je höher die Anzahl der Prozesse ist, desto langsamer sind die Unittests und desto geringer fällt der Geschwindigkeitsvorteil aus. Dies ist mit der wachsenden Datenmenge zu erklären, die mit steigender Prozessanzahl eingelesen werden muss. Wie an den Ausreißern in Abbildung 11.3b zu sehen ist, ist die Testlaufzeit jedoch nicht allein von der einzulesenden Datenmenge abhängig. Auch aus der Anzahl der Variablen, die auch die Anzahl der einzulesenden Dateien bestimmt (Abbildung 11.3c), sowie aus der Gesamtzahl der Codezeilen im Aufrufgraphen der zu testenden Prozedur (Abbildung 11.3d) ergibt sich kein eindeutiges Verhältnis zur Testlaufzeit. Daraus lässt sich schließen, dass es die Kombination mehrerer Faktoren ist, die die Laufzeit der Tests bestimmt. Auch die algorithmische Komplexität, die sich aus den erhobenen Daten nicht herauslesen lässt, spielt voraussichtlich eine Rolle, ebenso wie die Codeabdeckung der Tests. Letztere wird im kommenden Abschnitt diskutiert. Aus dem deutlichen Vorteil des PCs sowie geringerer Prozesszahlen lässt sich jedoch schließen, dass der Aufwand für das Einlesen der Daten eine entscheidende Rolle spielt.

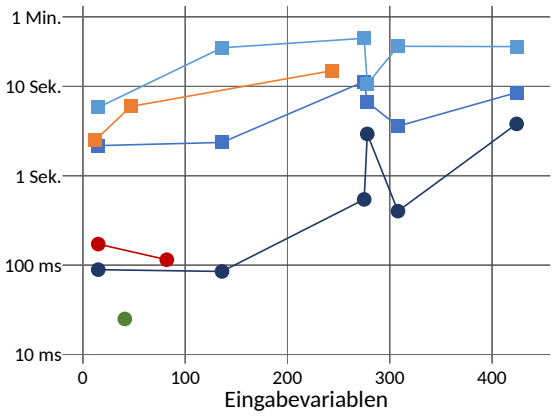
Der Geschwindigkeitsvorteil auf der Mistral relativiert sich weiter, berücksichtigt man die Zeit, die man auf die Ausführung des Tests warten muss. Während man auf dem PC interaktiv Programme ohne Verzögerung starten kann, muss man auf Hochleistungsrechnern wie der Mistral in der Regel warten bis einem der Scheduler die benötigten Ressourcen zuweist. Bei den durchgeführten Messungen betrugen diese Wartezeiten im Mittel ca. 20-30 Sekunden. Bei Tests mit Laufzeiten im Sekundenbereich stellen diese Wartezeiten den größten Anteil der Zeit, die man auf ein Testergebnis warten muss.



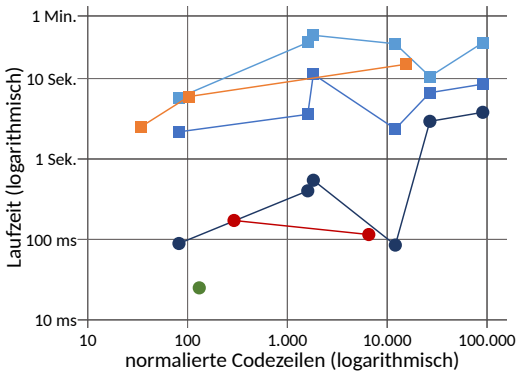
(a)



(b)



(c)



(d)

Abbildung 11.3: Unittestlaufzeiten

Bei der Bewertung des Geschwindigkeitsvorteil der Unittests gegenüber den entsprechenden E2E-Tests muss berücksichtigt werden, dass die generierten Unittests bereits eine umfangreiche Validierung der Ergebnisse der zu testenden Prozedur enthalten. Da hierbei eine Vielzahl von Variablen verglichen wird, stellt diese einen wesentlichen Teil des zeitlichen Aufwands dar. In den auf einen Zeitschritt reduzierten E2E-Tests sind derartige Überprüfungen nicht enthalten.

Nicht ermittelt wurde der Geschwindigkeitsvorteil, der sich beim Kompilieren der Unittests gegenüber dem vollen Modell ergibt. Hierzu hätten die komplexen Buildskripte der Modelle angepasst werden müssen, um einen reduzierten Build zu ermöglichen. Dies war im Rahmen der vorgenommenen Experimente leider nicht möglich. Stehen entsprechend angepasste Buildskripte zur Verfügung, die es erlauben, die zu testende Prozedur ohne die nicht benötigten Teile des Modells zu kompilieren, liegt der Geschwindigkeitsvorteil jedoch auf der Hand. Je niederrangig die Prozedur desto größer dieser Vorteil. Das Kompilieren eines gesamten Modells kann einige Minuten dauern, daher könnten sich hier größere Geschwindigkeitsvorteile als bei der Laufzeit ergeben.

11.2.5 Codeabdeckung

Entscheidend für die Tauglichkeit des FortranTestGenerators als Werkzeug zur Testerzeugung ist die Nützlichkeit der generierten Unittests. Um die Nützlichkeit der im Rahmen der hier beschriebenen Experimente erzeugten Unittests zu bewerten, wurde deren Codeabdeckung ermittelt. Zur Anwendung kommen dabei drei Abdeckungsmetriken: die Prozedurabdeckung, die Zeilenabdeckung und die Zweigabdeckung (siehe auch Abschnitt 3.7). Ermittelt wurden diese mit den Softwarewerkzeugen *gcov* (GNU, gcov) und *LCOV* (GitHub, LCOV).

Abbildung 11.4 zeigt die Codeabdeckung von E2E- und Unittests für alle in diesem Abschnitt betrachteten Prozeduren. Bemessungsgrundlage ist jeweils der gesamte Code innerhalb aller Unterprozeduren der jeweiligen PUT. Alle von der PUT nicht direkt oder indirekt aufgerufenen Prozeduren gehen nicht in die Berechnung ein. Die dunkleren Balken stellen jeweils das Ergebnis für den Unittest, die helleren, „hinteren“ Balken die Ergebnisse der E2E-Tests. Dort wo die helleren vollständig verdeckt sind, erreicht der Unittest jeweils die gleiche Codeabdeckung wie der E2E-Test aus dem er abgeleitet wurde. Die Zahlen unterhalb der Balken geben jeweils das Verhältnis zwischen Unittest- und E2E-Test-Abdeckung an.

Für die Messung der Codeabdeckung der E2E-Tests wurden diese vollständig, d.h. mit der vorgegebenen Simulationsdauer ausgeführt.

Die Prozeduren sind in der Grafik aufsteigend nach Anzahl der Unterprozeduren sortiert. Wie man sieht ist die Abdeckung bei den niederrangigeren Prozeduren im

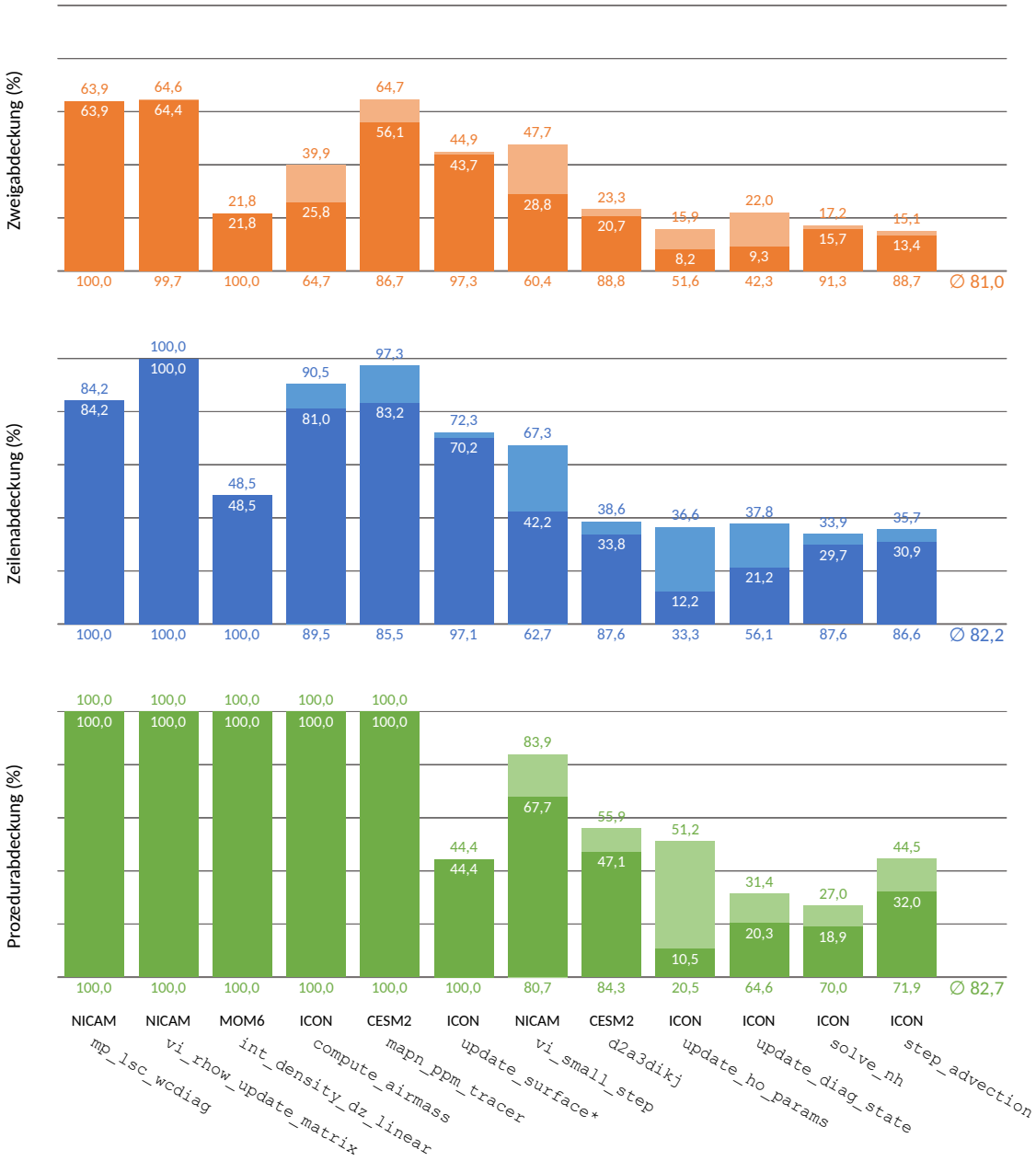


Abbildung 11.4: Codeabdeckung (alle Unterprozeduren)
* Bemessungsgrundlage für `update_surface` ohne JSBACH-Code (siehe auch Abschnitt 11.1.1)

Mittel höher als bei den höherrangigen. Eine Ausnahme bildet hier die niederrangige Prozedur `int_density_dz_linear`, bei der sowohl Unit- als auch E2E-Test nur auf eine Zeilenabdeckung von 48,5 % und eine Pfadabdeckung von 21,8 % kommen. Zudem schneidet insbesondere der Unittest von `compute_airmass` bei der Pfadabdeckung nicht so gut ab.

Die hier generierten Unittests enthalten genau einen Datensatz, der aus dem jeweiligen E2E-Test extrahiert wurde, d.h. sie replizieren eine Prozedurausführung aus dem E2E-Test. Daher kann ihre Abdeckung nicht höher sein als die des E2E-Tests. Dies ist eine grundsätzliche Beschränkung des Capture-&-Replay-Ansatzes. Um eine höhere Abdeckung zu erreichen, müssten weitere Testfälle extrahiert werden, ggf. aus anderen E2E-Tests, oder aus den vorhandenen Daten abgeleitet werden (siehe auch Abschnitt 8.2.3).

Berücksichtigt man die Tatsache, dass die Prozeduren im Unittest jeweils nur einmal ausgeführt werden, während die E2E-Tests mehrere Zeitschritte mit mehreren Prozedurausführungen durchlaufen und dabei auch weiterer Code ausgeführt wird, erscheinen die Abdeckungswerte, die die Unittests im Vergleich zu den E2E-Tests erreichen sehr gut. Einzig beim Unittest zu `update_ho_params` liegt die Rate bei Prozedur- und Zeilenabdeckung nicht über 50 %. Bei der Zweigabdeckung betrifft dies nur den Test von `update_diag_state`. Die in der Grafik angegebenen Mittelwerte stellen jeweils den Durchschnitt der zwölf Einzelwerte da – ungewichtet. Für alle drei Abdeckungsmetriken liegt dieses Mittel bei über 80 %.

Da die Bemessungsgrundlage der Code aller Unterprozeduren der jeweiligen PUT ist, ist zu berücksichtigen, dass einzelne dieser Prozeduren während der E2E-Tests auch unabhängig von der PUT aufgerufen werden können. Ein Teil der Abweichungen zwischen E2E- und Unittestabdeckung dürfte auf solche unabhängigen Unterprozeduraufrufe zurückzuführen sein. Insbesondere bei `update_ho_params`, wo die Prozedurabdeckung des E2E-Tests deutlich höher liegt als die des Unittests dürfte dies der Fall sein.

Betrachtet man nur die Codeabdeckung innerhalb der PUT und lässt den Code der Unterprozeduren unberücksichtigt (Abbildung 11.5), gleicht sich die Abdeckung der Unittests noch stärker der E2E-Abdeckung an. Lediglich die Unittests von zwei Prozeduren erreichen hier nicht die gleiche Zeilen- und Zweigabdeckung[21] wie die E2E-Tests aus denen sie abgeleitet wurden. Dies bedeutet, dass in dem meisten Fällen ein einziger aufgezeichneter Testdatensatz ausreicht, um dieselben Zeilen bzw. Zweige innerhalb einer PUT auszuführen wie ein vollständiger E2E-Test.

Erklären lässt sich dies damit, dass die überwiegende Zahl der Fallunterscheidungen in Klimamodellen sich auf die aktuelle Modellkonfiguration beziehen (z.B. soll

[21] Die Prozedurabdeckung ist in Abbildung 11.5 nicht dargestellt, da diese bei Beschränkung auf die PUT trivialerweise bei 100 % liegt.

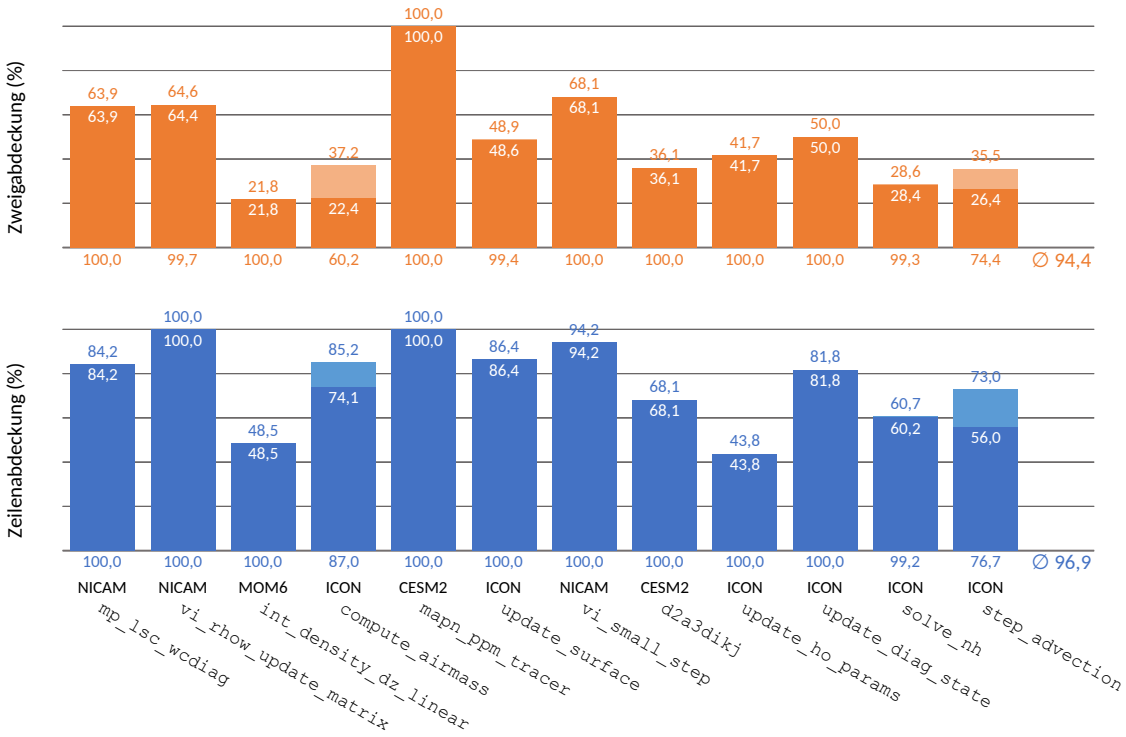


Abbildung 11.5: Codeabdeckung (nur PUT)

diese oder jene Parametrisierung für Größe X verwendet werden). Die Konfigurationsvariablen definieren den jeweiligen Test und ändern sich im Verlauf seiner Ausführung nicht, d.h. bei jeder Ausführung einer Prozedur werden dieselben Zweige dieser konfigurationsbezogenen Fallunterscheidungen ausgeführt. Die für den Unittest ausgewählte Ausführung unterscheidet sich dahingehend nicht von allen anderen Ausführungen. Zwar gibt es auch zeitliche Fallunterscheidungen (z.B. ist jetzt ein Zeitschritt, in dem eine Ausgabe stattfinden soll?) oder arithmetische (z.B. ist Variable x größer als Null?), jedoch deutlich weniger als konfigurationsbezogene.

11.3 Benutzerstudie

Mit den vorangehend beschriebenen Experimenten wurde die grundsätzliche Funktionsfähigkeit des FortranTestGenerators unter Beweis gestellt. Zudem wurde gezeigt, dass die einzelne auf Capture & Replay basierende Unittests Codeabdeckungen erreichen können, die vergleichbar sind mit vollständigen E2E-Tests. Der Geschwindigkeitsvorteil gegenüber auf einen Zeitschritt verkürzte E2E-Tests ist bei Ausführung der Tests auf Hochleistungsrechnern nicht so stark wie erhofft und nimmt mit

steigender Prozessanzahl ab. Auf dem PC ergeben sich jedoch signifikant kürzere Laufzeiten.

Eine Übertragung dieser Erkenntnisse auf den realen Einsatzkontext ist jedoch nur begrenzt möglich. So lässt sich etwa die tatsächliche Nützlichkeit der Tests ohne entsprechendes Fachwissen nicht beurteilen. Zudem folgte die Auswahl der zu testenden Prozeduren sowie der E2E-Tests praktischen anstelle von fachlichen Überlegungen. Auch die Benutzbarkeit des Werkzeugs kann ohne die Einbeziehung echter BenutzerInnen, die nicht gleichzeitig EntwicklerInnen des Werkzeugs sind, nicht beurteilt werden. Um die Tauglichkeit und Benutzbarkeit des FortranTestGenerators unter realen Einsatzbedingungen zu bewerten, wurde daher eine Benutzerstudie mit Entwicklern eines Klimamodells durchgeführt.

11.3.1 Hintergrund

Bei den Teilnehmern der Benutzerstudie handelt es sich um Entwickler des *ENIAC-Projekts (Enabling the ICON Model on Heterogeneous Architectures)*. Das am schweizerischen Hochleistungsrechenzentrum CSCS angesiedelte Projekt beschäftigt sich mit der Portierung von Teilen des ICON-Modells für die Ausführung auf Grafikprozessoren oder anderen sog. *Beschleunigern (accelerators)*. Ziel ist es, ICON effizient auf *Piz Daint*, dem Hochleistungsrechner des CSCS, dessen Rechenleistung sich zu einem großen Teil aus Grafikprozessen speist (vgl. cscs.ch, Piz Daint), ausführen zu können, damit dieses in Zukunft u.a. an der *Eidgenössischen Technischen Hochschule Zürich (ETH)* oder vom schweizerischen Wetterdienst *MeteoSchweiz* zur Klimaforschung und zur Wettervorhersage eingesetzt werden kann (vgl. pasc-ch.org, ENIAC).

Das der Studie zugrundeliegende Einsatzszenario wurde nicht speziell für die Studie entworfen. Stattdessen hatten die Entwickler ein konkretes Problem, zu dessen Lösung sie den FortranTestGenerator eingesetzt haben. Dabei ging es darum, Tests zu erzeugen, mit denen sich die Ergebnisse der portierten Prozeduren bei Ausführung auf GPUs mit den Ergebnissen bei Ausführung auf CPUs vergleichen ließen. Aus den von FTG erzeugten Tests wurden dabei von den Entwicklern sogenannte *Standalones* erzeugt. Dies sind unabhängig kompilierbare Codeeinheiten, die nur die für den Test benötigten Module enthalten. Um diese automatisch zu erstellen und ggf. zu erneuern, wenn es Änderungen am Modell gab, haben die Entwickler Skripte erstellt. In einigen Fällen wurden diese Standalones auch an Rechner- bzw. Compilerhersteller weitergegeben.

Offizieller Start des ENIAC-Projekts war im Juli 2017, jedoch begannen die Beteiligten sich bereits vorher, ab Anfang 2017, mit dem FortranTestGenerator zu beschäftigen. Drei Entwickler arbeiteten etwa bis Ende 2018 mit unterschiedlicher und

wechselnder Intensität mit dem Werkzeug, wobei Konfiguration und Templates größtenteils durch mich bereitgestellt wurden. FTG befand sich zu dieser Zeit noch in der Entwicklung; Fehlerberichte und Anregungen der Entwickler flossen direkt in diese mit ein. Zudem wurde ein Online-Fragebogen erstellt, um die Entwickler nach ihrer allgemeinen Erfahrung mit FTG zu befragen. Dieser wurde zwischen Juli und Oktober 2018 beantwortet.

Bei den Entwicklern bzw. den Studienteilnehmern handelt es sich um einen Studenten, der im Rahmen seiner Masterarbeit an dem Projekt mitwirkte, sowie zwei Wissenschaftlern mit Hintergrund Geophysik und HPC. Befragt nach ihrer Programmiererfahrung gaben diese 6, 12 und 35 Jahre an. Die Fortran-Kenntnisse wurden jeweils einmal mit „mittel“, „fließend“ und „Muttersprache“ angegeben. Somit deckten die drei Teilnehmer jeweils unterschiedliche Erfahrungsstufen ab.

Aufgrund der kleinen Zahl an Teilnehmern hat die Studie keinen repräsentativen Charakter. Zudem hatten die Teilnehmer sowohl dem FortranTestGenerator als auch meiner Person gegenüber grundsätzlich eine positive Einstellung. Beides kann die Ergebnisse möglicherweise beeinflusst haben. Dennoch bieten die Antworten einen ersten grundsätzlichen Eindruck zur Tauglichkeit und Benutzbarkeit des Werkzeugs.

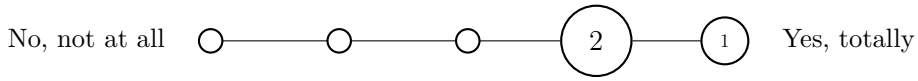
11.3.2 Ergebnisse

Der Hauptteil des Fragebogens bestand aus 17 Fragen zur Tauglichkeit und Benutzbarkeit des FortranTestGenerators, wobei die Einteilung in diese beiden Kategorien auf dem Fragebogen nicht kenntlich gemacht wurde. Zudem wurden die Fragen in einer anderen Reihenfolge gestellt als im Folgenden dargestellt. Zusätzlich wurde gefragt, ob die Teilnehmer auch bereits Erfahrungen mit KGEN (siehe auch Abschnitt 8.5.6) gemacht haben. Neben den hier aufgeführten Fragen wurden zudem weitere zum Hintergrund der Teilnehmer gestellt (dieser wurde im vorangegangenen Abschnitt bereits erörtert) sowie zu einzelnen technischen Aspekten, die für die Auswertung an dieser Stelle nicht relevant sind.

Mit Ausnahme von zwei Ja/Nein-Fragen und den KGEN-Fragen konnten für alle Fragen Antworten auf einer fünfstufigen Skala zwischen zwei gegensätzlichen Extremen angegeben werden, im Fragebogen nummeriert von 0 bis 4. Zudem gab es zu jeder Frage ein Freitextfeld, in dem zusätzliche Bemerkungen notiert werden konnten. Im Folgenden werden die Fragen und Antwortmöglichkeiten im englischen Original aufgeführt. Zudem wird jeweils die Anzahl der einzelnen Antworten angegeben. Die Bemerkungen werden übersetzt und zusammengefasst wiedergegeben.

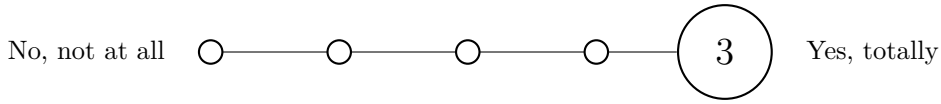
Tauglichkeit

T1 Have you been successfull in using FTG?



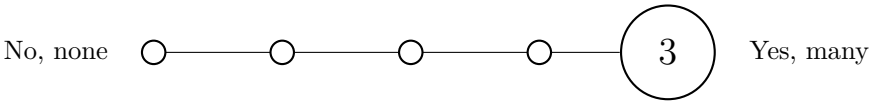
Bemerkungen: *Für einige der Standalones waren umfangreiche manuelle Anpassungen nötig, insbesondere aufgrund abstrakter Datentypen/Polymorphie. / Der Einsatz von FTG war sehr erfolgreich. Zwar entstanden aufgrund von Kinderkrankheiten zahlreiche Probleme, diese wurden vom Entwickler jedoch schnell behoben oder durch „Workarounds“ umgangen.*

T2 Has FTG proven to be useful in your use case(s)?



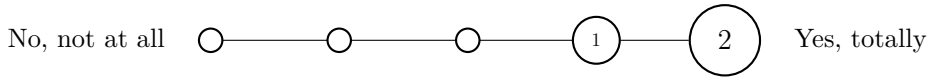
Bemerkung: *Sobald ein Test funktioniert, was zum Teil viel Zeit kostet, ist dieser sehr nützlich. Wir können ihn fast immer mit allen Compilern kompilieren, während eine volle Übersetzung von ICON häufig nicht funktioniert. Zudem ist Validierung und „Benchmarking“ sehr viel einfacher mit den „Standalones“.*

T3 Have you found errors in your code with FTG generated tests?



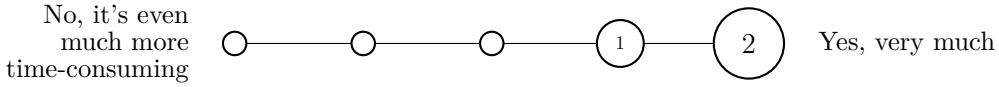
Bemerkung: *Sobald ein Standalone existiert, ist es sehr viel einfacher meine (vielen) Fehler zu finden.*

T4 Do you think, the tests generated with the help of FTG are trustworthy?



Bemerkung: *Wenn ein Test läuft, ist dieser zuverlässig, d.h. es ist unwahrscheinlich, dass er inkorrekterweise ohne Fehler durchläuft. Zumindest habe ich solch einen Fall noch nicht bemerkt.*

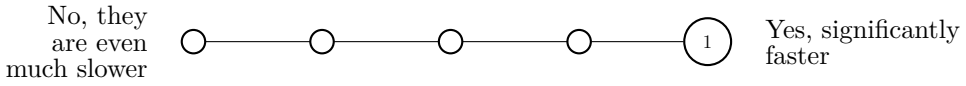
T5 Do you think, the use of FTG saves time compared to the manual writing of unit tests?



Bemerkung: *Ja. Hätten die ICON-EntwicklerInnen bereits testgetrieben entwickelt, hätte uns*

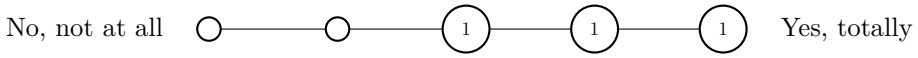
dies viel Zeit gespart. Ordentliche Unittests für ICON zu erstellen würde geschätzt ein Personenjahrzehnt dauern, während die Erstellung mit FTG einschließlich Fehlerkorrekturen einen Personenmonat gebraucht hat.

T6 Do FTG tests run faster than alternative end-to-end tests (i.e. normal model runs with suitable configurations)?



Bemerkung: *Einzelne Codeabschnitte („kernels“) laufen oft schneller als wenn diese innerhalb des Modells ausgeführt werden.*

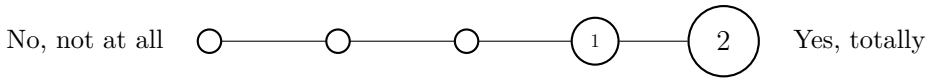
T7 Do you think, FTG encourages to create more unit tests?



Bemerkung: *Ich denke schon, aber viele ICON-EntwicklerInnen scheinen dies anders zu sehen.*

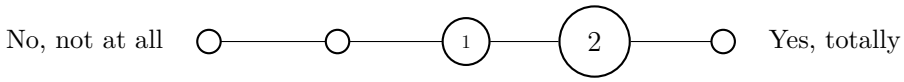
Benutzbarkeit

B1 Do you think, the general method of test data extraction implemented by FTG (Capture & Replay) is understandable?



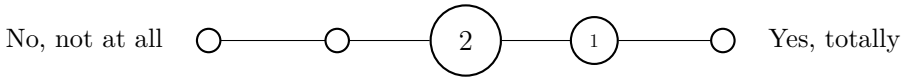
Bemerkung: *Während die Grundidee einfach zu verstehen ist, hatte ich einige Schwierigkeiten mit den einzelnen Prozessschritten, die zum Teil nicht ganz intuitiv sind.*

B2 Do you think, an average scientific programmer without formal education in computer science or software engineering will be able to use FTG?



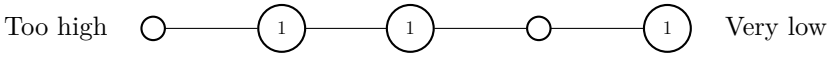
Bemerkungen: *Ich würde 4 antworten für gut strukturierte, d.h. modulare Software. Für große monolithische Softwarehaufen könnte der Einsatz von FTG mühsam sein. / Wenn ich es benutzen kann (als „Computational“), kann dies jede andere technisch gebildete Person auch.*

B3 Do you think, FTG fits well into your regular tool box?



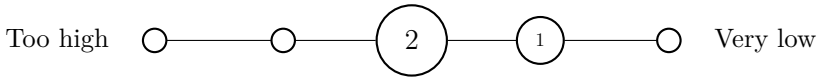
Bemerkung: *Dies ist ein Werkzeug speziell für ICON, mit begrenztem Nutzen außerhalb. Für mich als Entwickler, der ICON-Code auf GPUs portiert, sollte es zu meinem regulären Werkzeugkasten gehören. ICON-EntwicklerInnen, die wissenschaftlichen Code entwickeln, mögen vielleicht zustimmen, vielleicht aber auch nicht.*

B4 The initial effort for setting up FTG for your project was...



Bemerkung: *Das Verlassen auf Cheetah/Cheetah3 war und ist noch immer problematisch. Zudem gab es etwas Verwirrung um die Python-Version auf unserem Cray-System, welche inzwischen gelöst sein sollten.*

B5 The effort of using FTG for subsequent cases was. . .

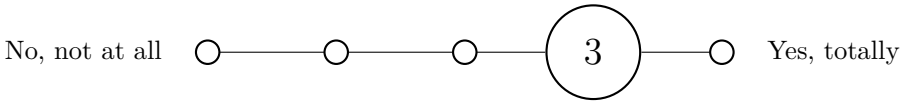


Bemerkungen: *Dies war sehr vom jeweiligen Standalones abhängig. Einige erzeugten sehr einfach Tests mit wenigen Abhängigkeiten, während andere die Hälfte von ICON benötigen und viel Arbeit, um zum Laufen gebracht zu werden. / Irgendetwas geht immer schief, so dass der Standalone-Code bearbeitet werden muss.*

B6 Have you modified or extended code generated by FTG?

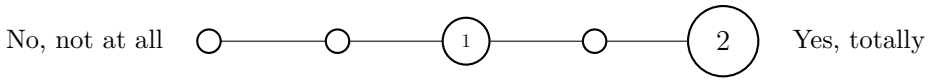
Bemerkungen: *Meistens um zusätzliche Daten aufzuzeichnen oder Benchmark-Informationen wie Zeitmessung einzufügen. / Fast immer notwendig bei den nicht trivialen Standalones, die wie erzeugt haben.*

B7 Do you think, the generated code is well readable?



Bemerkung: *Viel `ftg_*`-Code, aber grundsätzlich klar, was es tut.*

B8 Do you think, the generated code is easily modifiable?

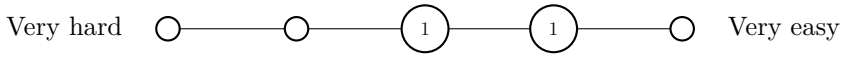


Bemerkung: *Die Anpassungen sind manchmal relativ kompliziert, insbesondere wenn es darum geht, abstrakte Datentypen, die nicht vernünftig aufgezeichnet werden, zu füllen.*

B9 Have you modified a [sic] FTG template?

Bemerkung: *Ich habe Störungen („pertubation options“) in die serialisierten Felder eingebaut, um Fehlergrößen und -fortpflanzungen zu untersuchen. Zudem habe ich OpenACC-Datentransfers in die Templates eingebaut, um unnötige Datentransfers innerhalb der portierten Routinen zu vermeiden.*

B10 If yes, how easy was the modification of the template?



Bemerkung: *Vermutlich konzeptionell einfach, aber ich bin leicht zu verwirren.*

KGEN

K1 Have you ever used KGEN?

K2 If yes, please give a short summary of your experience! *Versucht und sofort aufgegeben, da es keine neueren Fortran-Konstrukte (2003, 2008) unterstützt. / Ich habe es kurz probiert als es erstmals veröffentlicht wurde und bin schnell auf unüberwindbare Probleme gestoßen.*

K3 What do you think are the advantages/disadvantages of KGEN compared to FTG?

Ich denke, FTG ist benutzbarer (d.h. überhaupt benutzbar) im Kontext der ICON-Entwicklung. Muss aber zugeben, dass ich keine neuere KGEN-Versionen ausprobiert habe.

11.3.3 Diskussion

Der FortranTestGenerators konnte im ENIAC-Projekt erfolgreich eingesetzt werden (T1) und der Einsatz hat sich als nützlich erwiesen (T2). Es wird jedoch darauf hingewiesen, dass dies nur mit diversen manuellen Anpassungen möglich war, die zum Teil sehr aufwendig waren. Hier spielte jedoch auch der Entwicklungsstand der Werkzeuge zum Zeitpunkt der Studie eine Rolle. Insbesondere dynamische Datenstrukturen, basierend auf abstrakten Typen bereiteten Schwierigkeiten. fcg und FTG boten hier noch nicht die Möglichkeit, einzelnen abstrakten Typen bestimmte konkrete Untertypen zuzuordnen (siehe auch Abschnitt 10.2.2). Mit dieser Funktion konnten viele der Probleme, mit denen die ENIAC-Entwickler konfrontiert waren, behoben werden. Die in dieser Arbeit durchgeführten Experimente zeigen jedoch, dass nach wie vor einige Anpassungen unumgänglich sind.

Trotz der Schwierigkeiten konnte durch den Einsatz von FTG viel Zeit bei der Erstellung von Tests eingespart werden (T5). Die erstellten Tests halfen den Entwicklern dabei, etliche Fehler aufzudecken (T3), und werden als vertrauenswürdig bewertet (T4). Die Frage zur Laufzeit der erstellten Tests im Vergleich zur Laufzeit alternativer E2E-Tests wurde nur von einem Teilnehmer beantwortet (T6). Der abgegebene Kommentar legt nahe, dass diese etwas missverständlich formuliert war. Die Frage, ob FTG motiviert, mehr Unittests zu erstellen (T7), wird überwiegend positiv beantwortet, mit einer Einschränkung hinsichtlich der Einstellung anderer ICON-EntwicklerInnen zu dieser Frage.

Die Capture-&-Replay-Methode wird allgemein als gut verständlich bewertet (B1). Zudem herrscht überwiegend der Eindruck, dass eine durchschnittliche wissenschaftliche ProgrammiererIn in der Lage sein dürfte, FTG zu benutzen – mit einer kleinen Einschränkung in Abhängigkeit von der Architektur des Zielsystems (B2). Bei den Teilnehmern scheint jedoch der Eindruck zu bestehen, dass FTG ein ICON-spezifisches Werkzeug sei (Kommentare zu B3). Dies ist insofern richtig, als das ICON das Haupttestobjekt während der Entwicklung von FTG darstellte und die in ICON vorgefundenen Sprachkonstrukte wesentlich bei der Priorisierung der zu implementierenden Unterstützung war. Zudem existierten zum Zeitpunkt der Studie neben den generischen Templates nur ICON-spezifische Templates. Die in dieser Arbeit durchgeführten Experimente zeigen jedoch, dass FTG durchaus auch mit anderen Klimamodellen funktioniert.

Die Antworten auf die Frage nach dem initialen Aufwand, FTG für das Projekt einzurichten (B4), ergab kein eindeutiges Bild. Der Aufwand für die Erstellung einzelner Tests wird überwiegend als mittelhoch bewertet (B5). Auch hier wird wieder auf die notwendigen Anpassungen, deren Umfang sich jedoch von Fall zu Fall unterscheidet, hingewiesen.

Der von FTG generierte Code wird als gut lesbar (B7) und überwiegend als leicht änderbar (B8) beschrieben, mit einer Einschränkung, die sich erneut auf abstrakte Datentypen bezieht. Von der Modifizierbarkeit der Templates wurde auf interessante Weise Gebrauch gemacht (B9), wobei die Änderbarkeit der Templates als mittel bis leicht bewertet wird (B10).

Insgesamt lässt sich sagen, dass die Bewertung von Tauglichkeit und Benutzbarkeit des FortranTestGenerators durch die an der Studie beteiligten Entwickler überwiegend positiv ausfällt. Die Antworten auf die Fragen zu KGEN bestätigen bisherige Erfahrungen, dass dessen Einsatz selten erfolgreich ist. Ein praktischer Vergleich mit FTG lässt sich daher nicht aufstellen.

11.4 Zusammenfassung

In diesem Kapitel wurde die Validierung und Evaluation des FortranTestGenerators beschrieben. Gemäß Wieringa (2014, S. 31) versteht man in der Design-Science-Methode unter Validierung die Erprobung eines Artefakts unter kontrollierten Laborbedingungen, während mit Evaluation die Beobachtung des Einsatzes eines Artefakts im realen Kontext gemeint ist.

Zu Validierung wurden mit dem FortranTestGenerator Unittests für 17 Prozeduren aus vier verschiedenen Klimamodellen erzeugt. Die Ein- und Ausgabedaten wurden dabei nach dem Capture-&-Replay-Prinzip aus bereits vorhandenen E2E-Tests der jeweiligen Modelle extrahiert. In 12 Fällen konnten die Prozedurausführungen erfolgreich reproduziert werden, d.h. die Prozeduren lieferten in den Unittests für alle Ausgabevariablen die exakt gleichen Ergebnisse wie in den E2E-Tests. In 3 Fällen gab es in Abweichungen in einzelnen Variablen, die jedoch auf nichtdeterministisches Verhalten zurückzuführen sind und sich somit auch zwischen zwei E2E-Test-Ausführungen ergeben. Diese Reproduktionen können somit auch als erfolgreich bezeichnet werden. In einem Fall gab es Abweichungen in einzelnen Variablen, für die keine Erklärung gefunden werden konnte. In einem anderen Fall konnte kein lauffähiger Test erzeugt werden. Grund hierfür waren auf Vererbung basierende dynamische Datenstrukturen, die nicht vollständig analysiert und wiederhergestellt werden konnten.

Die hohe Erfolgsquote sowie die Tatsache, dass unter den erfolgreichen Prozeduren aus einige sehr hochrangige Prozeduren, mit bis über 1.000 Unterprozeduren waren, zeigt die grundsätzliche Funktionsfähigkeit des Ansatzes bzw. des Werkzeugs. Jedoch waren für fast alle Prozeduren manuelle Anpassungen am Anwendungs- oder Testcode nötig.

Daneben wurden einige quantitativen Aspekte diskutiert, insbesondere der zeitliche Aufwand der Testerstellung, die Laufzeit der Unittests verglichen mit den jeweiligen

E2E-Tests sowie die Codeabdeckung der Unittests. Der zeitliche Aufwand für Codeanalyse und -generierung erscheint – intuitiv betrachtet – für ein derartiges Werkzeug im akzeptablen Bereich. Für die hochrangigste Prozedur mit über 300.000 zu analysierenden Codezeilen lag der Aufwand auf dem PC bei rund 5 Minuten und auf dem Login-Knoten der Mistral bei etwas mehr als 8 Minuten. Bei den niederrangigeren Prozeduren mit unter 100.000 zu analysierenden Codezeilen lag der Aufwand zwischen einer Sekunde und einer halben Minute.

Bei der Laufzeit der Tests ergibt sich ein deutlicher Vorteil der Unittests gegenüber vollständigen E2E-Tests, d.h., wenn diese den gesamten in ihrer Konfiguration vorgegebenen Zeitraum simulieren. Verkürzt man die Laufzeit der E2E-Tests auf einen Zeitschritt, ergibt sich nur für einige der untersuchten Unittests eine signifikant kürzere Laufzeit. Dies betrifft insbesondere Ausführungen auf der Mistral. Festzustellen ist hierbei, dass der Laufzeitvorteil der Unittests mit wachsender Anzahl paralleler Prozesse abnimmt. Dabei macht sich mutmaßlich der hohe I/O-Aufwand der Capture-&-Replay-Tests bemerkbar. Berücksichtigt man zusätzlich die sich aus dem Mehrbenutzersystem von Hochleistungsrechnern wie der Mistral ergebenden Wartezeiten, muss man feststellen, dass sich hier aus der Verwendung der Unittests kaum Zeitgewinne ergeben. Anders sieht dies auf dem PC aus. Hier waren die Unittests in den meisten Fällen deutlich schneller als die entsprechenden E2E-Tests mit einem Zeitschritt.

Bei der Codeabdeckung erreichten die meisten der generierten Unittests gegenüber den vollständigen E2E-Tests sehr gute Werte. Bei Betrachtung aller Unterprozeduren erreichten die Unittests sowohl bei der Prozedur- als auch bei der Zeilen- und Zweigabdeckung im Schnitt über 80 % der Abdeckung der jeweiligen E2E-Tests. Auf die eigentlich zu testende Prozedur bezogen lag die Zeilen- und Zweigabdeckung nur bei zwei Prozeduren unterhalb der Abdeckung der E2E-Tests.

Zur Evaluation wurde eine Benutzerstudie mit drei Entwicklern durchgeführt, die den FortranTestGenerator verwendeten, um Tests für die Portierung von Teilen des ICON-Modells auf GPUs zu erstellen. Nach etwas mehr als einem Jahr der Arbeit mit dem FortranTestGenerator wurden diese mit Hilfe eines Online-Fragebogens nach ihrer Erfahrung mit dem Werkzeug befragt. Dabei wurden sowohl Tauglichkeit wie auch Benutzbarkeit des Werkzeugs positiv bewertet. Auch hier waren jedoch in vielen Fällen manuelle Anpassungen notwendig sind, um funktionierende Tests zu erhalten. Zu berücksichtigen ist dabei jedoch, dass sich der FortranTestGenerator zum Zeitpunkt der Studie noch in einem frühen Entwicklungsstadium befand.

Zusammenfassend lässt sie urteilen, dass der FortranTestGenerator ein funktionsfähiges, taugliches und benutzbares Werkzeug zur automatisierten Unterstützung der Unittesterzeugung für einzelne Prozeduren von Klimamodellen ist. Auch das durch das Werkzeug umgesetzte Capture & Replay erscheint somit als ein tauglicher Ansatz für die Unittesterstellung im Kontext Klimamodellierung.

Kapitel 12

Zusammenfassung und Ausblick

Klimamodelle sind Softwaresysteme, mit denen sich numerische Simulationen des Klimasystems der Erde durchführen lassen. Sie sind das wichtigste Werkzeug der Klimawissenschaften, um die physikalischen und chemischen Prozesse, die das Klima der Erde bestimmen, zu erforschen und um Prognosen für die Zukunft zu erstellen. Im Hinblick auf die globale Erwärmung hängen nicht nur der wissenschaftliche Erkenntnisgewinn, sondern auch politische Entscheidungen von höchster gesellschaftlicher Bedeutung von der Zuverlässigkeit der Modelle ab. Um diese zu gewährleisten und den Softwareentwicklungsprozess beherrschbar zu gestalten, spielt das Testen der Modellsoftware in der Klimamodellierung eine ebenso so große Rolle wie in anderen Softwarenentwicklungskontexten.

Jedoch verfügen Klimamodelle über spezifische Eigenschaften, die sie von typischen in der Softwaretestliteratur untersuchten Systemen unterscheiden, u.a.:

- Ihre Komplexität entsteht nicht aus zahlreichen Funktionen und Interaktionsmöglichkeiten mit der BenutzerIn, sondern aus der Komplexität der parallelen, numerischen Algorithmen, den Wechselwirkungen zwischen den verschiedenen Modellkomponenten sowie ihrer hohen Konfigurierbarkeit.
- Es existieren keine „korrekten“ Modelle, sondern nur nützliche Approximationen. Somit können ihre Ergebnisse nicht definitiv als richtig oder falsch, sondern nur nach ihrer Güte bewertet werden.
- Sie benötigen ein hohes Maß an Rechenressourcen und werden daher i.d.R. auf Hochleistungsrechnern, die von entsprechenden Rechenzentren bereitgestellt und verwaltet werden, ausgeführt.
- Ihre EntwicklerInnen sind i.d.R. ExpertInnen ihrer jeweiligen wissenschaftlichen Disziplin und keine ausgebildeten SoftwareentwicklerInnen oder InformatikerInnen.
- Sie sind in Fortran geschrieben.

Hieraus ergab sich die erste von vier Forschungsfragen: Wie sieht die Praxis des Softwaretestens in solch einem Umfeld aus? Um diese Frage zu beantworten, wurde in dieser Arbeit eine qualitative Studie an vier Klimaforschungsinstituten durchgeführt. Diese beinhaltete u.a. eine mehrjährige Feldstudie am Max-Planck-Institut für Meteorologie in Hamburg, das zusammen mit dem Deutschen Wetterdienst das ICON-Modell entwickelt. Dabei wurden u.a. Interviews geführt, eine Umfrage durchgeführt und an Entwicklertreffen teilgenommen. Ergänzt wurde dies durch Interviews am Geophysical Fluid Dynamics Laboratory (GFDL) in Princeton, USA, der Japan Agency for Marine Earth Science and Technology (JAMSTEC) in Yokohama, Japan sowie dem RIKEN Center for Computational Science in Kobe, Japan.

Ein Ergebnis dieser Studie war, dass neben der wissenschaftlichen Evaluation der Modelle, für die es eine Vielzahl etablierter Methoden gibt, das Regressionstesten eine hohe Bedeutung einnimmt. Dabei werden bereits getestete Funktionalitäten und Eigenschaften nach Änderungen am Modell erneut überprüft. Dies beinhaltet zum Beispiel technische Tests, etwa ob das Kompilieren und Ausführen mit verschiedenen Compilern und Rechnerplattformen funktioniert, ob der Checkpoint/Restart-Mechanismus korrekt arbeitet oder dass die Anzahl der parallelen Prozesse und Threads keinen Einfluss auf die Ergebnisse hat. Zudem wird überprüft, ob Änderungen, die sich nicht auf die Modellergebnisse auswirken sollen, dies auch nicht tun oder dass Änderungen keine sonstigen unerwünschten Nebenwirkungen haben. Der Umfang und das Vorgehen beim Regressionstesten unterscheiden sich von Team zu Team, ebenso der Grad der Formalisierung und der Automatisierung. Einige Teams haben zudem ihre eigenen speziellen Testmethoden für bestimmte Fragestellungen entwickelt.

Gemeinsam ist den Teams, dass diese auch beim Regressionstesten zum größten Teil auf eine Form von Tests setzen, die in dieser Arbeit Ende-zu-Ende-Tests genannt wird. Dabei wird das Hauptprogramm des Klimamodells ausgeführt und alle Phasen einer Simulation durchlaufen. Dazu gehören die Modellinitialisierung, die Zeitschleife, die Aufräumphase und ggf. das Postprocessing. Unittests kleinerer Codeabschnitte wie etwa einzelner Prozeduren werden hingegen nur sehr selten eingesetzt. Dies gilt insbesondere für die fachlichen, „wissenschaftlichen“ Prozeduren.

In dieser Arbeit wird argumentiert, dass entgegen der herrschenden Praxis Unittests eine sinnvolle Ergänzung zu den bestehenden Ende-zu-Ende-Tests darstellen können. So ließen sich diese im Allgemeinen schneller kompilieren und ausführen und liefern so den EntwicklerInnen ein schnelleres Feedback während der Entwicklung. Zudem erleichtern sie durch ihre Fokussierung auf einen begrenzten Codeabschnitt die Lokalisierung von Fehlerursachen sowie die Analyse und Reproduzierbarkeit der Testergebnisse. Dies wird bestätigt durch die beschriebene Studie, in der die EntwicklerInnen über gute Erfahrungen mit den wenigen vorhandenen Unittests berichteten.

Als Antwort auf die Forschungsfrage 2 wurde in dieser Arbeit somit die Nichtverwendung von Unittests als softwaretechnisches Defizit identifiziert.

Um die Forschungsfrage 3 zu beantworten, wurden die Gründe für die Abwesenheit von Unittests untersucht. Neben menschenbezogenen Gründen und dem Testorakelproblem, welches sich beim Regressionstesten i.d.R. nicht so stark stellt, wurde vor allem der als zu hoch angesehene Aufwand für die Erstellung der Tests als Hindernis identifiziert. Hierbei steht insbesondere der Aufwand für die Erstellung geeigneter numerischer Testdaten im Vordergrund. Viele Prozeduren in Klimamodellen benötigen eine hohe, zum Teil dreistellige Anzahl von Variablen als Eingabedaten, bei denen es sich zu einem Großteil wiederum um Arrays mit tausenden Elementen handelt. Eine manuelle Erstellung sinnvoller, konsistenter Belegungen all dieser Datenpunkte erscheint so kaum möglich.

Die Forschungsfrage 4 befasst sich daher mit der Frage wie EntwicklerInnen von Klimamodellen beim Erstellen von Unittests unterstützt werden können. Da mit einer derartigen Unterstützung vor allem der Aufwand für die Erstellung von Testdaten reduziert werden soll, ist es notwendig, Teile der damit verbundenen Arbeit zu automatisieren. Als Leitlinie für die Gestaltung dieser Automatisierung dient in dieser Arbeit die Anwendungsorientierung. Dies bedeutet, dass bei der Formulierung der Anforderungen an eine Lösung die AnwenderInnen, d.h. die EntwicklerInnen von Klimamodellen, und deren Aufgaben im Vordergrund stehen. Zu den Anforderungen gehören daher u.a. eine benutzergerechte Handhabung, die sich an den Fähigkeiten und Gewohnheiten der AnwenderInnen orientiert, sowie die Bedingung, nützliche Tests für reale Klimamodellen erstellen zu können. Diese sollen zudem signifikant schneller sein als vergleichbare Ende-zu-Ende-Tests. Hinzu kommen weitere Randbedingungen, die sich aus dem (software-)technischen Umfeld der Klimamodellierung ergeben.

Daraufhin wurden fünf Ansätze analysiert, die in der Literatur unter dem Stichwort „automatische Testdatenerstellung“ diskutiert werden: die symbolische Ausführung, das modellbasierte Testen, das kombinatorische Testen, das zufallsbasierte Testen sowie das suchbasierte Testen. Dabei erschien keiner der Ansätze geeignet, um im Rahmen dieser Arbeit als Grundlage für eine anwendungsorientierte Lösung für die Erstellung großer Mengen numerischer Testdaten zu dienen.

Als alternative Lösung wurde daher der Capture-&-Replay-Ansatz ausgewählt. Dabei handelt es sich um ein halbautomatisches Verfahren, bei dem Testdaten aus bereits existierenden, aufwendigeren Testfällen, wie etwa manuellen GUI-Tests oder wie hier aus umfangreichen Ende-zu-Ende-Tests gewonnen werden. Die Auswahl des Testfalls obliegt der EntwicklerIn bzw. TesterIn selbst, während die Testdaten mit Hilfe eines geeigneten Werkzeugs bei der Ausführung des Ende-zu-Ende-Tests automatisiert aufgezeichnet werden.

Während das allgemeine Prinzip des Capture & Replay bereits von anderen Autoren beschrieben und umgesetzt wurde – zum Teil unter anderen Namen, liegt der Beitrag dieser Arbeit in der Übertragung des Verfahrens auf den Anwendungsfall Klimamodellierung, der Analyse der in diesem Kontext notwendigen Arbeitsschritte sowie der Optionen bei der Gestaltung eines entsprechenden Werkzeugs, auch unter anwendungsorientierten Gesichtspunkten, und schließlich in der Umsetzung eines neuartigen Werkzeugs, dem FortranTestGenerator (FTG; GitHub, fortesg).

Mit Hilfe des FortranTestGenerator lassen sich auf Capture & Replay basierende Unittests für einzelne Prozeduren von Klimamodellen oder anderen Fortran-Anwendungen erzeugen. Dazu generiert FTG zum einen Code zur Instrumentierung der Originalanwendung, der dazu dient alle Ein- und Ausgabedaten aufzuzeichnen, und zum anderen ein Testprogramm, in dem diese Eingabedaten geladen werden, die zu testende Prozedur mit diesen ausgeführt und das Ergebnis mit den aufgezeichneten Ausgabedaten validiert wird. Der so generierte Test kann anschließend von der EntwicklerIn/TesterIn bearbeitet werden, indem sie etwa Überprüfungen entfernt bzw. ergänzt oder indem sie durch Veränderung der Eingabedaten weitere Testfälle erzeugt.

FTG besteht aus zwei Komponenten: dem separaten Werkzeug FortranCallGraph (fcg), welches mit Hilfe einer statischen Codeanalyse die Ein- und Ausgabevariablen der zu testenden Prozedur identifiziert und dem eigentlichen Testgenerator, der auf Basis der von fcg gelieferten Informationen den Instrumentierungs- und Testcode generiert. Die in FortranCallGraph implementierte Codeanalyse basiert auf einem pragmatischen Ansatz, bei dem mit Hilfe regulärer Ausdrücke zum einen nach Verwendungen einzelner Dummyargumente und globaler Variablen gesucht wird und zum anderen nach Konstrukten, die weitere Analysen nach sich ziehen, wie etwa Prozeduraufrufe oder Zuweisungen. Die Codegenerierung des FortranTestGenerators basiert auf anpassbaren Templates, die mit Hilfe der Template-Engine Cheetah (GitHub, Cheetah3) verarbeitet werden. Dadurch ist es möglich, den generierten Code umfangreich an die jeweilige Zielumgebung anzupassen. So können etwa Sonderbehandlungen für bestimmte Variablen oder Datentypen eingefügt werden, Bibliotheken in das Testprogramm eingebunden werden oder die Art und Weise, wie Ein- und Ausgabedaten geschrieben und gelesen werden, bestimmt werden. Die auf GitHub veröffentlichte Version des FortranTestGenerators enthält bereits einige vorgefertigte Standardtemplates. Diese verwenden die I/O-Bibliothek Serialbox2 (GitHub, SB2) zum Speichern und Laden der Variablenbelegungen.

Gleichzeitig zu dieser Arbeit entstand am US-amerikanischen Klimaforschungsinstitut NCAR ein vergleichbares Werkzeug namens KGEN (GitHub, KGEN; Kim u. a., 2016). Auch dieses ist in der Lage, auf Capture & Replay basierende, isolierte Tests einzelner Prozeduren (bzw. beliebiger Codeabschnitte) zu erstellen. Die wesentlichen Unterschiede zwischen dem FortranTestGenerator und KGEN sind:

- Der andere Codeanalyse-Ansatz: die auf einem abstrakten Syntaxbaum basierende Codeanalyse von KGEN bedingt, dass der Parser den gesamten Quellcode syntaktisch verarbeiten kann. Da der eingesetzte Parser jedoch viele neuere Sprachkonstrukte von Fortran nicht unterstützt, führt dies häufig zu Abstürzen. Der in dieser Arbeit umgesetzte pragmatische Analyseansatz bedingt nicht, den gesamten Quelltext zu verstehen, sondern sucht darin nur nach relevanten Konstrukten. Dies macht die Analyse robuster.
- KGEN zeichnet für Ein- und Ausgabevariablen mit benutzerdefiniertem Verbunddatentyp immer sämtliche Komponenten auf, während FTG sich auf die Komponenten beschränkt, die von der zu testenden Prozedur tatsächlich benötigt werden.
- Mit Hilfe der Templates kann der von FTG generierte Code umfangreich an die jeweilige Zielumgebung angepasst werden. Dies ist mit KGEN nicht möglich.
- Während der Code zum Aufzeichnen der Daten in KGEN über die ganze Anwendung verteilt wird und sich aufgrund seiner Struktur kaum zum Bearbeiten durch die EntwicklerIn eignet, wird dieser von FTG nur an definierten Punkten in das Modul, welches die zu testende Prozedur enthält, eingefügt. Zudem wurde bei der Gestaltung der mitgelieferten Standardtemplates auf gute Lesbarkeit des resultierenden Codes geachtet.
- Der gesamte Testprozess läuft in KGEN vollautomatisch von der Codegenerierung über das Kompilieren bis zur Ausführung und Auswertung des Tests. Im Sinne Buddes und Züllighovens (1990) stellt es sich daher eher wie ein *Automat* dar, während der FortranTestGenerator stärkere *werkzeugartige* Züge aufweist. Zwar läuft die Codeanalyse und -generierung in FTG auch, einmal angestoßen, automatisch ab, jedoch behält die AnwenderIn die Kontrolle über den Gesamtprozess der Testerstellung. So kann sie die Reihenfolge des Vorgehens in bestimmten Grenzen selbst bestimmen, mit Hilfe der Templates die Codegenerierung beeinflussen sowie Zwischen- und Endprodukte bearbeiten.

Erprobt wurde der FortranTestGenerator zum einen mit Hilfe von Experimenten mit vier verschiedenen Klimamodellen und zum anderen mit einer Benutzerstudie im Rahmen eines realen Entwicklungsprojekts. Die Experimente bestanden aus der Generierung von Tests für insgesamt 17 Prozeduren und dem Vergleich der Ergebnisse, welche diese in Test und Originalanwendung liefern. Dabei konnte die allgemeine Funktionsfähigkeit des Werkzeugs gezeigt werden. In zwölf Fällen erzeugten die Prozeduren in Test und Originalanwendung vollständig identische Ergebnisse. In drei Fällen gab es lediglich einzelne Abweichungen aufgrund von Nichtdeterminismus innerhalb der Prozeduren. In einem Fall gab es Abweichungen in einzelnen Variablen, für die keine Erklärung gefunden werden konnte und in einem Fall konnte kein

lauffähiger Test erzeugt werden. Grund hierfür waren auf Vererbung basierende dynamische Datenstrukturen, die von fcg/FTG (noch) nicht vollständig analysiert und wiederhergestellt werden können.

Die quantitative Auswertung der Experimente lieferte u.a. zwei zentrale Ergebnisse:

1. Die erzeugten Unittests erreichen mit einer einzelnen Prozedurausführung fast die gleiche Codeabdeckung wie die Ende-zu-Ende-Tests aus denen die jeweiligen Testdaten aufgezeichnet wurden. Betrachtet man sämtliche Unterprozeduren der zu testenden Prozeduren lag die Prozedur-, Zeilen- und Zweigabdeckung in den Experimenten im Mittel bei über 80 %, verglichen mit den entsprechenden Werten der jeweiligen Ende-zu-Ende-Tests. Allein auf die zu testende Prozedur beschränkt lag die relative Zeilen- und Zweigabdeckung nur in zwei von zwölf ausgewerteten Fällen bei unter 100 %.
2. Bei Ausführung auf dem Test-PC erreichten die meisten Unittests deutlich kürzere Laufzeiten als entsprechenden Ende-zu-Ende-Tests, auch wenn deren Simulationszeit auf einen Zeitschritt reduziert wurde. Auf dem Hochleistungsrechner Mistral am DKRZ war der Zeitgewinn nicht so hoch wie erhofft. Je höher die Zahl der parallelen Prozesse, mit denen beide Testarten ausgeführt wurden, desto geringer fiel dieser aus. Berücksichtigt man zudem die Wartezeiten, die sich aus dem Mehrbenutzerbetrieb des Hochleistungsrechners ergeben, entsteht aus der Verwendung der Unittests kaum ein Zeitvorteil verglichen mit kurz laufenden Ende-zu-Ende-Tests.

Die Benutzerstudie wurde mit drei Entwicklern eines Projekts, das sich mit der Portierung von Teilen des Klimamodells ICON auf GPUs befasst, durchgeführt. Mit den hierbei erstellten Tests sollten vor allem die Ergebnisse der portierten Prozeduren mit deren ursprünglichen Ergebnissen verglichen werden. Nach etwas mehr als einem Jahr der Arbeit mit dem FortranTestGenerator wurden die beteiligten Entwickler mit Hilfe eines Online-Fragebogens zu ihren Erfahrungen mit dem Werkzeug befragt. Dabei bewerteten sie die Tauglichkeit und Benutzbarkeit überwiegend positiv und den Einsatz des Werkzeugs als erfolgreich.

Sowohl in den Experimenten als auch in der Benutzerstudie wurde jedoch deutlich, dass in den meisten Fällen manuelle Anpassungen am Anwendungs- oder Testcode notwendig sind, um einen funktionierenden Test zu erstellen. Hieraus ergeben sich Ansätze für eine mögliche Weiterentwicklung des Werkzeugs. So könnte etwa die Unterstützung für vererbungsbedingte Polymorphie verbessert werden. Zwar können die sich hieraus konkret ergebenden dynamischen Strukturen statisch nicht ermittelt werden, jedoch könnte stattdessen Code generiert werden, der diese zur Laufzeit erkennt. Weitere kleinere Verbesserungen wären eine erweiterte Unterstützung veralteter FORTRAN-77-Syntax oder die Unterstützung von SUBROUTINEN außerhalb von Modulen.

Diese Erweiterungen wären relativ einfach umzusetzen und benötigten nur weitere Entwicklungsarbeit. Andere Anpassungen waren aufgrund der grundsätzlichen Funktionsweise des FortranTestGenerator nötig. So können etwa individuell als `PRIVATE` oder `PROTECTED` deklarierte Variablen nicht automatisch exportiert werden, da FTG nur Code einfügen – an definierten Einfügepositionen, aber keinen Code entfernen kann. Ebenso das Wiederherstellen der Belegung von lokalen `SAVE`-Variablen ist prinzipbedingt nicht möglich, da eine Instrumentierung des zu testenden Codes im Replaymodus nicht vorgesehen ist. Auch Probleme, die aus nicht initialisierten Variablen entstehen, lassen sich durch das Werkzeug nicht beheben. Hilfreich wäre hier eine bessere Unterstützung der AnwenderInnen bei der Identifizierung der notwendigen Anpassungen. So könnte etwa die, ohnehin noch zu erstellende Dokumentation einen Leitfaden enthalten, wie in bestimmten Situationen vorzugehen ist. Auch könnte das Werkzeug Warnungen ausgeben, wenn etwa eine `SAVE`-Variable oder eine individuelle `PRIVATE`-Deklarierung gefunden wurden. In einigen Situationen werden solche Warnungen bereits ausgegeben, z.B. wenn eine Prozedur nicht gefunden werden kann oder eine rekursive Datenstruktur ignoriert wird.

Um einen breiteren Einsatz des FortranTestGenerators, ggf. auch über die Testgenerierung hinaus, zu erlauben, könnte das System der Einfügepunkte flexibler gestaltet werden. Derzeit lässt sich mit FTG nur Code an den vordefinierten Punkten einfügen. Ebenso lässt sich neben dem Instrumentierungscode nur eine weitere Datei für das Testprogramm erzeugen. Auch hier könnte eine Flexibilisierung interessant sein, etwa um zusätzlich zum Test auch Buildskripte oder Dokumentationen zu erzeugen.

Einen Ansatz für weitere Forschung bietet das I/O-Verhalten der mit dem FortranTestGenerator erzeugten Tests. Kernprinzip des Capture-&-Replay-Ansatzes ist es, den Zustand einer Anwendung vor Ausführung des zu testenden Codeabschnitts abzuspeichern und im Test wiederherzustellen, damit der Teil der Anwendung, der zu diesem Zustand geführt hat, während des Tests nicht ausgeführt werden muss. Man kann daher sagen, das Prinzip basiert auf dem Tausch von Rechenressourcen gegen I/O-Ressourcen. Wie sich zeigt, lässt sich auf diese Weise nicht immer eine Verkürzung der Testlaufzeit herstellen. Insbesondere mit steigender Zahl paralleler Prozesse verschiebt sich das Verhältnis zwischen eingesparter Rechenzeit zur zusätzlich benötigten I/O-Zeit in ungünstiger Weise. Zwar ist die Verkürzung der Laufzeit nicht das einzige Ziel der Verwendung von Unittests, auch die Verkürzung der Übersetzungszeit sowie die Fokussierung des Tests gehören maßgeblich dazu, dennoch sollte es das Ziel sein, den mit dem Capture-&-Replay-Prozess verbundenen I/O-Aufwand zu minimieren, sowohl was den benötigten Speicherplatz als auch den zeitlichen Aufwands des Ladens der Daten angeht. Dieser Aspekt konnte in dieser Arbeit noch nicht behandelt werden.

Darüber hinaus wäre eine weitere Evaluation des Capture-&-Replay-Ansatzes und des FortranTestGenerators wünschenswert, sowohl im Kontext Klimamodellierung

als auch in anderen Anwendungsbereichen. Grundsätzlich eignet sich FTG aus technischer Sicht für alle Fortran-Anwendungen. Ein Einsatz ergäbe überall dort Sinn, wo eine manuelle Unittesterstellung ähnlich problematisch ist, entsprechende größere Tests zur Verfügung stehen, aus denen die Testdaten extrahiert werden können, und etwaige MPI-Kommunikation wie in Klimamodellen im Wesentlichen prozedurintern stattfindet. In weitergehenden Evaluationen könnten zudem die Fehlererkennungsfähigkeit systematischer untersucht werden, ebenso wie die praktischen Möglichkeiten der Testfallableitung. Hierzu könnte das Capture-&-Replay-Prinzip auch mit anderen Testmethoden kombiniert werden, wie etwa dem Einheitentest oder der Domänentransformation, dem kombinatorischem Testen in Bezug auf verschiedene Konfigurationsoptionen oder dem metamorphen Testen.

Die Grundlage für diese Dissertation bildeten Beobachtungen der Softwareentwicklungspraxis am Max-Planck-Institut für Meteorologie in Hamburg, welche durch Besuche an anderen Klimaforschungsinstituten ergänzt werden konnten. Die EntwicklerInnen von Klimamodellen investieren viel Zeit in das Testen ihrer Modelle, auch wenn sie dies nicht immer als solches bezeichnen. Gleichzeitig steigen durch komplexer werdende Modelle und zunehmend heterogene Hardwarearchitekturen die Anforderungen an Entwicklungs- und Testprozesse. Zum einen sollen diese beherrschbar bleiben, zum anderen zugänglicher für Kooperationen mit fachfremden ExpertInnen. Vor diesem Hintergrund entstand diese Arbeit mit dem Anspruch, eine anwendungsorientierte Lösung für eine effiziente und effektive Ergänzung zur existierenden Testpraxis zu schaffen. Der vorgeschlagene Lösungsansatz wurde in dieser Dissertation zum einen auf konzeptioneller und methodischer Ebene diskutiert und zum anderen in Form eines Softwarewerkzeugs umgesetzt. Dessen Funktionsfähigkeit, Tauglichkeit und Benutzbarkeit wurde mit Hilfe von Experimenten und dem praktischen Einsatz in einem realen Entwicklungsprojekt erfolgreich erprobt. Ob der in dieser Arbeit vorgeschlagene Ansatz Einzug in die Praxis der Klimamodellierung findet, hängt nicht zuletzt von der Aufgeschlossenheit der EntwicklerInnen gegenüber neuen Methoden und Werkzeugen ab sowie der Bereitschaft der verantwortlichen Personen deren Weiterentwicklung zu fördern. Die Entwicklung des dem FortranTestGenerators ähnlichen Werkzeugs KGEN am NCAR zeigt, dass diese zumindest in Teilen der Community durchaus vorhanden ist.

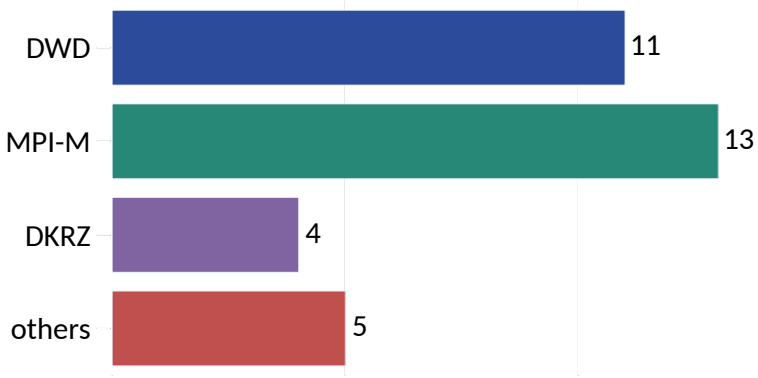
Anhang A

ICON-Umfrage 2014

Im Sommer 2014 wurde im Rahmen dieser Arbeit eine Umfrage unter EntwicklerInnen des Klimamodells ICON zum Thema Testen durchgeführt. Ziel war es, ein erstes Bild von den Testgewohnheiten der EntwicklerInnen zu erstellen. Diese wurde sowohl mit Hilfe von Fragebögen als auch in Form von Interviews durchgeführt. Von 41 EntwicklerInnen, die um Teilnahme an der Umfrage gebeten wurden, nahmen 33 teil. Die Ergebnisse werden im Folgenden im englischen Original aufgeführt. Zum Zeitpunkt der Umfrage verwendete das ICON-Team noch SVN für die Versionskontrolle und hatte noch nicht das Gatekeepersystem etabliert (siehe auch Abschnitt 4.4.1). Jede EntwicklerIn konnte jederzeit Änderungen am Hauptentwicklungszweig (*trunk*) vornehmen.

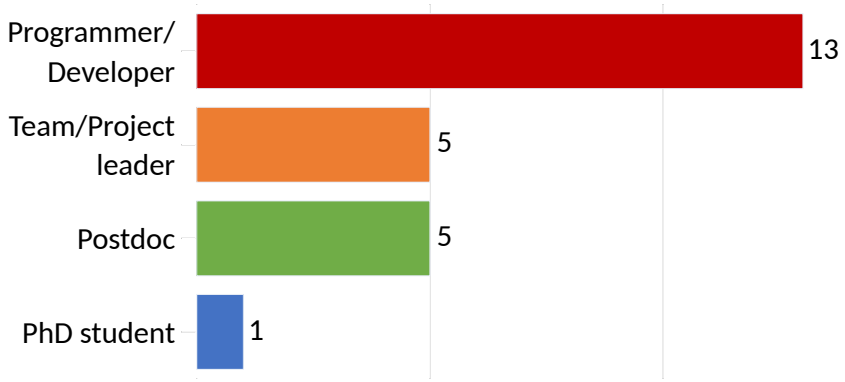
A. General Questions

1. Which is your organization?

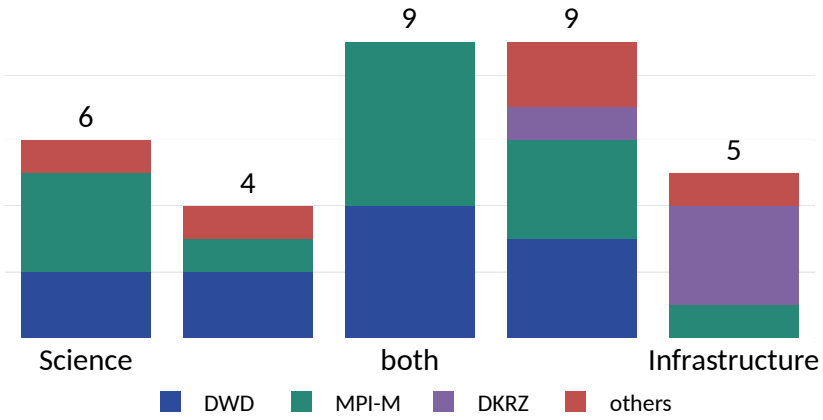


others: 2x CSCS, 1x KIT, 1x TROPOS

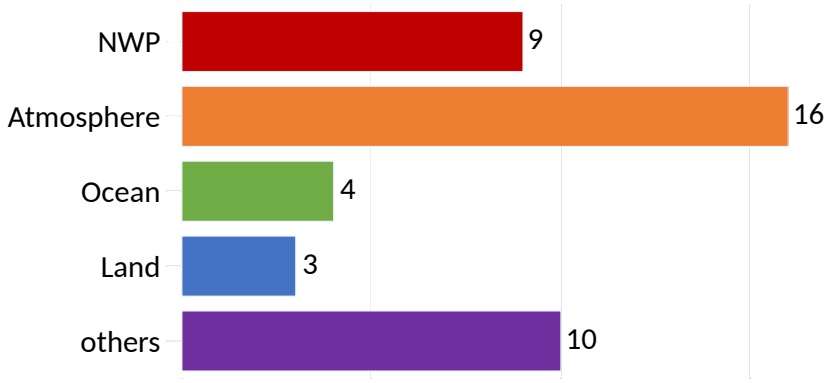
2. What is your position?



3. What is your Focus? Science or Technical Infrastructure?

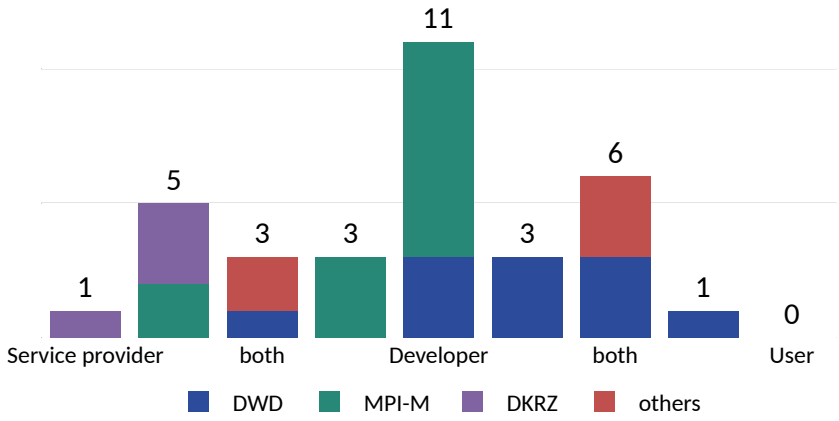


4. What is your main modelling area?

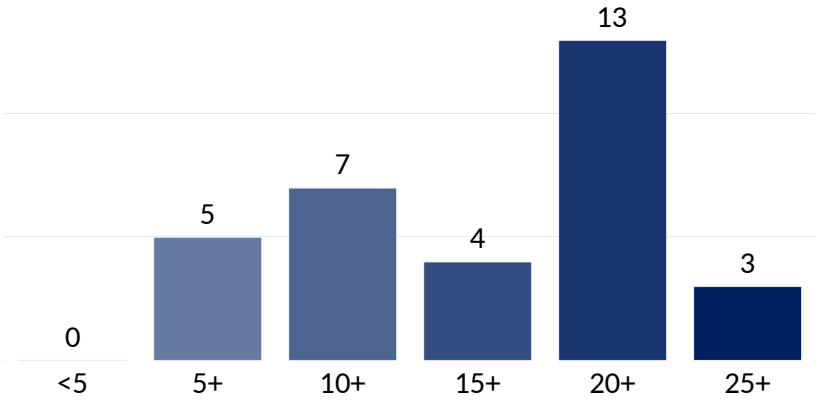


others: "Infrastructure", "Parallelization", "Optimization", "Large Eddy Simulation", "Aerosols", "Cloud Microphysics", "Land/Atmosphere interaction"

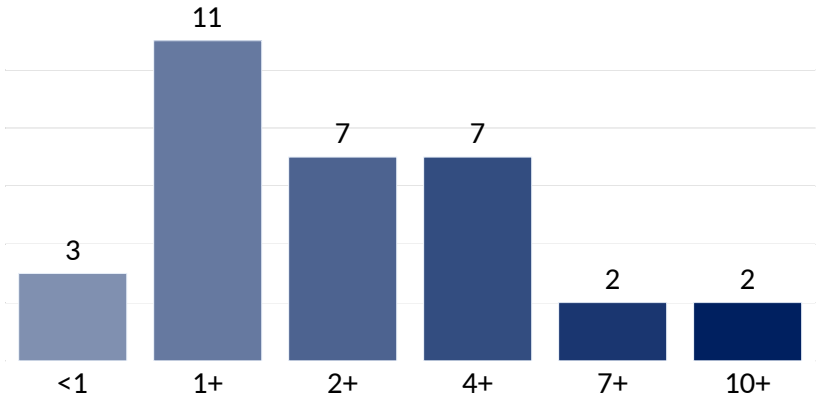
5. Do you see yourself more as an ICON developer, as a user, or as a service provider for the project?



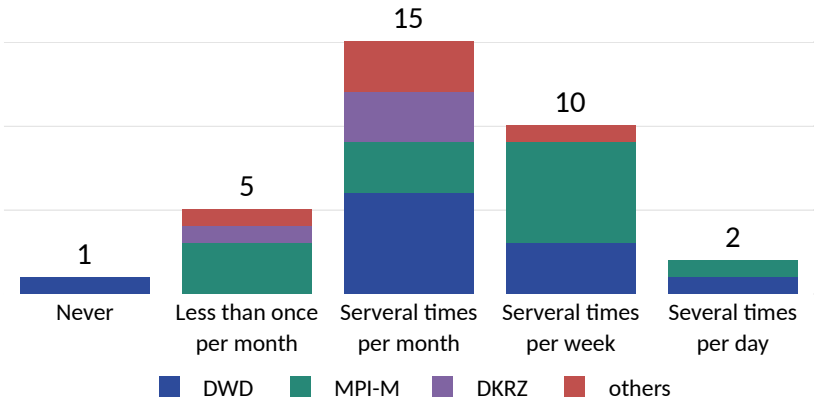
6. How many years of programming experience do you have?



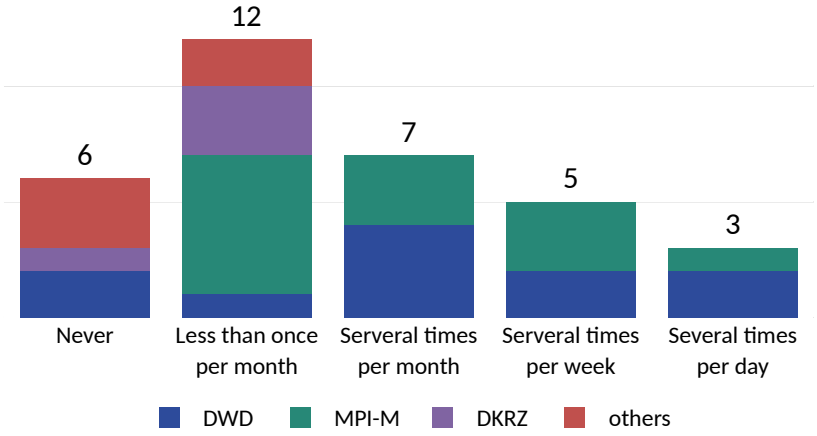
7. How long have you been working with ICON?



8. How often have you checked out a new ICON version from the SVN trunk in the last twelve months?



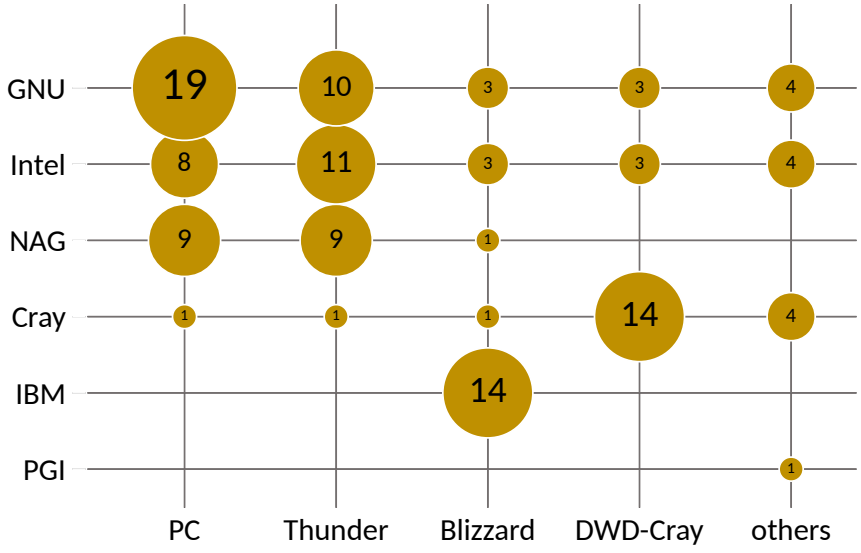
9. How often have you committed to the SVN trunk in the last twelve months?



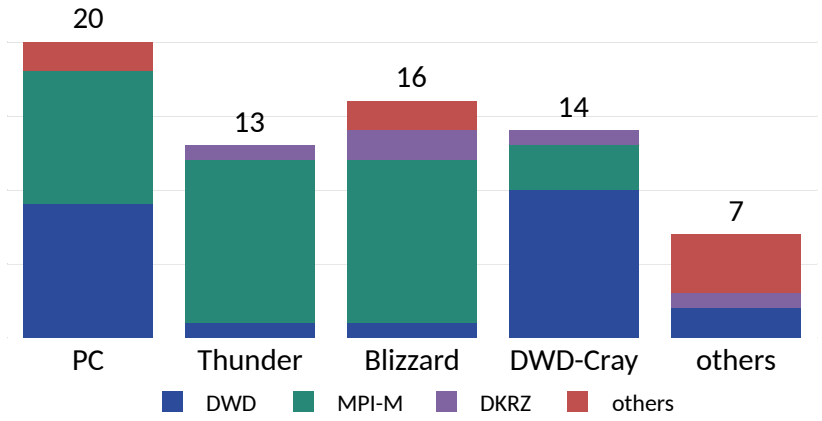
B. Personal Expectations on Checked Out/Downloaded ICON Code

10. What *do you expect* from freshly checked out/downloaded ICON code?

a) Code must compile on/with...



b) Model must run on...

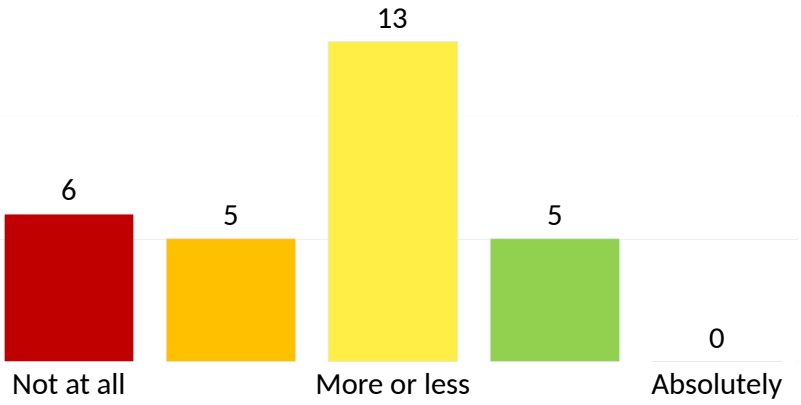


c) Experiments that must be able to run:

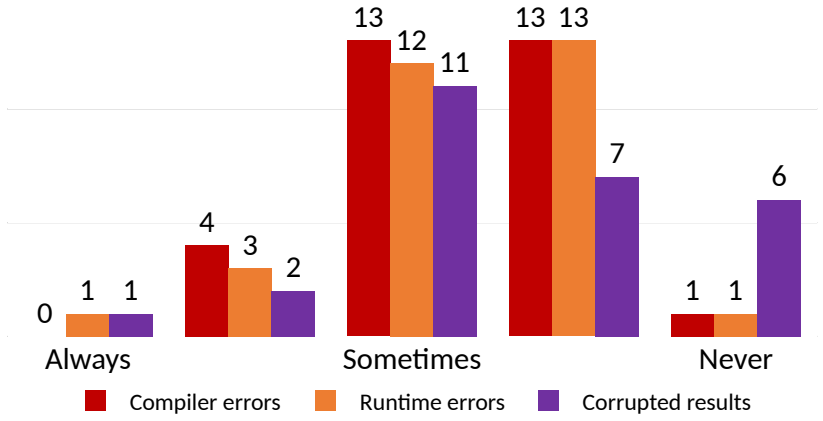
- NWP runs
- Nonhydrostatic with NWP and LES, global / local area mode with different resolutions, few days simulation time, restarts, MPI
- restart, parallelization, short regression test (update); future tests should include module tests for sensibly small functional units

- limited area experiments with high resolution, at least 416m and ICON-LES model needed for HD(CP)^2 project.
- NH-core + NWP physics, R2B4, R3B7
- R2B4/5/6/7/8 // global // 10days, 400days
- nonhydrostatic dynamical core // MPI-OpenMP hybrid execution // Asynchronous Output and restart checkpointing // Real-Data Initialization
- Theoretically, all configurations should be tested.
- (can't say yet)
- non-hydrostatic, NWP, R2B3 Nest npto [?] R2B11
- NH-dynamical core, NWP-physics
- 100m-1km grid resolution with ICON-LES with restart
- NWP setup
- - Standard test cases // - Short episodes in NWP and/or AMIP mode
- * NH-core with NWP-physics + restart // * idealized testcases (Jablonowski, transport) // * hybrid, MPI-only, OpenMP-only
- NWPsetup [unleserlich] nesting, restart, asynchronous output, domain merging, processor splitting
- Dycore, parallel execution, reproducibility on different configuration
- Must pass all defined tests!
- MPI or better Hybrid-Parallel version, NWP-part [?], plus ART modules (volcanic ash, mineral radionuclides dust -> in future), R2B4-6
- All automatic test should work. I mostly concentrate on ocean test, i.e. exp*oce* files. // The weak part of the story is the tests do not cover the model.
- AMIP, nh dycore, ECHAM, R2B4 // Results insensitive to changes in number of restarts, MPI processes, OpenMP threads, and changes in nproma
- coupled aquaplanet configuration: hydrostatic atmosphere coupled to ocean
- 1-2 km grid resolution with boundary layers prefetching
- hydrostatic and non-hydrostatic core, ECHAM physics // AMIP experiments // R2B4 [?]
- NH Dycore + LES physics
- All
- OMIP type experiments // Bunch of simplified experiments (basin tests, shallow water tests)
- All buildbot tests, AMIP, OMIP
- (currently) AMIP with non-hydrostatic core in R2B4, $\approx$ 10 years
- RCE (64x64; 120 km), AMIP is priority (r2b4)
- ocean // hydrostatic atmosphere // non-hydrostatic atmosphere // R2B4 10days + 10years // identical restart test (1d + 1d = 2d) // mpi-parallel

11. Do you think the current test practice in ICON meets these expectations?



12. How often do errors occur in freshly checked out/downloaded ICON versions?



13. From your perspective, what should *always* be tested before new code is committed to the trunk?

- It should be tested if the code compiles, and a test run should be made.
- Compilation on different architectures and with different compilers, bit identical results with different domain decomposition, different nproma settings
- * Build // * Unit tests // * Functional test of the change in question
- The performance of the code in terms of wall clock and memory usage
- Unified configuration on own platform with operational compiler. Keeping bitwise identical results when no change on physics was made.
- compilation on one platform // any simple run testing your change
- Correct compilation/linking - OpenMP pragmas included.
  Perhaps a very small test completing the initialization and entering the time loop, in order to detect logical errors in the control flow.
- Everything should be tested. However, for this new faster tests are required.
- there should be no compiler errors // some kinds of unit tests
- (assuming trunk means the stable branch discussed at the meeting) // compilation on all relevant platforms // L1 and unit tests as available // L2 tests // additional tests to stress interplay between components (dycore, atmosphere, ocean, IO, parallel infrastructure. . .)
- Compile + run on super computer and PC
- That the code compiles, produces identical results with various MPI tasks.
- Compilation
- Standard test cases
- - does the code compile? (at least on one of the following compilers: cray, intel, gfortran) // - if you know, that your changes should change the results, it must be checked/verified before checkin!
- correctness of syntax; correct execution of basic [?] tests; if new options are added, check that existing [?] functionality remain unchanged
- Complete set of unit tests. If tests are not passed, developer needs to document why, and what will be undertaken to correct it
- - Compile // - Link // - Initial // - Initial + restart // - model component // - Unit tests for base infrastructure (mtime, support, output control)
- Compilation on Cray, Run with Debug option, e.g. array bounds check
- 1. Compile test with NAG+GCC/GFORTRAN // 2. runtime test with low resolution ocean and atmosphere experiment (only some time steps) // 3.
- must compile

- coupled aquaplanet configuration (atmosphere coupled to ocean) as this test is very sensitive to programming errors.
- The behaviour of code on performance and whether expected results are gained. Impact on other interfaces of code due to my implementation.
- Compilation with at least two compilers on linux and blizzard. // Model still runs.
- Technical tests, compile
- A predefined set of experiments in particular including a metric for evaluating the experimental result from a physical perspective. The definition of the metric is nontrivial.
- Compiling with nag, running of some steps at least in the model part that has been changed (atmos, oce,..)
- The BuildBot experiments at least should run, with changes to results noted in the commit message.
- Code should compile with at least one compiler on one platform
- for RCE, AMIP // update, nproma, parallel (MPI, openmp), restart test
- compile, run for 2 timesteps, run at least 2-3 key-runscripts (or full buildbot)

14. What tests should be run nightly by a build system?

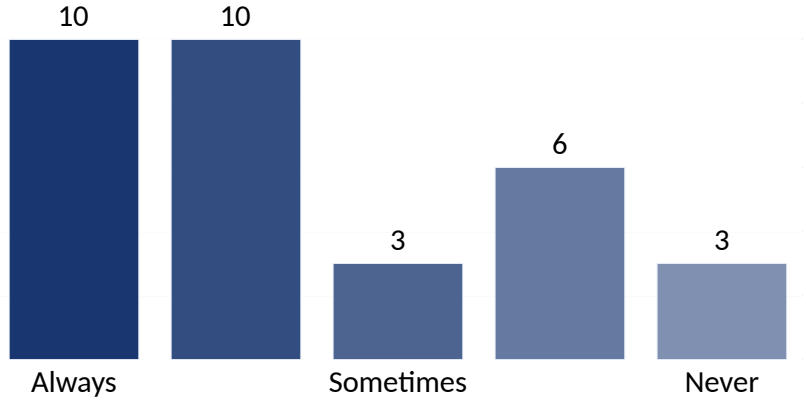
- different domain configuration, several settings for namelist parameters
- at least same as answer 13
- All of Q 10, plus all short science tests, monitoring of compute throughput
- - build tests to see if the code compiles properly on other machines // - Performance tests to compare the performance with previous runs and detect possible sources of performance degradation with smaller tests
- Different configurations and compiler
- a hand full setups including realistic global NWP
- From my perspective (a test may address more than one of the following points): // - At least one nonhydrostatic simulation with real-data // - At least one hybrid-parallel setup // - At least one comparison between restarted and non-restarted simulation // - At least one setup with asynchronous I/O modules (to detect MPI deadlocks) // - Compilation without Ocean, testbed etc. // - Setup with nested grids which are also turned on and off during simulation
- Everything.
- unit tests // low res, small scaling experiments
- compilation on all relevant platforms // unit tests (as available) // L1 if possible
- Compile, run, restart in set of configurations
- I am not aware of various tests in the build system, but I am happy with what run now.
- dito
- - compilation on all available machines // - restart-test for NWP-setup // - a couple of simple idealized testcases which check individual features of the code
- a predefined test suite [unleserlich] of the code branches as possible while running with low computational cost
- Nightly unit tests and regression tests.
- Build (compile & link)
- 1. Compile on all platforms // 2. Short runtime tests (many) // 3. Small number of test, which cover the complete IO of all components
- - portability // - restart // - MPI // - OpenMP // - nproma
- “compiler tests” and short runs with restart in between
- Large runs requiring several hours of simulations.
- Technical tests (nproma, parallel, restart) on all compilers and machines. // Simple comparison of testcases to reference simulation.
- - Compilation on most compilers // - Restart // - Parallelization/OpenMP
- As 13
- technical tests: compiling, restart, I/O, ptest
- tests like already done using buildbot
- - Compiling on different platforms // - short and simple tests (e.g. baroclinic wave test)
- tests of 13. for a very restricted number of model config. over a few time steps.
- technical test; parallel test; short forecast or simulation runs (5-100 timesteps)

15. What tests should be run regularly but less frequently?

- NWP forecast
- longer simulation times
- Longer reference runs, parallel performance scaling, monitoring of I/O throughput
- Performance tests with big runs on at least 1024 cores with high resolution experiments
- Stability tests on longer simulation times and test of closures (e.g. water budget)
- 400 days, both NWP and ECHAM physics and verification runs with plots
- - Compilation process generating a list on unused use dependencies, unused variables combined with an automatic SVN request for these lines, s.t authors can be directly addressed.
- - Extended tests of everything // - Performance tests
- higher res, higher scaling tests
- L2 and the additional tests mentioned in 13
- Default simulations, eg. set of forecasts to check forecast skill.
- Usually tests which check the dynamical core are run occasionally, i.e. when there is change in the kernel. Other tests should be run frequently.
- - more expensive tests (including nests, starting/stopping nests during runtime, parallelization checks, multiple nest setups)
- Tests of NWP forecast scores (two-month test suite with subsequent verification)
- Advanced system tests, eg. ICON dycore+physics, ICON coupled (in future) etc.
- - Initial // - Initial + restart // - model component // - AMIP // - OMIP // - parallel NWP routine // - coupled CMIP5 conto [?] / pre-release // regular (biweekly) and [unleserlich] on demand
- Real NWP test case, with nest
- [ocean] long simulation which show physical results (rather than technical ones), i.e. 500y runs [weekly]
- full experiments
- 10-20yr AMPI type simulation // 100yr OMPI type simulation
- Scaling tests or performance test due to new implementation. So some resources should be available to others and not all are taken by few tests.
- Test full experiments or forecasts
- Limited Area Runs // Operational Runs like AMIP, NWP
- long runs
- See answer 13
- longer OMIP/AMIP experiments, at least when results are not identical/bit-identical anymore
- - mass & energy conservation // - climate (at least 10 years) of AMIP simulation
- longer test over one year or years for checking scientific results
- ocean - 2-3 keypoint runscripts for 50-150 years every week; data of last year or decade + plots archived

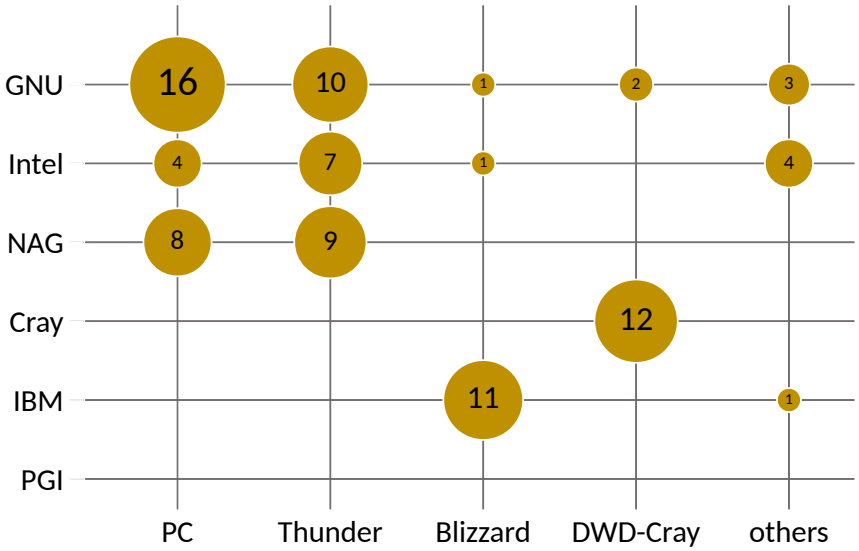
C. Personal Practice

16. How often do you work with SVN branches?

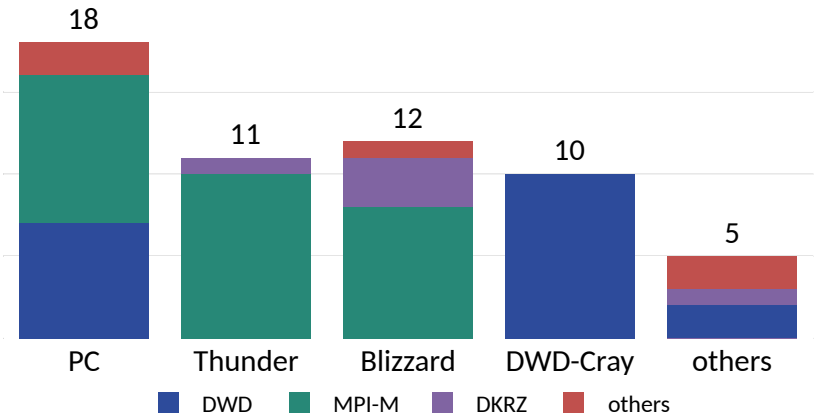


17. What *do you* test before committing to the trunk?

a) Code compiles on/with...



b) Model runs on...



c) Experiments that run:

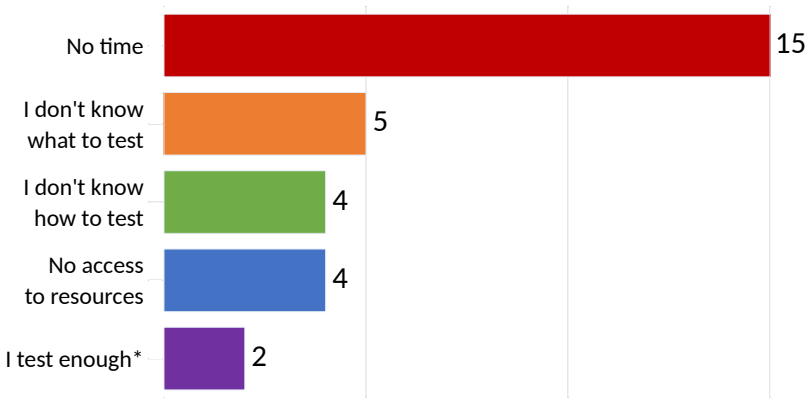
- short NWP forecast at lower resoultion and I look at some fields
- non-hydrostatioc core, NWP LES, global / local area mode, MPI
- restart, parallelization, short regression test (update)
- high resolution local area experiments used in HD(CP)² project with ICON-LES
- NH-core + NWP physics on winter and summer cases. // Keeping bitwise identical results when no change on physics was made.
- 60 forecasts of 10 days each for Jan/Jul 2012
- - Several variants of a small real-data test with the new functionality // disabled/enabled. // + "Checksuite": // - Small setup with nests // - The same small setup restarted
- nat_ape_r2b4 and r2b5 // some lam setup for hdcp2
- NH-Dyn, NWP-physics
- 100m-1km grid resolution with ICON-LES and NWP physics.
- - ape test case (nh, hs) // - NWP run
- - global NWP setup starting from IFS-analysis (often) // - global run (NWP) with restart (sometimes) // In general chosen experiments strongly depend on the introduced code changes ⇒ no static test suite
- situation-dependent: if results remain remain bit-identical, one test running through the modified code is sufficient; for changes in physics schemes, a two-month [unleserlich] suite with verification is executed
- dycore only
- Predefined test suite
- siehe 10
- Depending on the changeset I test // (1) ocean only experiments // (2) some atmosphere experiments, that are run by buildbot
- As 10
- ECHAM Physics, 1-2 km resolution, 3 mins, off. // Most experiments run successfully but if the grid has a resolution less than 500m than problems may occur.
- AMIP non-hydrostatic ECHAM physics r2b4 // one month with restart
- NH Core + LES physics // + NWP physics
- Ocean OMIP, Box
- Depending on the problem/change done: // short technical tests (e.g. testing of Sebastian Rast), longer simulations.
- Full set of BuildBot experiments

- (currently) AMIP type simulation, non-hydrostatic core, ECHAM package, R2B4, // depending on the change I did in the code: a few time steps – 20 years (incl. restart, ≈ 100 cpus)
- amip r2b4 // rce 64x64, 120 km
- ocean only: test_parallel ; test_numerics; oce_default // R2B4, R2B2, restart test, 10-100 timesteps // long run oce_mpiom > 100 y

18. Further tests that you run regularly from time to time:

- High resolution tests, longer forecasts
- climate runs 400 days
- Set of forecast to check forecast skill
- The test from point 17 is sufficient for my purposes.
- - idealized tracer tests (tracer only)
- [unleserlich]-month NWS test suite @[unleserlich]km with verification; less frequently with 13km
- -valgrind runs
- Performance of new turbulence code
- 500 year ocean experiments, mostly over the weekend
- our own testsuite for the coupling library currently under development.
- Tests for different grid resolution – with different namelist options.
- longer AMIP experiments on thunder
- Restart, paralelization
- OMIP
- don‘t know yet
- ocean: Atlantic box (AquaAtlanticBox_[unleserlich]79km) oce_mpiom) // buildbot – for full compiler/parallel test (still difficult and uncomfortable to use)

19. What keeps you from testing more?



* Die letzte Option wurde von zwei Teilnehmenden eigenständig hinzugefügt.

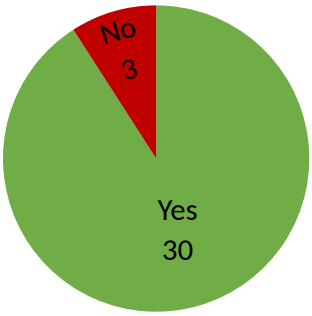
Antworten im zusätzlichen Freitextfeld:

- Just recently got a buildbot account, so tests on different platforms may be easier to perform now. // Sometimes I find it difficult to find out, which parts of the code have to be tested / which setups are suitable for testing which part of the code.
- So far, I mainly did changes for buildbot, doxygen
- I am interested in scaling tests on a high core number (more than 1024). Getting resources for this purpose is not possible very often and needs time for planning the tests.

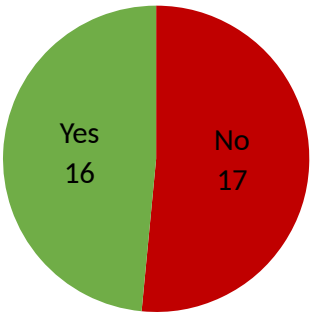
- climate runs 400 days
- Complex setup: // - existing scripts are user specific // - missing real-case data // - time-consuming runs // Automatic testing (Buildbot) // - slow response // - complicated scripts
- Takes too long. // When using buildbot for testing, I always have to create an SVN-branch that I have to link to my local GIT repository.
- My commits are usually to the ART trunk. Changes in the icon trunk are very seldom.
- stage of develop. too early to perform meaningful tests
- personal feeling: tested enough
- belief that tests have been sufficient
- Currently only working on dycore. Other tests should be performed centrally on trunk.
- Own changes don't affect deeper code structure
- I hardly know how to test atmosphere. Just run test, output test and restart test are possible for me. Otherwise I need help (which is ok for a guy from the ocean or infrastructure).
- Nothing
- No point
- Definition of a suitable test set is difficult, see 13
- Most parts of the code operate out of my area of expertise
- don't know yet
- unit tests would be nice, but I am not familiar with
- Errors that frequently occur in fast ocean development cycles

D. Unit Tests

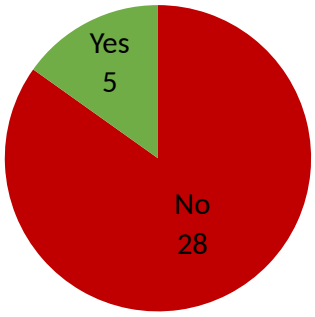
20. Have you ever heard about *Unit Testing* before?



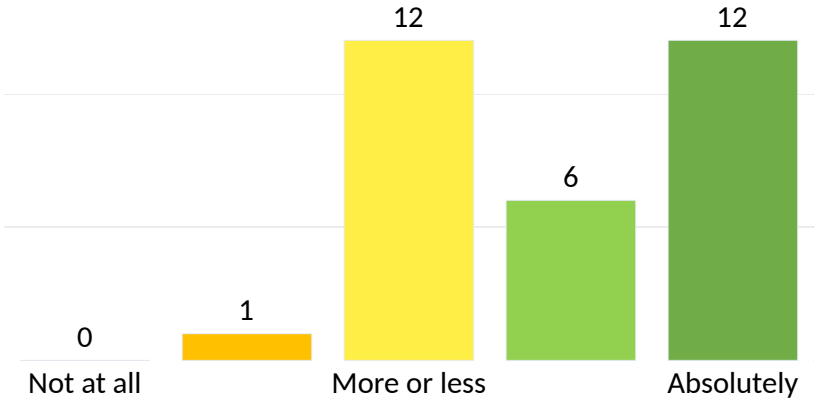
21. Have you ever worked with Unit Tests?



22. Have you ever worked with Unit Tests in ICON?



23. Do you think Unit Tests can be useful for ICON?



24. What do you think are the main problems with Unit Tests for ICON?

- It might be useful to test IO, physics packages. I think it might be difficult providing useful input and verifying data for different resolutions.
- the "historical grown" code may cause problems, including some connections between different code parts. It may be difficult to test all branches and combinations of switches (in for case constructs) in such a complex system. Sometimes it may be not clear what the "real" result of some code part has to be → "fuzzy testing"?
- Identification/separation of sensibly small functional units with sufficiently defined behaviour
- At the moment I am not sure whether these tests will cover problems arising due the combination and interaction of several modules, subroutines etc. in ICON
- - Unit testing framework polluting the code // - Huge amount of input data required for relevant setups (mostly parallel execution)
- ICON is not build in a way that you could easily test parts of it independently. Most modules have a huge number of dependencies with other modules. Many of these dependencies are not obvious.
- - someone has to define and write the tests, provide input data and validity checks for the output // - most guys are only interested in physical quantities or just some "good" physical behaviour
- - the lack of them! // - assuming they will exist more and more: the ease of use, also through automated systems like buildbot, and a thorough documentation // - ability to seamlessly select individual tests or entire suits of them // - well organized and understandable infrastructure to keep all relevant input/ output / comparison files for tests which require such

- Clearly, it's in the development phase and not yet clear to me how I can really benefit from it. The reason being simply that I hava sufficient check-tools for my development.
- Seperation of a working environment
- - time // - automation
- - Simplicity of usage // - Psychological: convicing/forcing developers to employ them and correct code if they fail.
- Selection of the right components to be tested. It should focus on the infrastructure components which should be moved into "external" libraries.
- 1. ICON is not designed to be "unit"-tested. Esp. the use of public module variables makes it hard to guess the correct initialization for a unit test of a module. // 2. Scientific questions aren't easy to checked that way, because there might not be a yes/no answer to it. // 3. The Testbed is a great tool for writing unit-tests, because the default initialization is used.
- The number of units is large.
- People are not aware of its benefits and their lack of personal experience how this can speed up the development.
- I think a common following were other people (developers) of talk about it and, the influences you to use it.
- Defining unit tests for often very long and complex functions/subroutines
- The complicated nature of ICON + no proper guidance
- Unit Tests would require some effort as the code is not perfectly modular. // functioning of the model depends on interplay of the results of many routines (e.g. Concerning numerical stability)
- unit tests would be nice, but I am not familiar with
- - no time to develop meaningful tests // - too fast development, too less time attention to testing

E. Further Comments

- As I have not done much ICON coding yet, some answers are given from work experience with other models
- On B.10.: I only work with revision that successfully completed all buildbot tests on all machines. // On C.16.+17.: I use git and make extensive use of branches. I only create SVN-branches when I have completed a task and want to run all buildbot tests on all commits related to this task before commiting them to the trunk.
- I am firmly against a loose scheme to check into the trunk. Either: a) A gatekeeeper tests/checks in changes, or b) developers do rigorous testing before checking in.
- Parallel testing internals for MPI and OpenMP. Developed in 1998 for ECHAM for testing parallelization correctnes.
- Viele nervige Fehler wären vermeidbar, wenn die Kommunikation besser wäre. Änderungen sollten mit mehr Leuten besprochen werden, bevor sie committed werden. Oft fehlt ein Design-Phase, an der mehre Entwickler beteiligt sind.
- My answers in section C are related to my work with ECHAM.
- In the last 12 month I'vw hardly worked with ICON. In the 12 month before, I implemented the land model JSBACH in ICON. Soon, I will start to work with ICON again. Comments marked with (currently) are my expectations, what I will do in the near future.

F. War Stories

Describe the hardest bug in ICON and the way to find and fix it as detailed as possible (modules, routines, variable names and/or ticket numbers are welcome):

- Adaption of the COSMO version to ICON structure, while keeping bitwise identical results when change dimensions in arrays. On a vector compiler it is a mess. Releases r7336 and r7374,
- Mich stören am meisten Bugs im Compiler und den Libraries. Nur als Stichwort: Neulich habe ich folgenden (bereits bekannten aber noch nicht behobenen Bug in der MPI-Implementierung von Intel weitergegeben. Bei bestimmten MPI-Task-Anzahlen liefert die MPI_IN_PLACE-Variante des Befehls MPI_ALLREDUCE falsche Resultate.
  Gemerkt hat man das dann in einer fehlerhaften Gebietszerlegung der ICON-Interpolations-Utilities
- One of the many thing that happened:
  When refactoring a part of the code we were very happy that all small tests, which we ran every few minutes were successful. At some point it was obvious that there should have been an bug (statistically). We then noticed that our code did not run in the tests. Instead a nearly identical piece of code was executed, that had only minor functional differences.
  Afterwards, it took us a couple of days to remove most of the duplicated code in the domain decomposition. We removed many hundreds of lines of code. . . (e.g. r13256, r15355, r15355)
- openMPI bug in a routine which only caused problems after 24h of simulation time and only on PC.
- I can not really classify any bug I had in ICON as a particalary hard. Perhaps, the most annoying bug is with the flag p_test_run.
  After the code runs sufficiently good, the final test includes the run with this flag, which will tell you if there are some sync-problems with MPI communication.
- I had to isolate a bug in the Cray compiler for GPU (OpenACC) which manifested itself in [unleserliches] small indexing offsets into fields (i became i+1) resulting in small but $>$ roundoff errors. Was only possible to locate with a unit test comparing GPU vs. CPU results.
- - Not in ICON but in ECHAM: NEC changed the MPI library without notice and the library introduced a runtime error. As we did use an mpi compiler wrapper we did not know the library version. . .
  - Learning that compilers have bugs. . .
- Hardes bug [unleserliches] be parallel run with segmentation errors, inconsistent results.
- The problem are not “bugs” in the common sense but wrong usage of subroutines, wrong structure of icon (spoiled operator splitting, inconsistent time stepping). Consequence: results are rubbish.
  But: these are conceptual problems, not just “bugs”.
  In order to solve these:
  - organisation of code and people
  - description what does what
  - test units for scientific outcome
  I cannot describe a “specific bug”, but still echam physics is not proven to work. There is no “method” how to prove that the parts do what is expected. That‘s bad and a conceptual problem.
- 3844 tracer conversion violated
  from 5.8. to 2.10.2013 4 people involved

Anhang B

Fragenkatalog zur Teststudie 2017

Im Vorwege der Besuche am GFDL, bei JAMSTEC und am R-CCS wurde der folgende Fragenkatalog an die teilnehmenden Teams versendet. Dieser stellte keinen festen Fragebogen dar, sondern diente vielmehr als Diskussionsgrundlage während der Interviews. Nicht immer wurden alle Themen (gleichermaßen) besprochen.

Goals

- To learn about general testing practices and specific testing methods at several climate modeling centers
- To find out about similarities and differences
- To create a terminology of testing specific for the climate modeling domain
- NOT: To rate the maturity of testing practices

Benefits for the participating modeling centers:

- A more systematic view on the own practices
- To learn from each other

Outcome:

- A collection of interviews with developers of at least three different models
- A joint publication of a study on testing practices in the climate modeling community, also including the definition of a terminology

Target

CMIP6 models

- in active development
- with user base beyond the core developers
- candidates:
  - ICON (Germany)
  - MIROC (Japan) – F77 code
  - NICAM (Japan)
  - Dynamico (France)
  - GFDL models (USA)

Questions

General

- Model name?
- Is there a reference paper which can be cited when referring to the model?
- How long in development/use?
- Who are the core developers? And how many? What is the background/previous knowledge of the developers?
- Who supports the developer team?
- Who are the "users"? What is the background/previous knowledge of the users?
- Who provides support for the users? What kind of support?

Software Architecture

- Size of the model in Lines of Code (KLOC)
- Which high-level components are part of the model? (Submodels, Coupler, . . .)
- Used libraries?
- How many binaries are produced?
- Execution platform / target supercomputer
- Important Technologies (Compiler (vendor/version), Vectorisation, MPI, OpenMP, OpenACC, . . .)?
- How is the separation of NWP and climate research reflected by the architecture? Which parts are NWP only/climate only, which are shared?
- Integrated Performance Engineering?

Development Process

- Structure of the developer team(s)
- Are they following a specific (classic or agile) development method?
- What Tools are used (for version control, issue tracking, continuous integration. . .)?
- What kind of documents are produced (once and routinely)
- Code Structure
  - Which code branches exist?
  - Who is responsible for merging the branches?
  - How is decided which features are integrated into the main branch?
  - What are the formal criteria for code to be merged into the main branch?
- What is driving the development? Who is doing the strategic planning?
- How does the release process look like?
  - How many, how often releases?
  - Who is responsible?
  - What are the formal criteria for a version to be released?

  - Who is using official releases?
- How is the separation between NWP and climate reflected by the organizational structure?

Testing

Model Validation (once and/or routinely)

- Is there a formal quality assurance process established?
- Participation in CMIP?
- Comparison with observational data?
- Measuring of NWP forecast skill?
- Ensemble runs?
- Idealistic scenarios?
- Some standard experiments?
- Anything else?
- Any interesting papers about the model's evaluation?

Technical Tests/Regression Tests

- Are there test suites running regularly (e.g. daily)?
- What is included?
  - Standard experiments?
  - Important configurations (resolution, time-step, include submodels, ...)?
  - Parallel vs. serial?
  - Checkpoint/Restart?
  - Several platforms/compiler?
  - Performance tests?
  - Anything else?
- What tools are used?

- What tests are executed when merging new code to the main branch?
- What tests are executed for a release?

Developer Tests

- How do individual developers try out their code changes/additions?
- How does debugging generally look like?
- What tools are available for the developer? What tools are used?
- Do developers have access to important test suites?
- Do developers have access to all relevant platforms/compilers?
- Is there something like unit tests?
- If yes, how do they look like?
- If no, why not? What do you see as they main obstacles?

General discussion

- War stories - . . . about especially mean bugs.
- What makes you believe that the model is "doing the right thing"?
- Do you care whether the model validates fine because it is programmed correctly or it does what does just by chance? For example, what if an algorithm is wrong by design but one systematic error in the code makes it performing well in the validation? Would that be just as good as you have done everything correctly just from the beginning?

Anhang C

Benutzung von FortranCallGraph

Im Folgenden wird die Benutzungsschnittstelle des Softwarewerkzeugs FortranCallGraph (fcg) beschrieben.

C.1 Konfigurationsdatei

Vor der Benutzung von fcg sind in der Konfigurationsdatei config_fortrancallgraph.py grundlegende Einstellungen bezogen auf die Zielanwendung einzutragen. Sie enthält die folgenden Konfigurationsvariablen:

CACHE_DIR Pfad des Verzeichnisses, in dem serialisierte Aufrufgraphen, für eine spätere Wiederverwendung abgelegt werden, siehe auch Abschnitt 9.4.4 (optional)

ASSEMBLER_DIRS Liste der Verzeichnisse, in denen sich die erzeugten Assemblerdateien der Zielanwendung befinden.

SOURCE_DIRS Liste der Verzeichnisse, in denen sich die Quellcodedateien der Zielanwendung befinden.

SOURCE_FILES_PREPROCESSED Boolean, der angibt, ob die in `SOURCE_DIRS` befindlichen Dateien bereits vom Präprozessor verarbeitet wurden (optional, Standard: `False`).

SPECIAL_MODULE_FILES Dictionary, in dem Module, die sich in Quellcodedateien befinden, deren Name nicht dem Modulnamen entspricht, dem tatsächlichen Dateinamen zugeordnet werden (optional). Diese Zuordnung könnte auch automatisch geschehen, da dies in der aktuellen Version von fcg jedoch noch nicht implementiert ist, ist die manuelle Zuordnung noch notwendig (optional).

EXCLUDE_MODULES Liste von Modulen, die vollständig von der Analyse ausgeschlossen werden sollen (optional).

IGNORE_GLOBALS_FROM_MODULES Liste von Modulen, deren MODULVARIABLEN nicht analysiert werden sollen (optional).

IGNORE_DERIVED_TYPES Liste von Verbunddatentypen, deren KOMPONENTEN nicht analysiert werden sollen (optional).

ALWAYS_FULL_TYPES Liste von Verbunddatentypen, deren KOMPONENTEN pauschal als verwendet gelten sollen (optional).

ABSTRACT_TYPE_IMPLEMENTATIONS Dictionary, in dem abstrakte Typen konkreten Typen zugeordnet werden, die an ihrer Stelle von der Analyse behandelt werden sollen, siehe auch Abschnitt 10.2.2 (optional).

C.2 Kommandozeilenschnittstelle

Im Folgenden ist die englische Beschreibung der Kommandozeilenschnittstelle aufgeführt, wie sie beim Aufruf von `./FortranCallGraph.py -help` erscheint:

```
usage: FortranCallGraph.py [-h]
                           (-p {tree,dot,list-subroutines,list-modules,list-
                              ↪ files} | -a {arguments,result,globals,all} | -
                              ↪ d {lines,statements} | -l {first,last,doc,
                              ↪ specs,use,contains,all} | -u {modules,files})
                           [-v VARIABLE] [-ml MAXLEVEL] [-po] [-ln] [-cc] [-q]
                           [-i IGNORE] [-cf CONFIGFILE]
                           module [subroutine]

Print or analyse a subroutine's call graph.

positional arguments:
  module                Module name
  subroutine            Subroutine or function name

optional arguments:
  -h, --help            show this help message and exit
  -p {tree,dot,list-subroutines,list-modules,list-files}, --printer {tree,dot,
     ↪ list-subroutines,list-modules,list-files}
                        Print the callgraph (tree: in a tree-like form, dot:
                        in DOT format for Graphviz, list-subroutines: only
                        list subroutines, list-modules: only list modules
                        containing subroutines from the call graph, list-
                        files: only list files containing subroutines from the
                        call graph).
  -a {arguments,result,globals,all}, --analysis {arguments,result,globals,all}
                        Analyze variable usage (arguments: only subroutine
                        arguments, result: only function result, globals: only
                        module variables, all: all together).
  -d {lines,statements}, --dump {lines,statements}
                        Dump subroutine or module source code (lines: original
                        source lines, statements: normalized source lines).
                        When no subroutine is given, the whole module is
                        dumped.
```

```
-l {first,last,doc,specs,use,contains,all}, --line {first,last,doc,specs,use,
    ↪ contains,all}
                        Show some interesting source lines of the subroutine
                        (first: the first line, containing the
                        SUBROUTINE/FUNCTION keyword, last: the last line,
                        containing the END keyword, doc: the first line of the
                        leading comment - the same as "first" when no comment
                        exists, specs: the last variable specification, use:
                        the last USE statement, contains: the CONTAINS
                        statement - -1 when there is no such statement, all:
                        all of the others).
-u {modules,files}, --use {modules,files}
                        Prints use dependencies of a subroutine (modules:
                        module names, files: file pathes).
-v VARIABLE, --variable VARIABLE
                        Restrict the analysis to the given variable which has
                        to be a subroutine argument and of a derived type.
                        Applicable with -a arguments.
-ml MAXLEVEL, --maxLevel MAXLEVEL
                        Limits depth of callgraph output. Applicable with -p.
-po, --pointersOnly     Limit result output to pointer variables. Applicable
                        with -a.
-ln, --lineNumbers      Add line numbers to the output. Applicable with -d.
-cc, --clearCache       Create a new call graph instead of using a cached one.
                        Applicable with -p or -a.
-q, --quiet             Reduce the output. Applicable with -a and -l.
-i IGNORE, --ignore IGNORE
                        Leave out subroutines matching a given regular
                        expression. Applicable with -p and -a.
-cf CONFIGFILE, --configFile CONFIGFILE
                        Import configuration from this file.
```

Die wichtigsten Optionen sind `-p` für die (Erzeugung und) Ausgabe des Aufrufgraphen sowie `-a` für die Analyse der Variablenverwendung. Zudem lässt sich mit `-d` der Quelltext eines Moduls oder einer Prozedur, mit `-l` die Zeilennummern von wichtigen Stellen innerhalb einer Prozedur und mit `-u` die USE-Abhängigkeiten einer Prozedur bzw. ihres Moduls ausgeben. Eine dieser fünf Optionen muss ausgewählt werden.

Anhang D

FortranTestGenerator

Im Folgenden wird die Benutzungsschnittstelle des Softwarewerkzeugs FortranTestGenerator (FTG) beschrieben.

D.1 Konfigurationsdatei

Vor der Benutzung von fcg sind in der Konfigurationsdatei config_fortrantestgenerator.py grundlegende Einstellungen bezogen auf die Zielanwendung einzutragen. Sie enthält die folgenden Konfigurationsvariablen:

FCG_DIR Pfad des Verzeichnisses, in dem sich fcg befindet.

FCG_CONFIG_FILE Pfad der zu verwendenden fcg-Konfigurationsdatei (optional).

TEMPLATE Pfad der Hauptdatei des zu verwendenden Templates, siehe auch Anhang D.3.1.

TEST_SOURCE_DIR Pfad des Verzeichnisses, in dem die Quellcodedatei des generierten Testprogramms abgelegt wird.

MODIFY_SOURCE_DIRS Liste von Verzeichnissen, in denen sich die Quellcodedateien der Zielanwendung befinden, die von FTG instrumentalisiert werden sollen, sofern dies nicht dieselben sind, die von fcg analysiert werden (optional).

BACKUP_SUFFIX Dateiendung der Kopien, die von den Quellcodedateien erstellt werden, bevor diese instrumentalisiert werden (optional, Standard: `ftg-backup`).

FTG_PREFIX Prefix der vom Template erzeugten Prozeduren, die von wiederholten Analysen ausgenommen werden sollen (optional, Standard: `ftg_`).

TEST_DATA_BASE_DIR Pfad des Verzeichnisses, an dem Testdaten abgelegt werden sollen. Kann im Template über die Variable `$dataDir` verwendet werden (optional, Standard: `.`).

Zusätzlich zu der FTG-eigenen Konfiguration muss das von FTG verwendete fcg konfiguriert werden. Die hierfür benötigten Variablen werden aus der Datei config_fortrancallgraph.py im fcg-Verzeichnis ausgelesen (siehe auch Anhang C.1). Alternativ können diese auch in config_fortrantestgenerator.py gesetzt werden. Ebenso können auch einzelne Variablen aus config_fortrancallgraph.py in config_fortrantestgenerator.py überschrieben werden.

D.2 Kommandozeilenschnittstelle

Im Folgenden ist die englische Beschreibung der Kommandozeilenschnittstelle aufgeführt, wie sie beim Aufruf von `./FortranTestGenerator.py -help` erscheint:

```
usage: FortranTestGenerator.py [-h] [-a] [-b] [-c] [-e] [-r] [-m] [-cc]
                               [-cf CONFIGFILE]
                               [module] [subroutine]

Generate test code.

positional arguments:
  module                Module name
  subroutine            Subroutine or function name

optional arguments:
  -h, --help            show this help message and exit
  -a, --restoreCapture  Restore only Capture Backup Files
  -b, --restore         Restore all Backup Files
  -c, --capture         Generate Capture Code
  -e, --export          Generate Export Code
  -r, --replay          Generate Replay Code
  -m, --measure         Measure Time
  -cc, --clearCache     Create a new call graph instead of using a cached one.
  -cf CONFIGFILE, --configFile CONFIGFILE
                        Import configuration from this file.
```

Die wichtigsten Optionen sind `-c` für die Erzeugung des Capturecodes und die Instrumentalisierung der Zielanwendung sowie `-r` für die Generierung des Testprogramms. Der benötigte Exportcode wird in beiden Fällen mitgeneriert und die entsprechenden Moduldateien instrumentalisiert. Er lässt sich auch separat mit `-e` erzeugen. Zudem lässt sich mit `-a` existierender Capturecode sowie mit `-b` Capture- und Exportcode entfernen. Eine dieser fünf Optionen muss ausgewählt werden, eine Kombination ist möglich.

D.3 Template BaseCompare

In den folgenden Abschnitten wird das vollständige in Abschnitt 9.6.5 eingeführte FTG-Template BaseCompare erläutert. Zum besseren Verständnis sei zunächst jedoch die allgemeine Oberklasse `FTGTemplate`, auf der jedes Template basiert, dargestellt (siehe auch Abschnitt 9.6.2):

FTGTemplate.tmpl

```
#import os
#def captureAfterUse
#end def captureAfterUse
##
#def captureBeforeContains
#end def captureBeforeContains
##
#def captureAfterLastSpecification
#end def captureAfterLastSpecification
##
#def captureBeforeEnd
#end def captureBeforeEnd
##
#def captureAfterSubroutine
#end def captureAfterSubroutine
##
#def exportAfterUse
#end def exportAfterUse
##
#def exportBeforeContains
#end def exportBeforeContains
##
#def replay
#end def replay
##
#def include(file)
  #include os.path.dirname(self._CHEETAH_src) + '/../' + file
#end def include
##
#attr part = ''
##
#if $part == 'captureAfterUse'
  $captureAfterUse
#elif $part == 'captureBeforeContains'
  $captureBeforeContains
#elif $part == 'captureAfterLastSpecification'
  $captureAfterLastSpecification
#elif $part == 'captureBeforeEnd'
  $captureBeforeEnd
#elif $part == 'captureAfterSubroutine'
  $captureAfterSubroutine
#elif $part == 'exportAfterUse'
  $exportAfterUse
#elif $part == 'exportBeforeContains'
  $exportBeforeContains
#elif $part == 'replay'
  $replay
#elif not $part
  #raise ValueError('No template part given')
#else
  #raise ValueError('Not a valid template part: ' + str($part))
```

```
52 #end if
```

Im oberen Teil (Zeilen 2–24) sind die vorgegebenen Methoden, die die jeweiligen Codeabschnitte repräsentieren, definiert (siehe auch Abschnitt 9.6.2). Dann folgt eine Methode `include` zur Einbindung anderer Templatedateien, die von den Untertemplates genutzt werden kann. In Zeile 30 wird ein Attribut `part` definiert, welches beim Aufruf des Templates gesetzt wird. Anhand des Wertes von `part`, wird dann im Hauptteil des Templates (ab Zeile 32) entschieden, welcher Codeabschnitt, d.h. welche Methode aufgerufen und ausgegeben wird.

D.3.1 Hauptdatei

Ein Template besteht in der Regel aus einem gleichnamigen Ordner mit einer Hauptdatei, die die Templateklasse, die `FTGTemplate` beerbt, und festlegt, welche Methoden überschrieben werden sollen, enthält. Handelt es sich wie in Fall von `BaseCompare` um eine direkte Unterklasse von `FTGCompare`, werden i.d.R. alle vorgegebenen Methoden überschrieben, es sei denn einige Codeabschnitte sollen bewusst leer bleiben. Daneben können weitere Dateien existieren, die von der Hauptdatei eingebunden werden. Es hat sich zur besseren Übersicht bewährt, den eigentlich Templatecode der einzelnen Codeabschnitte jeweils in eigene Dateien auszulagern.

BaseCompare/BaseCompare.tmpl

```
1  #extends FTGTemplate
2
3  #attr prologue = '! ========== BEGIN FORTRAN TEST GENERATOR (FTG)
       ↪ =========='
4
5  #attr epilogue = '! ========== END FORTRAN TEST GENERATOR (FTG)
       ↪ ============'
6
7  #def captureAfterUse
8    $include('BaseCompare/capture.afteruse.tmpl')
9  #end def captureAfterUse
10
11 #def captureBeforeContains
12   $include('BaseCompare/capture.beforecontains.tmpl')
13 #end def captureBeforeContains
14
15 #def captureAfterLastSpecification
16   $include('BaseCompare/capture.afterlastspecification.tmpl')
17 #end def captureAfterLastSpecification
18
19 #def captureBeforeEnd
20   $include('BaseCompare/capture.beforeend.tmpl')
21 #end def captureBeforeEnd
22
23 #def captureAfterSubroutine
```

```
24    $include('BaseCompare/capture.aftersubroutine.tmpl')
25  #end def captureAfterSubroutine
26
27  #def exportBeforeContains
28    $include('BaseCompare/export.beforecontains.tmpl')
29  #end def exportBeforeContains
30
31  #def replay
32    $include('BaseCompare/replay.tmpl')
33  #end def replay
34
35  #def ftgWrite($var)
36    #set global $ftgWrite_var = $var
37    $include('BaseCompare/capture.ftgWrite.tmpl')
38  #end def ftgWrite
39
40  #def ftgRead($var)
41    #set global $ftgRead_var = $var
42    $include('BaseCompare/replay.ftgRead.tmpl')
43  #end def ftgRead
44
45  #def ftgCompare($var)
46    #set global $ftgCompare_var = $var
47    $include('BaseCompare/replay.ftgCompare.tmpl')
48  #end def ftgCompare
```

In Zeile 1 enthält die Templateklasse durch das Schlüsselwort `#extends` den Verweis auf die Oberklasse `FTGTemplate`. Danach werden zwei Attribute definiert: `$prologue` (Zeile 3) und `$epilogue` (Zeile 5). Diese enthalten Zeichenketten, welche jeweils vor und nach jedem eingefügten Codeabschnitt stehen. In diesem Beispiel handelt es sich um einfache Kommentarzeilen, es könnten hier aber beispielsweise auch Präprozessordirektiven eingesetzt werden.

Auf die Attribut-Definitionen folgen die Definitionen der Codeabschnitt-Methoden (Zeilen 7–33). Die Methode `exportBeforeContains` wird von diesem Template nicht überschrieben, der entsprechende Codeabschnitt bleibt somit leer. Alle sieben überschriebenen Methoden binden jeweils separate Dateien ein, in denen der eigentliche Templatecode der jeweiligen Abschnitte enthalten ist. Es folgen ab Zeile 35 die Definitionen von drei weiteren Methoden, die im (ausgelagerten) Code der anderen Methoden verwendet werden: `ftgWrite`, `ftgRead` und `ftgCompare`.

D.3.2 captureAfterUse

BaseCompare/capture.afteruse.tmpl

```
1 $prologue
2
3 USE mpi
4 USE m_ser_ftg, ONLY: ftg_set_serializer, ftg_set_savepoint, ftg_write,
      ↪ ftg_register_only, ftg_destroy_savepoint, ftg_destroy_serializer
5
6 $epilogue
```

Der Codeabschnitt captureAfterUse wird am Kopf des PUT-Moduls eingefügt, um benötigte Variablen und Prozeduren anderer Module einzubinden, wie hier etwa aus der MPI- und der Serialbox2-Bibliothek (siehe auch Abschnitt 9.6.4). Maskiert wird der Codeabschnitt mit den zuvor in den Variablen `$prologue` und `$epilogue` definierten Zeichenketten (Zeilen 1+6).

Zeile 4 ist länger als die in Fortran erlaubten 134 Zeichen. Der Zeilenumbruch und die richtige Einrückung im generierten Code erfolgt automatisch durch FTG.

D.3.3 captureBeforeContains

BaseCompare/capture.beforecontains.tmpl

```
1 $prologue
2
3 INTEGER, PARAMETER :: ftg_${subroutine.name}_capture_round = 1
4 INTEGER :: ftg_${subroutine.name}_round = 0
5
6 $epilogue
```

Der Codeabschnitt captureBeforeContains wird in das PUT-Modul vor dem `CONTAINS`-Schlüsselwort eingefügt. Er dient im Wesentlichen der Definition von Modulvariablen, die für das Capture benötigt werden. Mit dem BaseCompare-Template wird standardmäßig die erste Ausführung der PUT aufgezeichnet. Dies ist in der Konstante `ftg_${subroutine.name}_capture_round` festgelegt. Wenn nicht die erste, sondern eine andere abzählbare Ausführung aufgezeichnet werden soll, kann der Wert dieser Konstante entsprechend verändert werden - entweder bereits im Template oder im generierten Code.

Die Modulvariable `ftg_${subroutine.name}_round` wird mit jeder Ausführung der PUT um eins erhöht bis sie den Wert von `ftg_${subroutine.name}_capture_round` erreicht hat. Für die Platzhaltervariable `${subroutine.name}` wird bei der Generierung des Codes der Name der PUT eingefügt.

D.3.4 captureAfterLastSpecification

BaseCompare/capture.afterlastspecification.tmpl

```
1 $prologue
2
3 ftg_${subroutine.name}_round = ftg_${subroutine.name}_round + 1
4 IF (ftg_${subroutine.name}_capture_active()) THEN
5   CALL ftg_${subroutine.name}_capture_data($commaList("'input'", $args))
6 END IF
7
8 $epilogue
```

Der Codeabschnitt captureAfterLastSpecification wird in die PUT direkt nach der letzten Variablendeklaration, d.h. vor der ersten Anweisung, eingefügt. Er dient der Aufzeichnung der Eingabevariablen der PUT. Hier wird zunächst die bereits erwähnte Modulvariable `ftg_${subroutine.name}_round`, mit der die Anzahl der Ausführungen gezählt werden, inkrementiert (Zeile 3). Anschließend wird mit Hilfe der Funktion `ftg_${subroutine.name}_capture_active()` geprüft, ob die aktuelle Ausführung der PUT aufgezeichnet werden soll (Zeile 4). Ist dies der Fall, wird die Subroutine für das Aufzeichnen der Eingabedaten aufgerufen (Zeile 5). Die zur FTG Template-API gehörende Funktion `$commaList` erzeugt eine kommaseparierte Ausgabe aller Parameter, in diesem Fall aus der Zeichenkette `'input'` und allen PUT-Dummyargumenten, welche in der API-Variable `$args` enthalten sind.

D.3.5 captureBeforeEnd

BaseCompare/capture.beforeend.tmpl

```
1  $prologue
2
3  IF (ftg_${subroutine.name}_capture_active()) THEN
4    CALL ftg_${subroutine.name}_capture_data($commaList("'output'", $args,
       ↪ $result))
5  END IF
6
7  $epilogue
```

Der Codeabschnitt captureBeforeEnd wird in die PUT vor dem END-Schlüsselwort, d.h. nach der letzten Anweisung, eingefügt. Er dient der Aufzeichnung der Ausgabevariablen der PUT. Analog zum vorangegangenen Abschnitt wird hier zunächst geprüft, ob die aktuelle Ausführung der PUT aufgezeichnet werden soll, um dann ggf. die SUBROUTINE für das Aufzeichnen der Ausgabedaten aufzurufen. `$args.allOut` (Zeile 4) liefert eine Untermenge der PUT-DUMMYARGUMENTE, begrenzt auf die INOUT- und OUT-ARGUMENTE. `$result` liefert die Ergebnisvariable, sofern es sich bei der PUT, um eine FUNKTION handelt, ansonsten den speziellen Python-Wert `None`, welcher von `$commaList` ignoriert wird.

D.3.6 captureAfterSubroutine

BaseCompare/capture.aftersubroutine.tmpl

```
1  $prologue
2
3  LOGICAL FUNCTION ftg_${subroutine.name}_capture_active()
4
5    ftg_${subroutine.name}_capture_active = &
6      ftg_${subroutine.name}_round == ftg_${subroutine.name}_capture_round
7
8  END FUNCTION ftg_${subroutine.name}_capture_active
9
10 ⋮
11
12 SUBROUTINE ftg_${subroutine.name}_capture_data($commaList('stage', $args,
       ↪ $result))
13
14   $globals.imports
15   $types.imports
16
17   CHARACTER(*), INTENT(IN) :: stage
18   $args.specs(intent = 'in', allocatable = False)
19   #if $subroutine.isFunction:
```

```
20      $result.spec(intent = 'in', allocatable = False, optional = True)
21    #end if
22
23    INTEGER :: ftg_d1, ftg_d2, ftg_d3, ftg_d4, ftg_d5, ftg_d6
24    CHARACTER(len=256) :: ftg_name
25
26    CALL ftg_${subroutine.name}_init_serializer(stage)
27
28    $resetRegistrations
29    IF (stage == 'input') THEN
30      ! REQUIRED IN ARGUMENTS
31      #for $var in $args.intentIn.requireds.usedVariables
32        $ftgWrite($var) $clearLine
33      #end for
34
35      ! OPTIONAL IN ARGUMENTS
36      #for $var in $args.intentIn.optionals.usedVariables
37        IF (PRESENT($var.container(0))) THEN $mergeBegin('present')
38          $ftgWrite($var) $clearLine
39        END IF $mergeEnd('present')
40      #end for
41    END IF
42
43    ! REQUIRED OUT + INOUT ARGUMENTS
44    ⋮
45    ! OPTIONAL OUT + INOUT ARGUMENTS
46    ⋮
47    ! GLOBALS
48    ⋮
49
50    CALL ftg_${subroutine.name}_close_serializer(stage)
51
52  END SUBROUTINE ftg_${subroutine.name}_capture_data
53
54  $epilogue
```

Der Codeabschnitt captureAfterSubroutine wird in das PUT-Modul direkt nach der PUT eingefügt. Hier können zusätzliche Prozeduren definiert werden, die für das Capture benötigt werden. Zur Veranschaulichung ist hier nur ein Teil dieses Teiltemplates abgebildet. Nicht dargestellt sind zwei SUBROUTINEN zur Initialisierung und Aufräumen der Serialbox2-Bibliothek: `ftg_${subroutine.name}_init_serializer` und `ftg_${subroutine.name}_close_serializer`.

Dargestellt ist dagegen die SUBROUTINE `ftg_${subroutine.name}_capture_active` (Zeile 3–8), die aufgerufen wird, um zu überprüfen, ob die aktuelle Ausführung der PUT aufgezeichnet werden soll. In dieser einfachen Variante wird dazu lediglich überprüft, ob der Wert der Zählervariable `ftg_${subroutine.name}_round` der Konstanten `ftg_${subroutine.name}_capture_round` entspricht, also ob die PUT so oft ausgeführt wurde, wie festgelegt. Um den Aufzeichnungszeitpunkt zu bestimmen, kann der Wert von `ftg_${subroutine.name}_capture_round` entsprechend gesetzt

werden. Wenn eine komplexere Capturebedingung formuliert werden soll, kann `ftg_${subroutine.name}_capture_active` beliebig verändert werden, entweder bereits im Template oder im generierten Code.

Die Subroutine `ftg_${subroutine.name}_capture_data` (Zeile 12–52) ist für die Aufzeichnung der Variablen zuständig. Sie beginnt mit dem Import der benötigten Modulvariablen und Typen anderer Module (Zeile 14+15). In Zeile 17 wird das Dummyargument deklariert, über das angegeben wird, ob Ein- oder Ausgabedaten aufgezeichnet werden sollen. Dann folgen die Variablendeklarationen für die Dummyargumente der Subroutine mit Hilfe der `specs`-Platzhalterfunktion (Zeile 18). Wenn es sich bei der PUT um eine Funktion handelt, wird auch für das Funktionsergebnis ein Dummyargument deklariert (Zeilen 19–21).

Nach der Initialisierung der Serialbox2-Bibliothek (Zeile 26) wird, sofern wir uns beim Aufzeichnen der Eingabedaten befinden, nacheinander zunächst über die verwendeten Variablen/Komponenten der obligatorischen In-Argumente (Zeilen 31+33) und dann über die optionalen In-Argumente (Zeilen 36–40) iteriert. Anschließend sind die Inout- und Out-Argumente und die globalen Variablen an der Reihe (Zeilen 43–48, aus Platzgründen nicht abgedruckt). Diese werden sowohl für die Ein- als auch für die Ausgabedaten aufgezeichnet.

Für jede der benutzten Variablen/Komponenten wird der Code für das Aufzeichnen generiert. Auch wenn dieser durch die Verwendung der Serialbox2-Bibliothek in vielen Fällen nur aus einer Zeile pro Variable/Komponente besteht, ist die Generierung dieser Zeile(n) sehr komplex und daher in eine eigene Platzhaltermethode `ftgWrite` ausgelagert. Diese wird am Ende dieses Abschnitts erläutert. Für die optionalen Dummyargumente der PUT wird ein umschließendes `IF (PRESENT(...)) ... END IF` ausgegeben (Zeilen 37+39), da diese nur aufgezeichnet werden können, wenn sie beim Aufruf der PUT tatsächlich übergeben wurden. Die Platzhalterfunktionen `$mergeBegin` und `$mergeEnd` in diesen Zeilen bewirken, dass aufeinanderfolgende Fallunterscheidungen mit derselben Bedingung von FTG zusammengefasst werden.

D.3.7 export

BaseCompare/export.beforecontains.tmpl

```
1  #if $globals.exports or $module.name == $subroutine.moduleName and
       ↪ $subroutine.export
2  \#ifdef __FTG__
3    $prologue
4
5    $globals.exports
6    #if $module.name == $subroutine.moduleName
7      $subroutine.export
8    #end if
9
10   $epilogue
11 \#endif
12 #end if
```

Der Codeabschnitt exportBeforeContains wird ebenso wie captureBeforeContains direkt über dem CONTAINS-Schlüsselwort eines Moduls eingefügt, jedoch nicht nur im PUT-Modul, sondern in allen Modulen, von denen MODULVARIABLEN aufgezeichnet werden sollen. Er dient dazu, private MODULVARIABLEN und Typen, die von der PUT benötigt werden, durch Einfügen einer PUBLIC-Anweisung öffentlich zu machen. In dem PUT-Modul selbst kann auf gleiche Weise auch die PUT exportiert werden, sofern auch diese eigentlich privat ist.

Durch `$globals.exports` wird hier die PUBLIC-Anweisung für MODULVARIABLE und Typen ausgegeben (Zeile 5), mit `$subroutine.export` die PUT (Zeile 7). `$subroutine.export` wird nur ausgegeben, wenn der Name des aktuellen Moduls (`$module.name`) gleich dem Namen des PUT-Moduls ist (`$subroutine.moduleName`). Besteht in einem Modul kein Bedarf private Element zu exportieren, sind die entsprechenden Platzhalter leer. Dies wird zusätzlich in der umschließenden `#if`-Bedingung geprüft (Zeile 1), damit auch `$prologue` und `$epilogue` nur Falle nicht leerer Platzhalter ausgegeben werden. Um die ursprüngliche Kapselung nicht aufzubrechen, bietet es sich an, den Exportcode zusätzlich mit Präprozessordirektiven zu umschließen, wie hier beispielhaft in den Zeilen 2+11, um sicherzustellen, dass die eigentlich privaten Elemente nur exportiert werden, wenn ein entsprechendes Compilerflag gesetzt ist.

D.3.8 replay

BaseCompare/replay.tmpl

```
1  PROGRAM ftg_${subroutine.name}_test
2
3    USE mpi
4    USE m_ser_ftg, ONLY: ftg_set_serializer, ftg_set_savepoint,
         ↪ ftg_destroy_serializer, ftg_destroy_savepoint,
         ↪ ftg_print_serializer_debuginfo, ftg_field_exists, ftg_get_bounds,
         ↪ ftg_read, ftg_allocate_and_read_pointer,
         ↪ ftg_allocate_and_read_allocatable
5    USE m_ser_ftg_cmp, ONLY: ftg_compare
6    USE $module.name, ONLY: $subroutine.name
7
8    $globals.imports
9    $types.imports
10
11 IMPLICIT NONE
12
13 INTEGER (kind=4) :: rank, error, failure_count
14 $args.specs(intent = '', allocatable = True)
15 #if $subroutine.isFunction
16   $result.spec(name = $subroutine.name + '_result', intent = '',
         ↪ allocatable = False, pointer = False)
17 #end if
18
19 CALL MPI_INIT(error)
20 CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, error)
21
22 CALL ftg_${subroutine.name}_replay_input($commaList($args))
23
24 #if $subroutine.isFunction
25   ${subroutine.name}_result = ${subroutine.name}($commaList($args))
26 #else
27   CALL ${subroutine.name}($commaList($args))
28 #end if
29
30 #if $subroutine.isFunction
31     #set $resultArg = $subroutine.name + '_result'
32 #else
33   #set $resultArg = ''
34 #end if
35
36 CALL ftg_${subroutine.name}_compare_output($commaList($args.allOut,
       ↪ $resultArg, 'failure_count'))
37
38 IF (failure_count > 0) THEN
39   WRITE (*,'(A,I0,A,I0,A)') 'Rank #', rank, ': *** TEST FAILED ***'
40 ELSE
41   WRITE (*,'(A,I0,A)') 'Rank #', rank, ': *** TEST PASSED ***'
42 END IF
43
44 CALL MPI_FINALIZE(error)
45
46 CONTAINS
47 ⋮
```

Der Replaycodeabschnitt dient der Generierung des Testprogramms. Er ist der längste Abschnitt. An dieser Stelle ist nur das Hauptprogramm dargestellt.

Daneben enthält der Codeabschnitt zum einen SUBROUTINEN zum Laden der Ein- und Vergleichen der Ausgabedaten. Diese sind ähnlich aufgebaut wie die SUBROUTINE zum Schreiben der Ein- und Ausgabedaten in captureAfterSubroutine. Ebenfalls nicht aufgeführt sind zum anderen auch hier die SUBROUTINEN zum Initialisieren und Aufräumen der Serialbox2-Bibliothek.

Das Hauptprogramm importiert zunächst einige Elemente anderer Module, einschließlich der PUT (Zeilen 3–6) sowie die benötigten MODULVARIABLEN (Zeile 8) und Typen (Zeile 9). Die `USE`-Anweisungen für Letztere werden mit Hilfe der Platzhaltermethode `imports` erzeugt.

In Zeile 13 werden einige Hilfsvariablen deklariert. Anschließend erfolgt die Deklaration von Variablen, die als ARGUMENTE der PUT übergeben werden sollen (Zeilen 14–17) bzw. das Funktionsergebnis, sofern es sich bei der PUT um eine FUNKTION handelt (Zeilen 15–17). In Zeile 19 folgt die Initialisierung der MPI-Umgebung, welche am Ende (Zeile 44) wieder aufgeräumt wird. Außerdem wird in Zeile 20 die laufende Nummer (*Rang*) des aktuellen MPI-Prozesses abgefragt.

In Zeile 22 wird mit `ftg_${subroutine.name}_replay_input` die SUBROUTINE zum Laden der aufgezeichneten Eingabedaten ausgeführt und schließlich die PUT selbst aufgerufen (Zeilen 24–28). Handelt es sich bei der PUT um eine FUNKTION, wird der Rückgabewert der zuvor deklarierten Variable zugewiesen; handelt es sich um eine SUBROUTINE, wird diese mit dem `CALL`-Schlüsselwort aufgerufen.

Nach Ausführung der PUT werden die Ergebnisse in der SUBROUTINE `ftg_${subroutine.name}_compare_output` mit den aufgezeichneten Daten verglichen (Zeile 36). Anschließend wird eine Nachricht ausgegeben, ob der Test erfolgreich war. Jeder MPI-Prozess gibt sein eigenes Testergebnis aus, vorangestellt ist jeweils der Rang des Prozesses. In der Praxis bietet es sich an, darüber hinaus ein aggregiertes Ergebnis auszugeben. Zur Vereinfachung des Templatecodes wurde hiervon in diesem Beispiel abgesehen.

D.3.9 ftgWrite, ftgRead, ftgCompare

Während die bisher dargestellten Teiltemplates relativ leicht zu erläutern waren, sind die Templatemethoden `ftgWrite`, `ftgRead`, `ftgCompare`, die für die Generierung der Zeilen zum Schreiben, Lesen bzw. Vergleichen einer Variablen zuständig

sind, weitaus komplexer. Ein Grund für die Komplexität sind die Fortran-spezifischen Regeln zur Referenzierbarkeit von KOMPONENTEN.

Beispiel D.1: Komponentenreferenzierung

Gegeben sei die folgende Arrayvariable des Typs `B` aus Beispiel 2.8:

```
1 TYPE(B) :: bb(:)
```

Die KOMPONENTE `bb%a%r` ließe sich direkt im Quellcode referenzieren. Die Variable `bb` ist ein eindimensionales Array, die KOMPONENTEN `a` sowie `r` sind jeweils Skalare. Insgesamt würde `bb%a%r` somit wie ein eindimensionales Array vom Typ `REAL` behandelt werden, da `r` vom Typ `REAL` ist. Der entsprechende Aufruf der Serialbox2-SUBROUTINE zum Speichern der Variable sähe so aus:

```
2 CALL ftg_write('bb%a%r', bb%a%r, LBOUND(bb%a%r), UBOUND(bb%a%r))
```

Das erste ARGUMENT gibt den eindeutigen Namen der zu speichernden Variable an, das zweite ist die Variable selbst, das dritte und vierte Argument geben die Indexgrenzen des Arrays an. Die Prozedur `ftg_write` ist überladen, d.h. tatsächlich würde hier eine spezifische SUBROUTINE für eindimensionale `REAL`-Arrays aufgerufen werden.

Wäre nun aber beispielsweise die KOMPONENTE `a` vom Typ `B` als `POINTER` definiert, dürfte man `bb%a%r` nicht direkt im Quellcode referenzieren. Stattdessen müsste jedes Element von `bb` separat behandelt werden.

```
3 DO i = LBOUND(bb, 1), UBOUND(bb, 1)
4   WRITE (name, '(A,I0,A)') 'bb(', i, ')%a%r'
5   CALL ftg_write(name, bb(i)%a%r)
6 DO END
```

Dazu muss über alle Elemente von `bb` iteriert werden. Auch hier werden die Indexgrenzen mit Hilfe von `LBOUND` und `UBOUND` abgefragt. Mit Hilfe des `WRITE`-Befehls wird zunächst der Name, der nun den Index von `bb` enthält (z.B. `bb(1)%a%r`), zusammengesetzt. Anschließend wird `ftg_write` für die KOMPONENTE `%a%r` des aktuellen `bb`-Elements aufgerufen. Da es sich bei `bb(i)%a%r` nicht mehr um ein Array, sondern um ein Skalar handelt, entfällt die Angabe der Indexgrenzen.

Zur Problematik der Komponentenreferenzierung kommen u.a. Sonderbehandlungen für den Fall, dass per fcg-Konfiguration ein konkreter für einen abstrakten Typ eingesetzt wird, hinzu. Insgesamt führt dies zu folgendem Code für die Platzhaltermethode `ftgWrite`. `ftgRead` und `ftgCompare` sehen ähnlich aus. Der Abdruck an dieser Stelle soll lediglich die Komplexität verdeutlichen, auf eine detaillierte Erläuterung

wird verzichtet. Aufgrund dieser Komplexität ist eine Bearbeitung dieser Templateteile durch die BenutzerIn nur eingeschränkt bzw. nur mit Unterstützung möglich.

BaseCompare/capture.ftgWrite.tmpl

```
1  #set $var = $ftgWrite_var
2  #if $var.dim > 4 or $var.type.upper().startswith('CHARACTER'):
3    ! *** WARNING: Type not supported by serialbox ***
4    !     $var
5    !     $var.type, DIMENSION($var.dim)
6  #else
7    #set $closeStatements = []
8    #set $mandDim = $var.mandatoryDimensions
9    #set $indices = ['ftg_d1', 'ftg_d2', 'ftg_d3', 'ftg_d4', 'ftg_d5', '
         ↪ ftg_d6']
10   #set $d = 0
11   #set $ad = 0
12   #set $aliasVar = $var
13   #set $filledVar = $fillIndices($var, $mandDim, *$indices)
14   #set $filledAlias = $filledVar
15   #for $level in $var.levels
16     #set $container = $aliasVar.container($level - ($var.level - $aliasVar.
           ↪ level))
17     #set $filledContainer = $fillIndices($container, $ad, *$indices[$d -
           ↪ $ad:])
18     #if $container.polymorph
19       #set $alias = 'polym' + str($level)
20       SELECT TYPE ($alias => $filledContainer) $mergeBegin('switch' + str(
             ↪ $level))
21       TYPE IS ($container.dynamicType) $mergeBegin('switch' + str($level))
22       #silent $closeStatements.append('END SELECT ' + $mergeEnd('switch' +
             ↪ str($level)))
23       #set $aliasVar = $var.alias($alias, $level)
24       #set $filledAlias = $filledVar.alias($alias, $level)
25       #set $ad = 0
26     #else
27       #set $aa = $allocatedOrAssociated($filledContainer, $level)
28       #if $aa
29         IF ($aa) THEN $mergeBegin('if' + str($level))
30         #silent $closeStatements.append('END IF ' + $mergeEnd('if' + str(
               ↪ $level)))
31       #end if
32     #end if
33     #if $level < $var.level and $needsRegistration($filledContainer)
34       #if $d > 0
35         $writeVarNameWithFilledIndicesToString($container, "ftg_name", $d,
               ↪ *$indices)
36         #set $contName = 'ftg_name'
37       #else
38         #set $contName = '"' + str($container) + '"'
39       #end if
40       #set $contType = $container.type
41       #if $d < $container.totalDim
42         CALL ftg_register_only($contName, "$contType", LBOUND(
               ↪ $filledContainer), UBOUND($filledContainer))
43       #else
44         CALL ftg_register_only($contName, "$contType")
45       #end if
46       #silent $setRegistered($filledContainer)
47     #end if
48     #if $d < $mandDim
49       #set $loopDims = range($d + 1, $d + $container.dim + 1)
50       #set $d += $container.dim
```

```
51        #set $ad += $container.dim
52        #set $cd = 0
53        #for $ld in $loopDims
54          #set $cd += 1
55          DO ftg_d$ld = LBOUND($filledContainer, $cd), UBOUND(
                ↪ $filledContainer, $cd) $mergeBegin('do' + str($level))
56          #silent $closeStatements.append('END DO ' + $mergeEnd('do' + str(
                ↪ $level)))
57        #end for
58      #end if
59    #end for
60    #if $mandDim > 0
61      $writeVarNameWithFilledIndicesToString($var, 'ftg_name', $mandDim, '
            ↪ ftg_d1', 'ftg_d2', 'ftg_d3', 'ftg_d4', 'ftg_d5', 'ftg_d6')
62      #set $varName = 'ftg_name'
63    #else
64      #set $varName = '"' + str($var) + '"'
65    #end if
66    #if $mandDim < $var.totalDim
67      CALL ftg_write($varName, $filledAlias, LBOUND($filledAlias), UBOUND(
            ↪ $filledAlias))
68    #else
69      CALL ftg_write($varName, $filledAlias)
70    #end if
71    #for $close in reversed($closeStatements)
72      $close
73    #end for
74    #if 'END DO' in [s[0:6] for s in $closeStatements]
75
76    #end if
77  #end if
```

Anhang E

Experimente

Im Folgenden werden verschiedene Details zu den in Kapitel 11 beschriebenen Experimenten aufgeführt.

E.1 Testrechner

E.1.1 PC

Bei dem verwendeten Test-PC handelt es sich um ein Notebook des Typs *Lenovo ThinkPad X1 Carbon 2017 (20HQS03P00)* mit folgenden Spezifikationen:

Prozessor Intel Core i7-7500U (2x 2,7 GHz)

Arbeitsspeicher 16 GB DDR3

Speicher 512GB Solid State Disk M2

Betriebssystem Kubuntu 18.10

Dateisystem ext4

E.1.2 Mistral

Die verwendete Rechenknoten des Hochleistungsrechners Mistral des DKRZ verfügen über folgende Spezifikation (vgl. DKRZ, Configuration):

Prozessoren 2x 12-core Intel Xeon E5-2680 v3 (Haswell) @ 2,5 GHz

Arbeitsspeicher 64 GB

Betriebssystem Red Hat Enterprise Linux 6.4

Dateisystem Lustre

E.2 fcg/FTG-Konfigurationen

In diesem Abschnitt werden die fcg- und FTG-Konfigurationsdateien aufgeführt, die für die Experimente mit den jeweiligen Modellen verwendet wurden.

E.2.1 ICON

Aufgeführt ist die Konfigurationsdatei, die auf dem PC verwendet wurde. Bei der Konfiguration auf der Mistral unterscheiden sich lediglich die Ordnerpfade.

Die Konfiguration für die Subroutine `interface_full` unterscheidet sich von der hier aufgeführten darin, dass die zu JSBACH gehörenden Module nicht in `EXCLUDE_MODULES` enthalten sind. Für die Subroutinen `integrate_nh` sowie `update_surface` wurde zudem statt des Templates IconStandalone das Template IconJsbachMock verwendet.

config_fortrancallgraph.py

```
import os

FCG_DIR = os.path.dirname(os.path.realpath(__file__))
CACHE_DIR = FCG_DIR + '/cache/icon'

ICON_DIR = '/home/christian/workspace/icon'

ASSEMBLER_DIRS =
  [ ICON_DIR + '/build/x86_64-unknown-linux-gnu/src',
    ICON_DIR + '/build/x86_64-unknown-linux-gnu/externals/mtime/src',
    ICON_DIR + '/build/x86_64-unknown-linux-gnu/externals/self/src',
    ICON_DIR + '/build/x86_64-unknown-linux-gnu/externals/tixi/src',
    ICON_DIR + '/build/x86_64-unknown-linux-gnu/externals/yac/src',
    ICON_DIR + '/build/x86_64-unknown-linux-gnu/support'
  ]

SOURCE_DIRS =
  [ ICON_DIR + '/build/x86_64-unknown-linux-gnu/pp',
    ICON_DIR + '/build/x86_64-unknown-linux-gnu/externals/mtime/pp',
    ICON_DIR + '/build/x86_64-unknown-linux-gnu/externals/self/pp',
    ICON_DIR + '/build/x86_64-unknown-linux-gnu/externals/tixi/pp',
    ICON_DIR + '/build/x86_64-unknown-linux-gnu/externals/yac/pp',
    ICON_DIR + '/build/x86_64-unknown-linux-gnu/support'
  ]

SOURCE_FILES_PREPROCESSED = True

SPECIAL_MODULE_FILES =
  { 'mo_mcrph_sb': 'mo_2mom_mcrph_driver.f90',
    'mo_lrtm': 'mo_lrtm_driver.f90', 'ppm_extents': 'mo_extents.f90',
    'ppm_distributed_array': 'mo_dist_array.f90',
    'psrad_rrsw_kg16': 'mo_psrad_srtm_kgs.f90',
    'psrad_rrsw_kg17': 'mo_psrad_srtm_kgs.f90',
    'psrad_rrsw_kg18': 'mo_psrad_srtm_kgs.f90',
    'psrad_rrsw_kg19': 'mo_psrad_srtm_kgs.f90',
    'psrad_rrsw_kg20': 'mo_psrad_srtm_kgs.f90',
```

```
    'psrad_rrsw_kg21': 'mo_psrad_srtm_kgs.f90',
    'psrad_rrsw_kg22': 'mo_psrad_srtm_kgs.f90',
    'psrad_rrsw_kg23': 'mo_psrad_srtm_kgs.f90',
    'psrad_rrsw_kg24': 'mo_psrad_srtm_kgs.f90',
    'psrad_rrsw_kg25': 'mo_psrad_srtm_kgs.f90',
    'psrad_rrsw_kg26': 'mo_psrad_srtm_kgs.f90',
    'psrad_rrsw_kg27': 'mo_psrad_srtm_kgs.f90',
    'psrad_rrsw_kg28': 'mo_psrad_srtm_kgs.f90',
    'psrad_rrsw_kg29': 'mo_psrad_srtm_kgs.f90',
    'rrlw_planck': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg01': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg02': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg03': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg04': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg05': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg06': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg07': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg08': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg09': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg10': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg11': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg12': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg13': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg14': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg15': 'mo_psrad_lrtm_kgs.f90',
    'psrad_rrlw_kg16': 'mo_psrad_lrtm_kgs.f90',
    'rrlw_kg01': 'mo_lrtm_kgs.f90', 'rrlw_kg02': 'mo_lrtm_kgs.f90',
    'rrlw_kg03': 'mo_lrtm_kgs.f90', 'rrlw_kg04': 'mo_lrtm_kgs.f90',
    'rrlw_kg05': 'mo_lrtm_kgs.f90', 'rrlw_kg06': 'mo_lrtm_kgs.f90',
    'rrlw_kg07': 'mo_lrtm_kgs.f90', 'rrlw_kg08': 'mo_lrtm_kgs.f90',
    'rrlw_kg09': 'mo_lrtm_kgs.f90', 'rrlw_kg10': 'mo_lrtm_kgs.f90',
    'rrlw_kg11': 'mo_lrtm_kgs.f90', 'rrlw_kg12': 'mo_lrtm_kgs.f90',
    'rrlw_kg13': 'mo_lrtm_kgs.f90', 'rrlw_kg14': 'mo_lrtm_kgs.f90',
    'rrlw_kg15': 'mo_lrtm_kgs.f90', 'rrlw_kg16': 'mo_lrtm_kgs.f90',
    'yoesrta16': 'mo_srtm_kgs.f90', 'yoesrta17': 'mo_srtm_kgs.f90',
    'yoesrta18': 'mo_srtm_kgs.f90', 'yoesrta19': 'mo_srtm_kgs.f90',
    'yoesrta20': 'mo_srtm_kgs.f90', 'yoesrta21': 'mo_srtm_kgs.f90',
    'yoesrta22': 'mo_srtm_kgs.f90', 'yoesrta23': 'mo_srtm_kgs.f90',
    'yoesrta24': 'mo_srtm_kgs.f90', 'yoesrta25': 'mo_srtm_kgs.f90',
    'yoesrta26': 'mo_srtm_kgs.f90', 'yoesrta27': 'mo_srtm_kgs.f90',
    'yoesrta28': 'mo_srtm_kgs.f90', 'yoesrta29': 'mo_srtm_kgs.f90',
    'mtime': 'libmtime.f90', 'mtime_calendar': 'libmtime.f90',
    'mtime_julianday': 'libmtime.f90', 'mtime_date': 'libmtime.f90',
    'mtime_time': 'libmtime.f90', 'mtime_datetime': 'libmtime.f90',
    'mtime_timedelta': 'libmtime.f90', 'mtime_events': 'libmtime.f90',
    'mtime_eventgroups': 'libmtime.f90', 'mtime_utilities': 'libmtime.f90',
    'mtime_print_by_callback': 'libmtime.f90',
    'mtime_juliandelta': 'libmtime.f90',
    'mo_meteogram_config': 'mo_mtgrm_config.f90',
    'mo_meteogram_output': 'mo_mtgrm_output.f90',
    'mo_art_sedi_interface': 'mo_art_sedimentation_interface.f90',
    'mo_assimi_config_class': 'mo_assimi_config_class_dsl4jsb.f90',
    'mo_assimi_constants': 'mo_assimi_constants_dsl4jsb.f90',
    'mo_assimi_interface': 'mo_assimi_interface_dsl4jsb.f90',
    'mo_assimi_memory_class': 'mo_assimi_memory_class_dsl4jsb.f90',
    'mo_assimi_process': 'mo_assimi_process_dsl4jsb.f90',
    'mo_a2l_memory_class': 'mo_atm2land_memory_class_dsl4jsb.f90',
    'mo_atmland_interface': 'mo_atmland_interface_dsl4jsb.f90',
    'mo_carbon_config_class': 'mo_carbon_config_class_dsl4jsb.f90',
    'mo_carbon_constants': 'mo_carbon_constants_dsl4jsb.f90',
    'mo_carbon_init': 'mo_carbon_init_dsl4jsb.f90',
    'mo_carbon_interface': 'mo_carbon_interface_dsl4jsb.f90',
    'mo_carbon_memory_class': 'mo_carbon_memory_class_dsl4jsb.f90',
    'mo_carbon_process': 'mo_carbon_process_dsl4jsb.f90',
    'mo_hd_config_class': 'mo_hd_config_class_dsl4jsb.f90',
```

```
    'mo_hd_init': 'mo_hd_init_dsl4jsb.f90',
    'mo_hd_interface': 'mo_hd_interface_dsl4jsb.f90',
    'mo_hd_memory_class': 'mo_hd_memory_class_dsl4jsb.f90',
    'mo_hd_reservoir_cascade': 'mo_hd_reservoir_cascade_dsl4jsb.f90',
    'mo_hsm_class': 'mo_hsm_class_dsl4jsb.f90',
    'mo_hydro_config_class': 'mo_hydro_config_class_dsl4jsb.f90',
    'mo_hydro_constants': 'mo_hydro_constants_dsl4jsb.f90',
    'mo_hydro_init': 'mo_hydro_init_dsl4jsb.f90',
    'mo_hydro_interface': 'mo_hydro_interface_dsl4jsb.f90',
    'mo_hydro_memory_class': 'mo_hydro_memory_class_dsl4jsb.f90',
    'mo_hydro_process': 'mo_hydro_process_dsl4jsb.f90',
    'mo_hydro_util': 'mo_hydro_util_dsl4jsb.f90',
    'mo_interface_hd_ocean': 'mo_interface_hd_ocean_dsl4jsb.f90',
    'mo_jsb4_forcing': 'mo_jsb4_forcing_dsl4jsb.f90',
    'mo_jsb_base': 'mo_jsb_base_dsl4jsb.f90',
    'mo_jsb_class': 'mo_jsb_class_dsl4jsb.f90',
    'mo_jsb_config_class': 'mo_jsb_config_class_dsl4jsb.f90',
    'mo_jsb_control': 'mo_jsb_control_dsl4jsb.f90',
    'mo_jsb_domain_iface': 'mo_jsb_domain_iface_dsl4jsb.f90',
    'mo_jsb_grid_class': 'mo_jsb_grid_class_dsl4jsb.f90',
    'mo_jsb_grid': 'mo_jsb_grid_dsl4jsb.f90',
    'mo_jsb_interface': 'mo_jsb_interface_dsl4jsb.f90',
    'mo_jsb_io': 'mo_jsb_io_dsl4jsb.f90',
    'mo_jsb_io_iface': 'mo_jsb_io_iface_dsl4jsb.f90',
    'mo_jsb_io_netcdf': 'mo_jsb_io_netcdf_dsl4jsb.f90',
    'mo_jsb_io_netcdf_iface': 'mo_jsb_io_netcdf_iface_dsl4jsb.f90',
    'mo_jsb_lct_class': 'mo_jsb_lct_class_dsl4jsb.f90',
    'mo_jsb_lctlib_class': 'mo_jsb_lctlib_class_dsl4jsb.f90',
    'mo_jsb_memory_class': 'mo_jsb_memory_class_dsl4jsb.f90',
    'mo_jsb_model_class': 'mo_jsb_model_class_dsl4jsb.f90',
    'mo_jsb_model_init': 'mo_jsb_model_init_dsl4jsb.f90',
    'mo_jsb_model_usecases': 'mo_jsb_model_usecases_dsl4jsb.f90',
    'mo_jsb_namelist_iface': 'mo_jsb_namelist_iface_dsl4jsb.f90',
    'mo_jsb_nml_iface': 'mo_jsb_nml_iface_dsl4jsb.f90',
    'mo_jsb_parallel': 'mo_jsb_parallel_dsl4jsb.f90',
    'mo_jsb_parallel_iface': 'mo_jsb_parallel_iface_dsl4jsb.f90',
    'mo_jsb_physical_constants': 'mo_jsb_physical_constants_dsl4jsb.f90',
    'mo_jsb_process_class': 'mo_jsb_process_class_dsl4jsb.f90',
    'mo_jsb_process_factory': 'mo_jsb_process_factory_dsl4jsb.f90',
    'mo_jsb_task_class': 'mo_jsb_task_class_dsl4jsb.f90',
    'mo_jsb_test': 'mo_jsb_test_dsl4jsb.f90',
    'mo_jsb_tile_class': 'mo_jsb_tile_class_dsl4jsb.f90',
    'mo_jsb_tile': 'mo_jsb_tile_dsl4jsb.f90',
    'mo_jsb_time': 'mo_jsb_time_dsl4jsb.f90',
    'mo_jsb_time_iface': 'mo_jsb_time_iface_dsl4jsb.f90',
    'mo_jsb_utils_iface': 'mo_jsb_utils_iface_dsl4jsb.f90',
    'mo_jsb_var_class': 'mo_jsb_var_class_dsl4jsb.f90',
    'mo_jsb_varlist': 'mo_jsb_varlist_dsl4jsb.f90',
    'mo_jsb_varlist_iface': 'mo_jsb_varlist_iface_dsl4jsb.f90',
    'mo_jsb_version': 'mo_jsb_version_dsl4jsb.f90',
    'mo_jsb_vertical_axes': 'mo_jsb_vertical_axes_dsl4jsb.f90',
    'mo_jsb_vertical_axes_iface': 'mo_jsb_vertical_axes_iface_dsl4jsb.f90',
    'mo_land2atm_memory_class': 'mo_land2atm_memory_class_dsl4jsb.f90',
    'mo_pheno_config_class': 'mo_pheno_config_class_dsl4jsb.f90',
    'mo_pheno_constants': 'mo_pheno_constants_dsl4jsb.f90',
    'mo_pheno_init': 'mo_pheno_init_dsl4jsb.f90',
    'mo_pheno_interface': 'mo_pheno_interface_dsl4jsb.f90',
    'mo_pheno_memory_class': 'mo_pheno_memory_class_dsl4jsb.f90',
    'mo_pheno_process': 'mo_pheno_process_dsl4jsb.f90',
    'mo_phy_schemes': 'mo_phy_schemes_dsl4jsb.f90',
    'mo_phy_schemes_iface': 'mo_phy_schemes_iface_dsl4jsb.f90',
    'mo_physical_constants_iface': 'mo_physical_constants_iface_dsl4jsb.f90',
    'mo_rad_config_class': 'mo_rad_config_class_dsl4jsb.f90',
    'mo_rad_constants': 'mo_rad_constants_dsl4jsb.f90',
    'mo_rad_init': 'mo_rad_init_dsl4jsb.f90',
```

```
    'mo_rad_interface': 'mo_rad_interface_dsl4jsb.f90',
    'mo_rad_memory_class': 'mo_rad_memory_class_dsl4jsb.f90',
    'mo_rad_process': 'mo_rad_process_dsl4jsb.f90',
    'mo_seb_config_class': 'mo_seb_config_class_dsl4jsb.f90',
    'mo_seb_init': 'mo_seb_init_dsl4jsb.f90',
    'mo_seb_interface': 'mo_seb_interface_dsl4jsb.f90',
    'mo_seb_lake': 'mo_seb_lake_dsl4jsb.f90',
    'mo_seb_land': 'mo_seb_land_dsl4jsb.f90',
    'mo_seb_memory_class': 'mo_seb_memory_class_dsl4jsb.f90',
    'mo_sse_config_class': 'mo_sse_config_class_dsl4jsb.f90',
    'mo_sse_constants': 'mo_sse_constants_dsl4jsb.f90',
    'mo_sse_init': 'mo_sse_init_dsl4jsb.f90',
    'mo_sse_interface': 'mo_sse_interface_dsl4jsb.f90',
    'mo_sse_memory_class': 'mo_sse_memory_class_dsl4jsb.f90',
    'mo_sse_process': 'mo_sse_process_dsl4jsb.f90',
    'mo_sse_util': 'mo_sse_util_dsl4jsb.f90',
    'mo_turb_config_class': 'mo_turb_config_class_dsl4jsb.f90',
    'mo_turb_constants': 'mo_turb_constants_dsl4jsb.f90',
    'mo_turb_init': 'mo_turb_init_dsl4jsb.f90',
    'mo_turb_interface': 'mo_turb_interface_dsl4jsb.f90',
    'mo_turb_memory_class': 'mo_turb_memory_class_dsl4jsb.f90',
    'mo_util': 'mo_util_dsl4jsb.f90',
    'xt_core': 'xt_core_f.f90',
    'xt_idxlist_collection': 'xt_idxlist_collection_f.f90',
    'xt_idxlist_abstract': 'xt_idxlist_f.f90',
    'xt_idxsection': 'xt_idxsection_f.f90',
    'xt_mpi': 'xt_mpi_f.f90', 'xt_idxmod': 'xt_idxmod_f.f90',
    'xt_idxvec': 'xt_idxvec_f.f90',
    'xt_idxstripes': 'xt_idxstripes_f.f90',
    'xt_redist_base': 'xt_redist_f.f90',
    'xt_requests': 'xt_request_f.f90', 'xt_sort': 'xt_sort_f.f90',
    'xt_xmap_abstract': 'xt_xmap_f.f90',
    'xt_xmap_intersection': 'xt_xmap_intersection_f.f90'
  }

EXCLUDE_MODULES =
  [ #Standard libraries
    'iso_fortran_env', 'iso_c_binding', 'ifcore', 'mpi', 'omp_lib',
    #Infrastructure
    'mo_exception', 'mo_mpi',
    #MESSY
    'messy_main_channel_bi', 'messy_main_tracer_bi', 'messy_main_timer_bi',
    #?
    'mo_remap_config', 'mo_utilities',
    #GME
    'gme_data_parameters', 'prognostic_pp',
    #RTTOV
    'mo_rttov_ifc',
    #ONLYWW
    'mo_wwonly',
    #ECHAM
    'mo_filename', 'mo_netcdf', 'mo_time_control', 'mo_decomposition',
    'mo_interpo', 'mo_io', 'mo_time_conversion', 'mo_time_event',
    'mo_gaussgrid', 'mo_tr_scatter', 'mo_transpose', 'mo_geoloc',
    'mo_memory_base', 'mo_control', 'mo_semi_impl', 'mo_submodel',
    #COSMO
    'meteo_utilities', 'environment', 'utilities', 'pp_utilities',
    'kind_parameters', 'src_stoch_physics', 'turbulence_data', 'data_gscp',
    'data_parameters', 'data_constants', 'data_runcontrol', 'data_parallel',
    'data_modelconfig', 'data_soil', 'data_fields',
    #NUDGING
    'data_lheat_nudge', 'src_lheating',
    #PRISM
    'mod_prism_proto',
    #SCT
```

```
    'sct',
    #SCLM
    'data_1d_global',
    #ICON-ART
    'mo_art_init', 'mo_art_tracer', 'mo_art_diag_state',
    'mo_art_read_emissions', 'mo_art_emission_pntSrc', 'mo_art_integration',
    'mo_art_emission_dust', 'mo_art_emission_volc_2mom',
    'mo_art_emission_chemtracer', 'mo_art_modes',
    'mo_art_emission_dust_simple', 'mo_art_emission_pollen', 'mo_art_data',
    'mo_art_emission_volc_1mom', 'mo_art_emission_seas',
    'mo_art_modes_linked_list', 'mo_art_emission_gasphase',
    'mo_art_aerosol_utilities', 'mo_art_clipping', 'mo_art_sedi_2mom',
    'mo_art_sedi_1mom', 'mo_art_drydepo_radioact', 'mo_art_depo_2mom',
    'mo_art_photolysis', 'mo_art_decay_radioact', 'mo_art_gasphase',
    'mo_art_chemtracer', 'mo_art_washout_volc', 'mo_art_washout_radioact',
    'mo_art_washout_aerosol', 'mo_art_radiation_aero',
    'mo_art_prepare_aerosol', 'mo_art_2mom_driver', 'mo_art_diagnostics',
    'mo_art_aero_optical_props', 'mo_art_surface_value', 'mo_art_diag_types',
    'mo_art_unit_conversion', 'mo_art_prescribed_state',
    #JSBACH
    'mo_assimi_config_class', 'mo_assimi_constants', 'mo_assimi_interface',
    'mo_assimi_memory_class', 'mo_assimi_process', 'mo_a2l_memory_class',
    'mo_atmland_interface', 'mo_carbon_config_class', 'mo_carbon_constants',
    'mo_carbon_init', 'mo_carbon_interface', 'mo_carbon_memory_class',
    'mo_carbon_process', 'mo_carbon_constants', 'mo_hd_config_class',
    'mo_hd_init', 'mo_hd_interface', 'mo_hd_memory_class',
    'mo_hd_reservoir_cascade', 'mo_hsm_class', 'mo_hydro_config_class',
    'mo_hydro_constants', 'mo_hydro_init', 'mo_hydro_interface',
    'mo_hydro_memory_class', 'mo_hydro_process', 'mo_hydro_util',
    'mo_interface_hd_ocean', 'mo_jsb4_forcing', 'mo_jsb_base', 'mo_jsb_class',
    'mo_jsb_config_class', 'mo_jsb_control', 'mo_jsb_domain_iface',
    'mo_jsb_grid_class', 'mo_jsb_grid', 'mo_jsb_interface', 'mo_jsb_io',
    'mo_jsb_io_iface', 'mo_jsb_io_netcdf', 'mo_jsb_io_netcdf_iface',
    'mo_jsb_lct_class', 'mo_jsb_lctlib_class', 'mo_jsb_memory_class',
    'mo_jsb_model_class', 'mo_jsb_model_init', 'mo_jsb_model_usecases',
    'mo_jsb_namelist_iface', 'mo_jsb_nml_iface', 'mo_jsb_parallel',
    'mo_jsb_parallel_iface', 'mo_jsb_physical_constants',
    'mo_jsb_process_class', 'mo_jsb_process_factory', 'mo_jsb_task_class',
    'mo_jsb_test', 'mo_jsb_tile_class', 'mo_jsb_tile', 'mo_jsb_time',
    'mo_jsb_time_iface', 'mo_jsb_utils_iface', 'mo_jsb_var_class',
    'mo_jsb_varlist', 'mo_jsb_varlist_iface', 'mo_jsb_version',
    'mo_jsb_vertical_axes', 'mo_jsb_vertical_axes_iface'
  ]

IGNORE_GLOBALS_FROM_MODULES = EXCLUDE_MODULES + ['mtime']
IGNORE_DERIVED_TYPES = []
ALWAYS_FULL_TYPES = ['datetime', 'timedelta']

ABSTRACT_TYPE_IMPLEMENTATIONS = {'t_comm_pattern':('mo_communication_orig','
    ↪ t_comm_pattern_orig'),
                                 't_comm_pattern_collection':('
                                     ↪ mo_communication_orig','
                                     ↪ t_comm_pattern_collection_orig')}
```

config_fortrantestgenerator.py

```
import os

FTG_DIR = os.path.dirname(os.path.realpath(__file__))
FCG_DIR = FTG_DIR + '/../fortrancallgraph'

#TEMPLATE = FTG_DIR + '/templates/IconJsbachMock/IconJsbachMock.tmpl'
TEMPLATE = FTG_DIR + '/templates/IconStandalone/IconStandalone.tmpl'
```

```
ICON_DIR = '/home/christian/workspace/icon'
TEST_SOURCE_DIR = ICON_DIR + '/src/tests'
TEST_DATA_BASE_DIR = ICON_DIR + '/ftg'

MODIFY_SOURCE_DIRS = [ICON_DIR + '/src',
                      ICON_DIR + '/externals/mtime/src',
                      ICON_DIR + '/externals/self/src',
                      ICON_DIR + '/externals/yac/src',
                      ICON_DIR + '/externals/tixi/src',
                      ICON_DIR + '/externals/jsbach/src',
                      ICON_DIR + '/support']
```

E.2.2 MOM6

config_fortrancallgraph.py

```
import os

FCG_DIR = os.path.dirname(os.path.realpath(__file__))
CACHE_DIR = FCG_DIR + '/cache/mom6'

MOM6_DIR = '/home/christian/workspace/MOM6-examples'
ASSEMBLER_DIRS = [MOM6_DIR + '/build/gnu/ocean_only/savetemps',
                  MOM6_DIR + '/build/gnu/shared/savetemps']
SOURCE_DIRS = ASSEMBLER_DIRS + [MOM6_DIR + '/src']

SOURCE_FILES_PREPROCESSED = True

SPECIAL_MODULE_FILES =
  { 'regional_dyes':'dye_example.f90',
    'MOM_int_tide_input':'MOM_internal_tide_input.f90',
    'MOM_PressureForce_AFV': 'MOM_PressureForce_analytic_FV.f90',
    'MOM_PressureForce_blk_AFV':'MOM_PressureForce_blocked_AFV.f90',
    'MOM_PressureForce_Mont':'MOM_PressureForce_Montgomery.f90',
    'MOM_set_visc':'MOM_set_viscosity.f90',
    'USER_tracer_example':'tracer_example.f90',
    'null_fms_io_test':'test_fms_io.f90',
    'null_test_horiz_interp':'test_horiz_interp.f90',
    'null_mpp_domains_test':'test_mpp_domains.f90',
    'null_mpp_test':'test_mpp.f90',
    'null_mpp_io_test':'test_mpp_io.f90',
    'null_mpp_pset_test':'test_mpp_pset.f90',
    'null_test_xgrid':'test_xgrid.f90'
  }

EXCLUDE_MODULES = []
IGNORE_GLOBALS_FROM_MODULES = EXCLUDE_MODULES
IGNORE_DERIVED_TYPES = []
ALWAYS_FULL_TYPES = []
ABSTRACT_TYPE_IMPLEMENTATIONS = {}
```

config_fortrantestgenerator.py

```
import os
```

```
FTG_DIR = os.path.dirname(os.path.realpath(__file__))
FCG_DIR = FTG_DIR + '/../fortrancallgraph'

TEMPLATE = FTG_DIR + '/templates/Standalone/Standalone.tmpl'

MOM6_DIR = '/home/christian/workspace/MOM6-examples'
TEST_SOURCE_DIR = MOM6_DIR + '/src/ftgtests'
TEST_DATA_BASE_DIR = MOM6_DIR + '/ftg'

MODIFY_SOURCE_DIRS = [MOM6_DIR + '/src']
```

E.2.3 CESM2

config_fortrancallgraph.py

```
import os

FCG_DIR = os.path.dirname(os.path.realpath(__file__))
CACHE_DIR = FCG_DIR + '/cache/cesm'

CESM_DIR= '/home/christian/workspaces/eclipse/cesm'
ASSEMBLER_DIRS = [CESM_DIR+ '/scratch/fkessler-savetemps/bld']
SOURCE_DIRS = ASSEMBLER_DIRS

SOURCE_FILES_PREPROCESSED = True

SPECIAL_MODULE_FILES = {'t_drv_timers_mod':'t_driver_timers_mod.f90'}
EXCLUDE_MODULES = ['mpi', 'netcdf', 'pnetcdf',
                   'c_interface_mod', 'iso_c_binding']

IGNORE_GLOBALS_FROM_MODULES = EXCLUDE_MODULES
IGNORE_DERIVED_TYPES = []
ALWAYS_FULL_TYPES = []
ABSTRACT_TYPE_IMPLEMENTATIONS = {}
```

config_fortrantestgenerator.py

```
import os

FTG_DIR = os.path.dirname(os.path.realpath(__file__))
FCG_DIR = FTG_DIR + '/../fortrancallgraph'

TEMPLATE = FTG_DIR + '/templates/CesmStandalone/CesmStandalone.tmpl'

CESM_DIR= '/home/christian/workspaces/eclipse/cesm'
TEST_SOURCE_DIR = CESM_DIR + '/components/ftg'
TEST_DATA_BASE_DIR = CESM_DIR + '/ftg'

# See cesm/cases/fkessler/env_case.xml <group id="case_comps">
#   + cesm/cases/fkessler/Buildconf/camconf/Filepath
MODIFY_SOURCE_DIRS = [CESM_DIR + '/components/cam/src/control',
                      CESM_DIR + '/components/cam/src/cpl',
                      CESM_DIR + '/components/cam/src/ionosphere',
                      CESM_DIR + '/components/cam/src/chemistry/aerosol',
```

```
                        CESM_DIR + '/components/cam/src/chemistry/bulk_aero',
                        CESM_DIR + '/components/cam/src/chemistry/mozart',
                        CESM_DIR + '/components/cam/src/chemistry/pp_terminator',
                        CESM_DIR + '/components/cam/src/chemistry/utils',
                        CESM_DIR + '/components/cam/src/dynamics/fv',
                        CESM_DIR + '/components/cam/src/dynamics/tests',
                        CESM_DIR + '/components/cam/src/physics/cam',
                        CESM_DIR + '/components/cam/src/physics/simple',
                        CESM_DIR + '/components/cam/src/unit_drivers',
                        CESM_DIR + '/components/cam/src/utils',
                        CESM_DIR + '/cime/src/components/stub_comps',
                        CESM_DIR + '/cime/src/drivers/mct',
                        CESM_DIR + '/cime/src/share/esmf_wrf_timemgr',
                        CESM_DIR + '/cime/src/externals/pio1/pio',
                        CESM_DIR + '/cime/src/share/timing',
                        CESM_DIR + '/cime/src/share/util']
```

E.2.4 NICAM

config_fortrancallgraph.py

```
import os

FCG_DIR = os.path.dirname(os.path.realpath(__file__))
CACHE_DIR = FCG_DIR + '/cache/nicam'

NICAM_DIR = '/home/dkrz/k203071/models/NICAM.16/NICAM.16'
ASSEMBLER_DIRS = [NICAM_DIR + '/NICAM/savetemps']
SOURCE_DIRS = ASSEMBLER_DIRS

SOURCE_FILES_PREPROCESSED = True

SPECIAL_MODULE_FILES = {'mod_cio':'mod_cio_nojcup.f90',
                        'mod_tool_option':'mod_option.f90',
                        'mod_physics':'mod_phystep.f90'}
EXCLUDE_MODULES = ['mpi']

IGNORE_GLOBALS_FROM_MODULES = EXCLUDE_MODULES
IGNORE_DERIVED_TYPES = []
ALWAYS_FULL_TYPES = []

ABSTRACT_TYPE_IMPLEMENTATIONS = {}
```

config_fortrantestgenerator.py

```
import os

FTG_DIR = os.path.dirname(os.path.realpath(__file__))
FCG_DIR = FTG_DIR + '/../fortrancallgraph'

TEMPLATE = FTG_DIR + '/templates/NicamStandalone/NicamStandalone.tmpl'

NICAM_DIR = '/home/dkrz/k203071/models/NICAM.16/NICAM.16'
TEST_SOURCE_DIR = NICAM_DIR + '/NICAM/ftgtests'
TEST_DATA_BASE_DIR = NICAM_DIR + '/ftg'
```

```
MODIFY_SOURCE_DIRS = [NICAM_DIR + '/NICAM/nhm',
                      NICAM_DIR + '/NICAM/prep',
                      NICAM_DIR + '/NICAM/share',
                      NICAM_DIR + '/NICAM/swm',
                      NICAM_DIR + '/NICAM/tool']
```

Abkürzungen

AGCM	Atmospheric General Circulation Model
AOGCM	Atmosphere-Ocean General Circulation Model
API	application programming interface
AST	abstract syntax tree
C&R	Capture & Replay
CGCM	Coupled General Circulation Model
CI	Continuous Integration
CLI	command-line interface
CMIP	Coupled Model Intercomparison Project
CSCS	Centro Svizzero di Calcolo Scientifico
CUT	code under test
DKRZ	Deutsches Klimarechenzentrum
DWD	Deutscher Wetterdienst
E2E	end to end
ENIAC	Enabling the ICON Model on Heterogeneous Architectures
ESM	Earth System Model
ETH	Eidgenössische Technische Hochschule Zürich
fcg	FortranCallGraph
FMS	Flexible Modeling System
FTG	FortranTestGenerator
GCM	General Circulation Model
GFDL	Geophysical Fluid Dynamics Laboratory (USA)

GPU	graphics processing unit
GUI	graphical user interface
HPC	High-Performance Computing
I/O	input/output
IPCC	Intergovernmental Panel on Climate Change
JAMSTEC	Japan Agency for Marine Earth Science and Technology
KIT	Karlsruher Institut für Technologie
LHS	left hand side
LOC	lines of code
MIP	Model Intercomparison Project
MPI	Message Passing Interface
MPI-M	Max-Planck-Institut für Meteorologie
MUT	module under test
NCAR	National Center for Atmospheric Research (USA)
NWP	Numerical Weather Prediction
OGCM	Oceanic General Circulation Model
PUT	procedure under test
R-CCS	RIKEN Center for Computational Science (Japan)
RHS	right hand side
SVN	Subversion
SWA	Arbeitsbereich Softwaretechnik und -architektur (Universität Hamburg)
TROPOS	Leibniz-Institut für Troposphärenforschung
WMO	World Meteorological Organisation
WR	Arbeitsbereich Wissenschaftliches Rechnen (Universität Hamburg)

Literatur

Abbe, 1901
: Abbe, Cleveland: „The Physical Basis of Long-Range Weather Forecasts“. In: *Monthly Weather Review*, Nr. 29.12 (1901), S. 551–561.

Akel u. a., 2013
: Akel, Chadi; Yuriy Kashnikov; Pablo de Oliveira Castro und William Jalby: „Is Source-Code Isolation Viable for Performance Characterization?“ In: *2013 42nd International Conference on Parallel Processing*. 2013, S. 977–984.

Alexander und Easterbrook, 2015
: Alexander, Kaitlin und Steve Easterbrook: „The software architecture of climate models: a graphical comparison of CMIP5 and EMICAR5 configurations“. In: *Geoscientific Model Development*, Nr. 8.4 (2015), S. 1221–1232.

Almasi u. a., 2017
: Almasi, M. Moein; Hadi Hemmati; Gordon Fraser; Andrea Arcuri und Janis Benefelds: „An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application“. In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 2017, S. 263–272.

Ammann und Offutt, 2016
: Ammann, Paul und Jeff Offutt: *Introduction to Software Testing*. 2nd Edition. New York, NY, USA: Cambridge University Press, 2016.

Anand u. a., 2013
: Anand, Saswat; Edmund K. Burke; Tsong Yueh Chen; John Clark; Myra B. Cohen; Wolfgang Grieskamp; Mark Harman; Mary Jean Harrold und Phil McMinn: „An orchestrated survey of methodologies for automated software test case generation“. In: *Journal of Systems and Software*, Nr. 86.8 (2013), S. 1978–2001.

Arrhenius, 1896
: Arrhenius, Svante: „On the influence of carbonic acid in the air upon the temperature of the ground“. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, Nr. 41.251 (1896), S. 237–276.

Bagnara u. a., 2013
: Bagnara, Roberto; Matthieu Carlier; Roberta Gori und Arnaud Gotlieb: „Symbolic Path-Oriented Test Data Generation for Floating-Point Programs“. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, S. 1–10.

Balaji, 2013
: Balaji, V.: „Scientific Computing in the Age of Complexity“. In: *XRDS*, Nr. 19.3 (2013), S. 12–17.

Balaji, Benson u. a., 2016
Balaji, V.; Rusty Benson; Bruce Wyman und Isaac Held: „Coarse-grained component concurrency in Earth system modeling: parallelizing atmospheric radiative transfer in the GFDL AM3 model using the Flexible Modeling System coupling framework“. In: *Geoscientific Model Development*, Nr. 9.10 (2016), S. 3605–3616.

Balaji, Maisonnave u. a., 2017
Balaji, V.; Eric Maisonnave; Niki Zadeh; Bryan N. Lawrence; Joachim Biercamp; Uwe Fladrich; Giovanni Aloisio; Rusty Benson; Arnaud Caubel; Jeffrey Durachta; Marie-Alice Foujols; Grenville Lister; Silvia Mocavero; Seth Underwood und Garrett Wright: „CPMIP: measurements of real computational performance of Earth system models in CMIP6“. In: *Geoscientific Model Development*, Nr. 10.1 (2017), S. 19–34.

Balaji, Redler und Budich, 2013
Balaji, V.; René Redler und Reinhard Budich: *Earth System Modelling - Volume 4. IO and Postprocessing*. Springer Briefs in Earth System Sciences. Berlin/Heidelberg: Springer-Verlag, 2013.

Barr u. a., 2015
Barr, Earl T.; Mark Harman; Phil McMinn; Muzammil Shahbaz und Shin Yoo: „The Oracle Problem in Software Testing: A Survey“. In: *IEEE Transactions on Software Engineering*, Nr. 41.5 (2015), S. 507–525.

Basili u. a., 2008
Basili, Victor R.; Jeffrey C. Carver; Daniela Cruzes; Lorin M. Hochstein; Jeffrey K. Hollinsworth; Forrest Shull und Marvin V. Zelkowitz: „Understanding the High-Performance-Computing Community: A Software Engineer's Perspective“. In: *IEEE Software*, Nr. 25.4 (2008), S. 29–36.

Baxter u. a., 1998
Baxter, Ira D.; Andrew Yahin; Leonardo Moura; Marcelo Sant'Anna und Lorraine Bier: „Clone detection using abstract syntax trees“. In: *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 1998, S. 368–377.

Behrens u. a., 2018
Behrens, Jörg; Joachim Biercamp; Hendryk Bockelmann und Philipp Neumann: „Increasing parallelism in climate models via additional component concurrency“. In: *2018 IEEE 14th International Conference on e-Science (e-Science)*. 2018, S. 271–271.

Beizer, 1990
Beizer, Boris: *Software Testing Techniques*. 2nd Edition. Boston, MA, USA: International Thomson Computer Press, 1990.

Bjerknes, 1904
Bjerknes, Vilhelm: „Das Problem der Wettervorhersage, betrachtet vom Standpunkte der Mechanik und der Physik“. In: *Meteorologische Zeitschrift*, Nr. 21 (1904), S. 1–7.

Blackburn und Hoskins, 2013
Blackburn, Michael und Brian J. Hoskins: „Context and Aims of the Aqua-Planet Experiment“. In: *Journal of the Meteorological Society of Japan. Ser. II*, Nr. 91A (2013), S. 1–15.

Bonaventura, Redler und Budich, 2012
Bonaventura, Luca; René Redler und Reinhard Budich: *Earth System Modelling - Volume 2. Algorithms, Code Infrastructure and Optimisation*. Springer Briefs in Earth System Sciences. Berlin/Heidelberg: Springer-Verlag, 2012.

Botella, Gotlieb und Claude, 2005
Botella, Bernard; Arnaud Gotlieb und Michel Claude: „Symbolic execution of floating-point computations“. In: *Software Testing, Verification and Reliability*, Nr. 16.2 (2005), S. 97–121.

Brainerd, 2009
Brainerd, Walter S.: *Guide to Fortran 2003 Programming*. London: Springer-Verlag, 2009.

Brunotte u. a., 2001
Brunotte, Ernst; Hans Gebhardt; Manfred Meurer; Peter Meusburger und Josef Nipper: *Lexikon der Geographie*. Bd. 1 (A bis Gasg.) Heidelberg/Berlin: Spektrum Akademischer Verlag, 2001.

Budde und Züllighoven, 1990
Budde, Reinhard und Heinz Züllighoven: *Software-Werkzeuge in einer Programmierwerkstatt: Ansätze eines hermeneutisch fundierten Werkzeug- und Maschinenbegriffs*. Bd. 182. GMD-Bericht. Dissertation. München: Oldenbourg, 1990.

buildbot.net
Buildbot. Software Freedom Conservancy. URL: https://buildbot.net (besucht am 02. 12. 2019).

Burgess, 1993
Burgess, C. J.: *Software Testing Using an Automatic Generator of Test Data*. Techn. Ber. Bristol, UK, 1993.

Callendar, 1938
Callendar, Guy Stewart: „The artificial production of carbon dioxide and its influence on temperature“. In: *Quarterly Journal of the Royal Meteorological Society*, Nr. 64.275 (1938), S. 223–240.

Carrassi u. a., 2018
Carrassi, Alberto; Marc Bocquet; Laurent Bertino und Geir Evensen: „Data assimilation in the geosciences: An overview of methods, issues, and perspectives“. In: *Wiley Interdisciplinary Reviews: Climate Change*, Nr. 9.5 (2018), e535.

Carver u. a., 2007
Carver, Jeffrey C.; Richard P. Kendall; Susan E. Squires und Douglass E. Post: „Software Development Environments for Scientific and Engineering Software: A Series of Case Studies“. In: *29th International Conference on Software Engineering (ICSE'07)*. 2007, S. 550–559.

Cassez u. a., 2017
Cassez, Franck; Anthony M. Sloane; Matthew Roberts; Matthew Pigram; Pongsak Suvanpong und Pablo Gonzalez de Aledo: „Skink: Static Analysis of Programs in LLVM Intermediate Representation“. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Hrsg. von Axel Legay und Tiziana Margaria. Berlin/Heidelberg: Springer-Verlag, 2017, S. 380–384.

Castro u. a., 2015
Castro, Pablo De Oliveira; Chadi Akel; Eric Petit; Mihail Popov und William Jalby: „CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization“. In: *ACM Transactions on Architecture and Code Optimization*, Nr. 12.1 (2015), 6:1–6:24.

Ceccato u. a., 2015
Ceccato, Mariano; Alessandro Marchetto; Leonardo Mariani; Cu D. Nguyen und Paolo Tonella: „Do Automatically Generated Test Cases Make Debugging Easier? An Experimental Assessment of Debugging Effectiveness and Efficiency“. In: *ACM Transactions on Software Engineering and Methodology*, Nr. 25.1 (2015), 5:1–5:38.

Chandy und Ramamoorthy, 1972
Chandy, K. M. und C. V. Ramamoorthy: „Rollback and Recovery Strategies for Computer Programs“. In: *IEEE Transactions on Computers*, Nr. C-21.6 (1972), S. 546–556.

Charney, Fjörtoft und Neumann, 1950
Charney, Jule Gregory; Ragnar Fjörtoft und John von Neumann: „Numerical Integration of the Barotropic Vorticity Equation“. In: *Tellus A*, Nr. 2.4 (1950).

Chen, Cheung und Yiu, 1998
Chen, T. Y.; S. C. Cheung und S. M. Yiu: *Metamorphic testing: a new approach for generating next test cases*. Techn. Ber. HKUST-CS98-01. Department of Computer Science, The Hong Kong University of Science und Technology, 1998.

Chen, Leung und Mak, 2005
Chen, T. Y.; H. Leung und I. K. Mak: „Adaptive Random Testing“. In: *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*. Hrsg. von Michael J. Maher. Berlin/Heidelberg: Springer-Verlag, 2005, S. 320–329.

Chivers und Sleightholme, 2018
Chivers, Ian und Jane Sleightholme: *Introduction to Programming with Fortran*. 4. Aufl. Cham, Schweiz: Springer International Publishing, 2018.

Chow, 1978
Chow, Tsun S.: „Testing Software Design Modeled by Finite-State Machines“. In: *IEEE Transactions on Software Engineering*, Nr. SE-4.3 (1978), S. 178–187.

Ciupa u. a., 2011
Ciupa, I.; A. Pretschner; M. Oriol; A. Leitner und B. Meyer: „On the number and nature of faults found by random testing“. In: *Software Testing, Verification and Reliability*, Nr. 21.1 (2011), S. 3–28.

Clune, Finkel und Rilee, 2015
Clune, Thomas L.; Hal Finkel und Michael Rilee: „Testing and Debugging Exascale Applications by Mocking MPI“. In: *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*. SE-HPCCSE '15. Austin, Texas: ACM, 2015, S. 5–8.

Clune und Rood, 2011
Clune, Thomas L. und Richard B. Rood: „Software Testing and Verification in Climate Model Development“. In: *IEEE Software*, Nr. 28.6 (2011), S. 49–55.

Cohen u. a., 1996
Cohen, David M.; Siddhartha R. Dalal; Jesse Parelius und Gardner C. Patton: „The combinatorial design approach to automatic test generation“. In: *IEEE Software*, Nr. 13.5 (1996), S. 83–88.

Coles u. a., 2016
Coles, Henry; Thomas Laurent; Christopher Henard; Mike Papadakis und Anthony Ventresque: „PIT: A Practical Mutation Testing Tool for Java (Demo)“. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. Saarbrücken: ACM, 2016, S. 449–452.

Coward, 1988
Coward, P. David: „Symbolic Execution Systems - a Review“. In: *Software Engineering Journal*, Nr. 3.6 (1988), S. 229–239.

cscs.ch, Piz Daint
Piz Daint. Centro Svizzero di Calculo Scientifico (CSCS), 2018. URL: https://www.cscs.ch/computers/piz-daint/ (besucht am 02. 12. 2019).

Cseppentő und Micskei, 2015
Cseppentő, Lajos und Zoltán Micskei: „Evaluating Symbolic Execution-Based Test Tools“. In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 2015, S. 1–10.

Cseppentő und Micskei, 2017
Cseppentő, Lajos und Zoltán Micskei: „Evaluating code-based test input generator tools“. In: *Software Testing, Verification and Reliability*, Nr. 27.6 (2017), e1627.

Czerwonka, 2018
Czerwonka, Jacek: *Pairwise Testing - Available Tools*. 2018. URL: http://www.pairwise.org/tools.asp (besucht am 02. 12. 2019).

Dee u. a., 2011
Dee, D. P.; S. M. Uppala; A. J. Simmons; P. Berrisford; P. Poli; S. Kobayashi; U. Andrae; M. A. Balmaseda; G. Balsamo; P. Bauer; P. Bechtold; A. C. M. Beljaars; L. van de Berg; J. Bidlot; N. Bormann; C. Delsol; R. Dragani; M. Fuentes; A. J. Geer; L. Haimberger; S. B. Healy; H. Hersbach; E. V. Hólm; L. Isaksen; P. Kållberg; M. Köhler; M. Matricardi; A. P. McNally; B. M. Monge-Sanz; J.-J. Morcrette; B.-K. Park; C. Peubey; P. de Rosnay; C. Tavolato; J.-N. Thépaut und F. Vitart: „The ERA-Interim reanalysis: configuration and performance of the data assimilation

system“. In: *Quarterly Journal of the Royal Meteorological Society*, Nr. 137.656 (2011), S. 553–597.

Demeshko u. a., 2013
Demeshko, Irina; Naoya Maruyama; Hirofumi Tomita und Satoshi Matsuoka: „Multi-GPU Implementation of the NICAM Atmospheric Model“. In: *Euro-Par 2012: Parallel Processing Workshops*. Hrsg. von Ioannis Caragiannis; Michael Alexander; Rosa Maria Badia; Mario Cannataro; Alexandru Costan; Marco Danelutto; Frédéric Desprez; Bettina Krammer; Julio Sahuquillo; Stephen L. Scott und Josef Weidendorfer. Bd. 7640. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer-Verlag, 2013, S. 175–184.

DeMillo, Lipton und Sayward, 1978
DeMillo, Richard A.; Richard J. Lipton und Frederick G. Sayward: „Hints on Test Data Selection: Help for the Practicing Programmer“. In: *Computer*, Nr. 11.4 (1978), S. 34–41.

Dennis u. a., 2012
Dennis, John M.; Mariana Vertenstein; Patrick H. Worley; Arthur A. Mirin; Anthony P. Craig; Robert Jacob und Sheri Mickelson: „Computational performance of ultra-high-resolution capability in the Community Earth System Model“. In: *The International Journal of High Performance Computing Applications*, Nr. 26.1 (2012), S. 5–16.

Dessler, 2011
Dessler, Andrew: *Introduction to Modern Climate Change*. Cambridge Books Online. Cambridge University Press, 2011.

Dijkstra, 1972
Dijkstra, Edsger W.: „The Humble Programmer“. In: *Communications of the ACM*, Nr. 15.10 (1972), S. 859–866.

DKRZ, 2016
Mistral: Endausbau des Hochleistungsrechners am DKRZ nimmt Betrieb auf. Deutsches Klimarechenzentrum (DKRZ), 2016. URL: https://www.dkrz.de/kommunikation/aktuelles/mistral-endausbau-des-hochleistungsrechners-am-dkrz-nimmt-betrieb-auf (besucht am 02. 12. 2019).

DKRZ, Configuration
Configuration. Deutsches Klimarechenzentrum (DKRZ). URL: https://www.dkrz.de/up/systems/mistral/configuration (besucht am 04. 12. 2019).

Donner u. a., 2011
Donner, Leo J.; Bruce L. Wyman; Richard S. Hemler; Larry W. Horowitz; Yi Ming; Ming Zhao; Jean-Christophe Golaz; Paul Ginoux; S.-J. Lin; M. Daniel Schwarzkopf; John Austin; Ghassan Alaka; William F. Cooke; Thomas L. Delworth; Stuart M. Freidenreich; C. T. Gordon; Stephen M. Griffies; Isaac M. Held; William J. Hurlin; Stephen A. Klein; Thomas R. Knutson; Amy R. Langenhorst; Hyun-Chul Lee; Yanluan Lin; Brian I. Magi; Sergey L. Malyshev; P. C. D. Milly; Vaishali Naik; Mary J. Nath; Robert Pincus; Jeffrey J. Ploshay; V. Ramaswamy; Charles J. Se-

man; Elena Shevliakova; Joseph J. Sirutis; William F. Stern; Ronald J. Stouffer; R. John Wilson; Michael Winton; Andrew T. Wittenberg und Fanrong Zeng: „The Dynamical Core, Physical Parameterizations, and Basic Simulation Characteristics of the Atmospheric Component AM3 of the GFDL Global Coupled Model CM3“. In: *Journal of Climate*, Nr. 24.13 (2011), S. 3484–3519.

Drake, Jones und Carr, 2005

Drake, John B.; Philip W. Jones und George R. Carr: „Overview of the Software Design of the Community Climate System Model“. In: *The International Journal of High Performance Computing Applications*, Nr. 19.3 (2005), S. 177–186.

Driessen, 2010

Driessen, Vincent: *A successful Git branching model*. 2010. URL: https://nvie.com/posts/a-successful-git-branching-model/ (besucht am 02. 12. 2019).

Droste, Kuhn und Ludwig, 2015

Droste, Alexander; Michael Kuhn und Thomas Ludwig: „MPI-checker: Static Analysis for MPI“. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: ACM, 2015, 3:1–3:10.

Duran und Ntafos, 1984

Duran, J. W. und S. C. Ntafos: „An Evaluation of Random Testing“. In: *IEEE Transactions on Software Engineering*, Nr. SE-10.4 (1984), S. 438–444.

DWD, 2015

Modellumstellung ICON. Deutscher Wetterdienst (DWD), 2015. URL: https://www.dwd.de/DE/fachnutzer/luftfahrt/teaser/aktuelles/08_icon/08_icon_artikel.html (besucht am 02. 12. 2019).

DWD, Klimawandel

Klimawandel - ein Überblick. Deutscher Wetterdienst (DWD). URL: http://www.dwd.de/DE/klimaumwelt/klimawandel/ueberblick/ueberblick_node.html (besucht am 02. 12. 2019).

Easterbrook, 2009

Easterbrook, Steve: *Serendipity – Applying systems thinking to computing, climate and sustainability*. Blog. 2009. URL: https://www.easterbrook.ca/steve (besucht am 02. 12. 2019).

Easterbrook, 2010a

Easterbrook, Steve: *Initial value vs. boundary value problems*. Blog Post. 2010. URL: https://www.easterbrook.ca/steve/2010/01/initial-value-vs-boundary-value-problems (besucht am 02. 12. 2019).

Easterbrook, 2010b

Easterbrook, Steve: *What makes software engineering for climate models different?* Blog Post. 2010. URL: http://www.easterbrook.ca/steve/2010/03/what-makes-software-engineering-for-climate-models-different (besucht am 02. 12. 2019).

Easterbrook und Johns, 2009

Easterbrook, Steve und Timothy C. Johns: „Engineering the Software for Under-

standing Climate Change“. In: *Computing in Science & Engineering*, Nr. 11 (2009), S. 65–74.

Edvardsson, 1999
Edvardsson, Jon: „A Survey on Automatic Test Data Generation“. In: *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*. 1999, S. 21–28.

Edwards, 2010
Edwards, Paul N.: *A Vast Machine: Computer Models, Climate Data, and the Politics of Global Warming*. Cambridge, USA/London, UK: The MIT Press, 2010.

Edwards, 2011
Edwards, Paul N.: „History of climate modeling“. In: *Wiley Interdisciplinary Reviews: Climate Change*, Nr. 2.1 (2011), S. 128–139.

Egwutuoha u. a., 2013
Egwutuoha, Ifeanyi P.; David Levy; Bran Selic und Shiping Chen: „A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems“. In: *The Journal of Supercomputing*, Nr. 65.3 (2013), S. 1302–1326.

Elbaum u. a., 2009
Elbaum, Sebastian; Hui Nee Chin; Matthew B. Dwyer und Matthew Jorde: „Carving and Replaying Differential Unit Test Cases from System Test Cases“. In: *IEEE Transactions on Software Engineering*, Nr. 35.1 (2009), S. 29–45.

Eyring u. a., 2016
Eyring, Veronika; Sandrine Bony; Gerald A. Meehl; Catherine A. Senior; Bjorn Stevens; Ronald J. Stouffer und Karl E. Taylor: „Overview of the Coupled Model Intercomparison Project Phase 6 (CMIP6) experimental design and organization“. In: *Geoscientific Model Development*, Nr. 9.5 (2016), S. 1937–1958.

f2py.com
Peterson, Pearu: *The F2PY Project*. URL: http://www.f2py.com (besucht am 02. 12. 2019).

Feathers, 2004
Feathers, Michael: *Working Effectively with Legacy Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

Fields, 2014
Fields, Jay: *Working Effectively with Unit Tests*. Leanpub, 2014.

Flato u. a., 2013
Flato, Gregory; Jochem Marotzke; Babatunde Abiodun; Pascale Braconnot; Sin Chan Chou; William J. Collins; Peter Cox; Fatima Driouech; Seita Emori; Veronika Eyring; Chris Forest; Peter Gleckler; Eric Guilyardi; Christian Jakob; Vladimir Kattsov; Chris Reason und Markku Rummukaines: „Evaluation of Climate Models“. In: *Climate Change 2013: The Physical Science Basis. Contribution of Working Group I to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change*. Hrsg. von T.F. Stocker; D. Qin; G.-K. Plattner; M. Tignor;

S.K. Allen; J. Boschung; A. Nauels; Y. Xia; V. Bex und P.M. Midgley. Cambridge, UK/New York, USA: Cambridge University Press, 2013, S. 741–866.

Folk u. a., 2011
Folk, Mike; Gerd Heber; Quincey Koziol; Elena Pourmal und Dana Robinson: „An Overview of the HDF5 Technology Suite and Its Applications“. In: *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. Uppsala, Sweden: ACM, 2011, S. 36–47.

Ford u. a., 2012
Ford, Rupert; Graham Riley; René Redler und Reinhard Budich: *Earth System Modelling - Volume 5. Tools for Configuring, Building and Running Models*. Springer Briefs in Earth System Sciences. Berlin/Heidelberg: Springer-Verlag, 2012.

Fortran Wiki, 2011
fpp. Fortran Wiki, 2011. URL: http://fortranwiki.org/fortran/show/fpp (besucht am 02. 12. 2019).

Fortran Wiki, 2018
Unit testing frameworks. Fortran Wiki, 2018. URL: http://fortranwiki.org/fortran/show/Unit+testing+frameworks (besucht am 02. 12. 2019).

Fortran Wiki, 2019a
Fortran 2003 status. Fortran Wiki, 2019. URL: http://fortranwiki.org/fortran/show/Fortran+2003+status (besucht am 02. 12. 2019).

Fortran Wiki, 2019b
Fortran 2008 status. Fortran Wiki, 2019. URL: http://fortranwiki.org/fortran/show/Fortran+2008+status (besucht am 02. 12. 2019).

Fourier, 1822
Fourier, Jean Baptiste Joseph: *Th eorie analytique de la chaleur*. Paris, Frankreich: Firmin Didot, 1822.

Fowler, 1999
Fowler, Martin: *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

Fowler, 2006
Fowler, Martin: *Continuous Integration*. Blog Post. 2006. URL: https://martinfowler.com/articles/continuousIntegration.html (besucht am 07. 12. 2019).

Fowler, 2007
Fowler, Martin: *Mocks Aren't Stubs*. Blog Post. 2007. URL: https://martinfowler.com/articles/mocksArentStubs.html (besucht am 02. 12. 2019).

Fowler, 2010
Fowler, Martin: *Domain Specific Languages*. Boston, MA, USA: Addison-Wesley Professional, 2010.

Fowler, 2014
Fowler, Martin: *UnitTest*. Blog Post. 2014. URL: http://martinfowler.com/bliki/UnitTest.html (besucht am 07. 12. 2019).

Frankl und Iakounenko, 1998
Frankl, Phyllis G. und Oleg Iakounenko: „Further Empirical Studies of Test Effectiveness“. In: *SIGSOFT Software Engineering Notes*, Nr. 23.6 (1998), S. 153–162.

Fraser und Arcuri, 2011
Fraser, Gordon und Andrea Arcuri: „EvoSuite: Automatic Test Suite Generation for Object-oriented Software“. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: ACM, 2011, S. 416–419.

Fraser und Arcuri, 2014
Fraser, Gordon und Andrea Arcuri: „A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite“. In: *ACM Transactions on Software Engineering and Methodology*, Nr. 24.2 (2014), 8:1–8:42.

Fraser, Staats u. a., 2013
Fraser, Gordon; Matt Staats; Phil McMinn; Andrea Arcuri und Frank Padberg: „Does Automated White-box Test Generation Really Help Software Testers?“ In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. Lugano, Switzerland: ACM, 2013, S. 291–301.

Fraser und Zeller, 2011
Fraser, Gordon und Andreas Zeller: „Exploiting Common Object Usage in Test Case Generation“. In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. 2011, S. 80–89.

Fu und Su, 2017
Fu, Zhoulai und Zhendong Su: „Achieving High Coverage for Floating-point Code via Unconstrained Programming“. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, S. 306–319.

Galler und Aichernig, 2014
Galler, Stefan J. und Bernhard K. Aichernig: „Survey on test data generation tools“. In: *International Journal on Software Tools for Technology Transfer*, Nr. 16.6 (2014), S. 727–751.

Garousi und Mäntylä, 2016
Garousi, Vahid und Mika V. Mäntylä: „A systematic literature review of literature reviews in software testing“. In: *Information and Software Technology*, Nr. 80 (2016), S. 195–216.

GISTEMP, 2019
GISS Surface Temperature Analysis – Additional Analysis Plots. NASA Goddard Institute for Space Studies (GISS), 2019. URL: https://data.giss.nasa.gov/gistemp/graphs_v3/customize.html (besucht am 03. 12. 2019).

git-scm.com
git. Software Freedom Conservancy. URL: https://git-scm.com (besucht am 03. 12. 2019).

GitHub, CESM
Fischer, Chris u. a.: *GitHub: The Community Earth System Model*. The Earth System Community Modeling Portal (ESCOMP), 2019. URL: https://github.com/ESCOMP/cesm (besucht am 03. 12. 2019).

GitHub, Cheetah3
Broytman, Oleg: *GitHub: Cheetah3*. 2017. URL: https://github.com/CheetahTemplate3/cheetah3 (besucht am 03. 12. 2019).

GitHub, cloc
Danial, Al: *GitHub: cloc*. 2019. URL: https://github.com/AlDanial/cloc (besucht am 03. 12. 2019).

GitHub, fcg-Issues
Hovy, Christian u. a.: *GitHub: fortrancallgraph – Issues*. 2019. URL: https://github.com/fortesg/fortrancallgraph/issues (besucht am 03. 12. 2019).

GitHub, Flang
Klimowicz, Gary u. a.: *GitHub: Flang*. 2017. URL: https://github.com/flang-compiler/flang (besucht am 03. 12. 2019).

GitHub, fortesg
Hovy, Christian: *GitHub: FortranTestGenerator*. 2017. URL: https://github.com/fortesg (besucht am 03. 12. 2019).

GitHub, FTG-Issues
Hovy, Christian u. a.: *GitHub: fortrantestgenerator – Issues*. 2018. URL: https://github.com/fortesg/fortrantestgenerator/issues (besucht am 03. 12. 2019).

GitHub, jsbach-mock
Hovy, Christian: *GitHub: jsbach-mock*. 2019. URL: https://github.com/fortesg/jsbach-mock (besucht am 03. 12. 2019).

GitHub, KGEN
Kim, Youngsung: *GitHub: KGEN: Fortran Kernel Generator*. NCAR, 2015. URL: https://github.com/NCAR/KGen (besucht am 03. 12. 2019).

GitHub, LCOV
Oberparleiter, Peter u. a.: *GitHub: LCOV*. Linux Test Project, 2016. URL: https://github.com/linux-test-project/lcov (besucht am 03. 12. 2019).

GitHub, MOM6
Adcroft, Alistair u. a.: *GitHub: MOM6*. NOAA-GFDL, 2019. URL: https://github.com/NOAA-GFDL/MOM6 (besucht am 03. 12. 2019).

GitHub, MOM6-examples
Adcroft, Alistair u. a.: *GitHub: MOM6-examples*. NOAA-GFDL, 2019. URL: https://github.com/NOAA-GFDL/MOM6-examples (besucht am 03. 12. 2019).

GitHub, SB2
Thüring, Fabian; Hannes Vogt u. a.: *GitHub: Serialbox2*. ETH/CSCS, 2016. URL: https://github.com/GridTools/serialbox (besucht am 03. 12. 2019).

GitHub, SB2-PR134
Hovy, Christian: *GitHub: Serialbox2 - Pull Request #134: [Fortran] FortranTest-*

Generator frontend. ETH/CSCS, 2018. URL: https://github.com/GridTools/serialbox/pull/134 (besucht am 03. 12. 2019).

gitlab.com

GitLab. GitLab Inc. URL: https://about.gitlab.com (besucht am 03. 12. 2019).

Gleckler, Taylor und Doutriaux, 2008

Gleckler, P. J.; K. E. Taylor und C. Doutriaux: „Performance metrics for climate models". In: *Journal of Geophysical Research: Atmospheres*, Nr. 113.D6 (2008).

GNU, gcov

gcov—a Test Coverage Program. Free Software Foundation, Inc. URL: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html (besucht am 03. 12. 2019).

GNU, GFortran

Welcome to the home of GNU Fortran. Free Software Foundation, Inc., 2018. URL: https://gcc.gnu.org/fortran/ (besucht am 03. 12. 2019).

Goosse, 2015

Goosse, Hugues: *Climate system dynamics and modelling*. New York, USA: Cambridge University Press, 2015.

Gopinath, Jensen und Groce, 2014

Gopinath, Rahul; Carlos Jensen und Alex Groce: „Code Coverage for Suite Evaluation by Developers". In: *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. Hyderabad, India: ACM, 2014, S. 72–82.

Govett, Middlecoff und Henderson, 2010

Govett, Mark W.; Jacques Middlecoff und Tom Henderson: „Running the NIM Next-Generation Weather Model on GPUs". In: *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. CCGRID '10. Washington, DC, USA: IEEE Computer Society, 2010, S. 792–796.

Grech u. a., 2015

Grech, Neville; Kyriakos Georgiou; James Pallister; Steve Kerrison; Jeremy Morse und Kerstin Eder: „Static Analysis of Energy Consumption for LLVM IR Programs". In: *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems (SCOPES '15)*. Sankt Goar: ACM, 2015, S. 12–21.

Guillemot, 2010

Guillemot, Hélène: „Connections between simulations and observation in climate computer modeling. Scientist's practices and 'bottom-up epistemology' lessons". In: *Studies in History and Philosophy of Science Part B: Studies in History and Philosophy of Modern Physics*, Nr. 41.3 (2010). Special Issue: Modelling and Simulation in the Atmospheric and Climate Sciences, S. 242–252.

Haarsma u. a., 2016

Haarsma, Reindert J.; Malcolm J. Roberts; Pier Luigi Vidale; Catherine A. Senior; Alessio Bellucci; Qing Bao; Ping Chang; Susanna Corti; Neven S. Fučkar; Virginie Guemas; Jost von Hardenberg; Wilco Hazeleger; Chihiro Kodama; Torben Koenigk; L. Ruby Leung; Jian Lu; Jing-Jia Luo; Jiafu Mao; Matthew S. Mizielinski; Ryo Mizuta; Paulo Nobre; Masaki Satoh; Enrico Scoccimarro; Tido Semmler;

Justin Small und Jin-Song von Storch: „High Resolution Model Intercomparison Project (HighResMIP v1.0) for CMIP6“. In: *Geoscientific Model Development*, Nr. 9.11 (2016), S. 4185–4208.

Hamlet, 2002

Hamlet, Richard: „Random Testing“. In: *Encyclopedia of Software Engineering*. American Cancer Society, 2002.

Hannah u. a., 2015

Hannah, Nicholas; Alistair Adcroft; Robert Hallberg und Stephen Griffies: „Reproducibility and Transparency in Ocean-Climate Modeling“. In: *AGU Fall Meeting 2015*. Poster. 2015.

Hantel, 2013

Hantel, Michael: *Einführung Theoretische Meteorologie*. Berlin: Springer-Verlag, 2013.

Harris, 2014

Harris, Mark: *A Brief History of GPGPU*. Vortragsfolien. 2014. URL: https://www.cs.unc.edu/xcms/wpfiles/50th-symp/Harris.pdf (besucht am 08. 12. 2019).

Hartin u. a., 2015

Hartin, C. A.; P. Patel; A. Schwarber; R. P. Link und B. P. Bond-Lamberty: „A simple object-oriented and open-source model for scientific and policy analyses of the global climate system – Hector v1.0“. In: *Geoscientific Model Development*, Nr. 8.4 (2015), S. 939–955.

Hausfather, 2016

Hausfather, Zeke: *SimMod: A simple python based climate model*. Berkeley Earth, 2016. URL: http://berkeleyearth.org/simmod-a-simple-python-based-climate-model (besucht am 03. 12. 2019).

Heavens, Ward und Mahowald, 2013

Heavens, Nicholas G.; Daniel S. Ward und Natalie M. Mahowald: „Studying and Projecting Climate Change with Earth System Models“. In: *Nature Education Knowledge*, Nr. 4.5 (2013).

Heing-Becker, 2017

Heing-Becker, Marcel: „Capture & Replay für Fortran- und MPI-Programme“. Projektarbeit. Hamburg: Universität Hamburg, 2017.

Held und Suarez, 1994

Held, Isaac M. und Max J. Suarez: „A Proposal for the Intercomparison of the Dynamical Cores of Atmospheric General Circulation Models“. In: *Bulletin of the American Meteorological Society*, Nr. 75.10 (1994), S. 1825–1830.

Hemsoth, 2016

Hemsoth, Nicole: *A Closer Look at 2016 Top 500 Supercomputer Rankings*. Deutscher Wetterdienst (DWD), 2016. URL: https://www.nextplatform.com/2016/11/14/closer-look-2016-top-500-supercomputer-rankings (besucht am 03. 12. 2019).

Hess, Battisti und Rasch, 1993

Hess, Peter G.; David S. Battisti und Philip J. Rasch: „Maintenance of the Inter-

tropical Convergence Zones and the Large-Scale Tropical Circulation on a Water-covered Earth“. In: *Journal of the Atmospheric Sciences*, Nr. 50.5 (1993), S. 691–713.

Hetzel, 1988
Hetzel, Bill: *The Complete Guide to Software Testing*. 2nd Edition. New York, USA: John Wiley & Sons, Inc, 1988.

Hicinbothom, 1996
Hicinbothom, James H.: „Instrumented Interface Construction Evaluator“. In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, Nr. 40.17 (1996), S. 863–863.

Hicinbothom und Zachary, 1993
Hicinbothom, James H. und Wayne W. Zachary: „A Tool for Automatically Generating Transcripts of Human-Computer Interaction“. In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, Nr. 37.15 (1993), S. 1042–1042.

Hildebrand, 1990
Hildebrand, Knut: *Software Tools: Automatisierung im Software Engineering*. Berlin/Heidelberg: Springer-Verlag, 1990.

Hill u. a., 2004
Hill, Chris; Cecelia DeLuca; V. Balaji; Max Suarez und Arlindo Da Silva: „The architecture of the Earth System Modeling Framework“. In: *Computing in Science & Engineering*, Nr. 6.1 (2004), S. 18–28.

Hinsen, 2015
Hinsen, Konrad: „The Approximation Tower in Computational Science: Why Testing Scientific Software Is Difficult“. In: *Computing in Science & Engineering*, Nr. 17.4 (2015), S. 72–77.

Hoffman, 1998
Hoffman, Douglas: „A Taxonomy for Test Oracles“. In: *11th International Software Quality Week*. San Francisco, CA, USA, 1998.

Hovy und Kunkel, 2016
Hovy, Christian und Julian Kunkel: „Towards Automatic and Flexible Unit Test Generation for Legacy HPC Code“. In: *2016 Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE)*. 2016, S. 1–8.

Hovy u. a., einger. 2019
Hovy, Christian; Tomohiro Hajima; Luis Kornblueh; Seth Underwood und Hisashi Yashiro: „A Qualitative Study on Software Testing Practices in Climate Model Development“. 2019 eingereicht; nicht angenommen. Aktuell unveröffentlicht.

Hurrell u. a., 2013
Hurrell, James W.; M. M. Holland; P. R. Gent; S. Ghan; Jennifer E. Kay; P. J. Kushner; J.-F. Lamarque; W. G. Large; D. Lawrence; K. Lindsay; W. H. Lipscomb; M. C. Long; N. Mahowald; D. R. Marsh; R. B. Neale; P. Rasch; S. Vavrus; M. Vertenstein; D. Bader; W. D. Collins; J. J. Hack; J. Kiehl und S. Marshall: „The

Community Earth System Model: A Framework for Collaborative Research“. In: *Bulletin of the American Meteorological Society*, Nr. 94.9 (2013), S. 1339–1360.

Hutchins u. a., 1994

Hutchins, Monica; Herb Foster; Tarak Goradia und Thomas Ostrand: „Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria“. In: *Proceedings of 16th International Conference on Software Engineering*. 1994, S. 191–200.

Ince, 1987

Ince, D. C.: „The Automatic Generation of Test Data“. In: *The Computer Journal*, Nr. 30.1 (1987), S. 63–69.

IPCC, 2007

Solomon, S.; D. Qin; M. Manning; Z. Chen; M. Marquis; K. B. Averyt; M. Tignor und H. L. Miller: *Climate Change 2007: The Physical Science Basis. Contribution of Working Group I to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change*. Cambridge, UK/New York, USA: Cambridge University Press, 2007.

IPCC, 2013

Stocker, T.F.; D. Qin; G.-K. Plattner; M. Tignor; S.K. Allen; J. Boschung; A. Nauels; Y. Xia; V. Bex und P.M. Midgley: *Climate Change 2013: The Physical Science Basis. Contribution of Working Group I to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change*. Cambridge, UK/New York, USA: Cambridge University Press, 2013.

ipcc.ch

ipcc.ch. Intergovernmental Panel on Climate Change. URL: https://www.ipcc.ch (besucht am 03. 12. 2019).

ISO/IEC DIS 1539-1, 2017

Fortran 2018 Draft International Standard for Ballot (ISO/IEC JTC1/SC22/WG5 N2146) – ISO/IEC DIS 1539-1. Information technology — Programming languages — Fortran — Part 1: Base language. Genf, Schweiz: ISO/IEC, 2017.

Jablonowski, 1998

Jablonowski, Christiane: „Test der Dynamik zweier globaler Wettervorhersagemodelle des Deutschen Wetterdienstes: Der Held-Suarez Test“. Diplomarbeit. Universität Bonn, 1998.

Jablonowski und Williamson, 2006a

Jablonowski, Christiane und David L. Williamson: „A baroclinic instability test case for atmospheric model dynamical cores“. In: *Quarterly Journal of the Royal Meteorological Society*, Nr. 132.621C (2006), S. 2943–2975.

Jablonowski und Williamson, 2006b

Jablonowski, Christiane und David L. Williamson: *A baroclinic wave test case for dynamical cores of general circulation models: Model intercomparisons*. Techn. Ber. Boulder, USA: NCAR, 2006.

jenkins.io
Jenkins. Software in the Public Interest. URL: https://jenkins.io (besucht am 03. 12. 2019).

Jia und Harman, 2011
Jia, Y. und M. Harman: „An Analysis and Survey of the Development of Mutation Testing“. In: *IEEE Transactions on Software Engineering*, Nr. 37.5 (2011), S. 649–678.

Joshi und Orso, 2007
Joshi, Shrinivas und Alessandro Orso: „SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions“. In: *2007 IEEE International Conference on Software Maintenance*. 2007, S. 234–243.

Just, Jalali und Ernst, 2014
Just, René; Darioush Jalali und Michael D. Ernst: „Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs“. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. San Jose, CA, USA: ACM, 2014, S. 437–440.

Kalnay u. a., 1996
Kalnay, E.; M. Kanamitsu; R. Kistler; W. Collins; D. Deaven; L. Gandin; M. Iredell; S. Saha; G. White; J. Woollen; Y. Zhu; A. Leetmaa; R. Reynolds; M. Chelliah; W. Ebisuzaki; W. Higgins; J. Janowiak; K. C. Mo; C. Ropelewski; J. Wang; Roy Jenne und Dennis Joseph: „The NCEP/NCAR 40-Year Reanalysis Project“. In: *Bulletin of the American Meteorological Society*, Nr. 77.3 (1996), S. 437–471.

Kanewala und Bieman, 2014
Kanewala, Upulee und James M. Bieman: „Testing scientific software: A systematic literature review“. In: *Information and Software Technology*, Nr. 56.10 (2014), S. 1219–1232.

Kapus und Cadar, 2017
Kapus, Timotej und Cristian Cadar: „Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing“. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. Urbana-Champaign, IL, USA: IEEE Press, 2017, S. 590–600.

Kelly und Sanders, 2008
Kelly, Diane und Rebecca Sanders: „The challenge of testing scientific software“. In: *Proceedings of the 3rd Annual Conference of the Association for Software Testing (CAST)*. 2008, S. 30–36.

Kellyeh, 2018
Kellyeh, Tareq: „Enabling Single Process Testing of MPI in Massive Parallel Applications“. Masterarbeit. Hamburg: Universität Hamburg, 2018.

Kim u. a., 2016
Kim, Youngsung; John Dennis; Christopher Kerr; Raghu Raj Prasanna Kumar; Amogh Simha; Allison Baker und Sheri Mickelson: „KGEN: A Python Tool for Automated Fortran Kernel Generation and Verification“. In: *ICCS 2016. The Interna-*

tional Conference on Computational Science. Bd. 80. Procedia Computer Science. San Diego, CA, USA, 2016, S. 1450–1460.

Kindratenko u. a., 2009
Kindratenko, Volodymyr V.; Jeremy J. Enos; Guochun Shi; Michael T. Showerman; Galen W. Arnold; John E. Stone; James C. Phillips und Wen-mei Hwu: „GPU clusters for high-performance computing“. In: *2009 IEEE International Conference on Cluster Computing and Workshops*. 2009, S. 1–8.

King, 1975
King, James C.: „A New Approach to Program Testing“. In: *Proceedings of the International Conference on Reliable Software*. Los Angeles, California: ACM, 1975, S. 228–233.

King und Offutt, 1991
King, K. N. und A. Jefferson Offutt: „A fortran language system for mutation-based software testing“. In: *Software: Practice and Experience*, Nr. 21.7 (1991), S. 685–718.

Klammer und Kern, 2015
Klammer, Claus und Albin Kern: „Writing unit tests: It's now or never!“ In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2015, S. 1–4.

Korel, 1990
Korel, B.: „Automated software test data generation“. In: *IEEE Transactions on Software Engineering*, Nr. 16.8 (1990), S. 870–879.

Kuhn u. a., 2009
Kuhn, Rick; Raghu Kacker; Yu Lei und Justin Hunter: „Combinatorial Software Testing“. In: *Computer*, Nr. 42.8 (2009), S. 94–96.

Lattner und Adve, 2004
Lattner, Chris und Vikram Adve: „LLVM: a compilation framework for lifelong program analysis & transformation“. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, S. 75–86.

Lauritzen, Nair u. a., 2018
Lauritzen, P. H.; R. D. Nair; A. R. Herrington; P. Callaghan; S. Goldhaber; J. M. Dennis; J. T. Bacmeister; B. E. Eaton; C. M. Zarzycki; Mark A. Taylor; P. A. Ullrich; T. Dubos; A. Gettelman; R. B. Neale; B. Dobbins; K. A. Reed; C. Hannay; B. Medeiros; J. J. Benedict und J. J. Tribbia: „NCAR Release of CAM-SE in CESM2.0: A Reformulation of the Spectral Element Dynamical Core in Dry-Mass Vertical Coordinates With Comprehensive Treatment of Condensates and Energy“. In: *Journal of Advances in Modeling Earth Systems*, Nr. 10.7 (2018), S. 1537–1570.

Lauritzen und Goldhaber, 2017
Lauritzen, Peter Hjort und Steve Goldhaber: *Moist baroclinic wave with Kessler microphysics*. 2017. URL: http://www.cesm.ucar.edu/models/simpler-models-indev/fkessler/index.html (besucht am 03. 12. 2019).

Lawrence, Rezny u. a., 2018
Lawrence, Bryan N.; Michael Rezny; Reinhard Budich; Peter Bauer; Jörg Behrens; Mick Carter; Willem Deconinck; Rupert Ford; Christopher Maynard; Steven Mullerworth; Carlos Osuna; Andrew Porter; Kim Serradell; Sophie Valcke; Nils Wedi und Simon Wilson: „Crossing the chasm: how to develop weather and climate models for next generation computers?“ In: *Geoscientific Model Development*, Nr. 11.5 (2018), S. 1799–1821.

Lawrence, Oleson u. a., 2011
Lawrence, David M; Keith W Oleson; Mark G Flanner; Peter E Thornton; Sean C Swenson; Peter J Lawrence; Xubin Zeng; Zong-Liang Yang; Samuel Levis; Koichi Sakaguchi; Gordon B Bonan und Andrew G Slater: „Parameterization improvements and functional and structural advances in Version 4 of the Community Land Model“. In: *Journal of Advances in Modeling Earth Systems*, Nr. 3.1 (2011).

Lee und Hall, 2005
Lee, Yoon-Ju und Mary Hall: „A Code Isolator: Isolating Code Fragments from Large Programs“. In: *Languages and Compilers for High Performance Computing: 17th International Workshop, LCPC 2004, West Lafayette, IN, USA, September 22-24, 2004, Revised Selected Papers*. Hrsg. von Rudolf Eigenmann; Zhiyuan Li und Samuel P. Midkiff. Berlin/Heidelberg: Springer-Verlag, 2005, S. 164–178.

Lill und Saglietti, 2012
Lill, R. und F. Saglietti: „Model-based testing of autonomous systems based on Coloured Petri Nets“. In: *ARCS 2012*. 2012, S. 1–5.

Lin und Rood, 1997
Lin, Shian-Jiann und Richard B. Rood: „An explicit flux-form semi-lagrangian shallow-water model on the sphere“. In: *Quarterly Journal of the Royal Meteorological Society*, Nr. 123.544 (1997), S. 2477–2498.

lstc.com, LS-DYNA
LS-DYNA. Livermore Software Technology Corporation, 2018. URL: http://www.lstc.com/products/ls-dyna (besucht am 03. 12. 2019).

Lynch, 2008
Lynch, Peter: „The origins of computer weather prediction and climate modeling“. In: *Journal of Computational Physics*, Nr. 227.7 (2008). Predicting weather, climate and extreme events, S. 3431–3444.

Lynch, 2016
Lynch, Peter: „An artist's impression of Richardson's fantastic forecastfactory“. In: *Weather*, Nr. 71.1 (2016), S. 14–18.

Mahadik und Thakore, 2016
Mahadik, Pranali Prakash und D. M. Thakore: „Survey on Automatic Test Data Generation Tools and Techniques for Object Oriented Code“. In: *International Journal of Innovative Research in Computer and Communication Engineering*, Nr. 4.1 (2016), S. 357–364.

Manabe und Bryan, 1969
Manabe, Syukuro und Kirk Bryan: „Climate Calculations with a Combined Ocean-Atmosphere Model“. In: *Journal of the Atmospheric Sciences*, Nr. 26.4 (1969), S. 786–789.

McCabe, 1976
McCabe, T. J.: „A Complexity Measure“. In: *IEEE Transactions on Software Engineering*, Nr. SE-2.4 (1976), S. 308–320.

McGuffie und Henderson-Sellers, 2005
McGuffie, Kendal und Ann Henderson-Sellers: *A Climate Modelling Primer*. 3rd Edition. Chichester, UK: John Wiley & Sons, 2005.

McMinn, 2011
McMinn, P.: „Search-Based Software Testing: Past, Present and Future“. In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 2011, S. 153–163.

McMinn, Stevenson und Harman, 2010
McMinn, Phil; Mark Stevenson und Mark Harman: „Reducing Qualitative Human Oracle Costs Associated with Automatically Generated Test Data“. In: *Proceedings of the First International Workshop on Software Test Output Validation*. STOV '10. Trento, Italy: ACM, 2010, S. 1–4.

Méndez, Tinetti und Overbey, 2014
Méndez, Mariano; Fernando G. Tinetti und Jeffrey L. Overbey: „Climate Models: Challenges for Fortran Development Tools“. In: *2014 Second International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*. 2014, S. 6–12.

mercurial-scm.org
mercurial. URL: https://www.mercurial-scm.org (besucht am 03. 12. 2019).

Meszaros, 2006
Meszaros, Gerard: *XUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ, USA: Addison-Wesley, 2006.

Metcalf, 2011
Metcalf, Michael: „The seven ages of Fortran“. In: *Journal of Computer Science & Technology*, Nr. 11.1 (2011), S. 1–8.

MeteoSchweiz, 2016
Das neue Wettervorhersagemodell für den Alpenraum. Bundesamt für Meteorologie und Klimatologie MeteoSchweiz, 2016. URL: http://www.meteoschweiz.admin.ch/home/suche.subpage.html/de/data/news/2016/3/das-neue-wettervorhersagemodell-fuer-den-alpenraum.html (besucht am 03. 12. 2019).

Michel, Rueher und Lebbah, 2001
Michel, C.; M. Rueher und Y. Lebbah: „Solving Constraints over Floating-Point Numbers“. In: *Principles and Practice of Constraint Programming — CP 2001*. Hrsg. von Toby Walsh. Berlin/Heidelberg: Springer-Verlag, 2001, S. 524–538.

Miller und Spooner, 1976
Miller, Webb und David L. Spooner: „Automatic Generation of Floating-Point Test Data“. In: *IEEE Transactions on Software Engineering*, Nr. SE-2.3 (1976), S. 223–226.

Millman und Aivazis, 2011
Millman, K. Jarrod und Michael Aivazis: „Python for Scientists and Engineers“. In: *Computing in Science & Engineering*, Nr. 13.2 (2011), S. 9–12.

Moss u. a., 2010
Moss, Richard H.; Jae A. Edmonds; Kathy A. Hibbard; Martin R. Manning; Steven K. Rose; Detlef P. van Vuuren; Timothy R. Carter; Seita Emori; Mikiko Kainuma; Tom Kram; Gerald A. Meehl; John F. B. Mitchell; Nebojsa Nakicenovic; Keywan Riahi; Steven J. Smith; Ronald J. Stouffer; Allison M. Thomson; John P. Weyant und Thomas J. Wilbanks: „The next generation of scenarios for climate change research and assessment“. In: *Nature*, Nr. 463 (2010), S. 747–756.

MPI-M, Obtain ICON
How to obtain a copy of the ICON model code. Max-Planck Institut für Meteorologie. URL: https://code.mpimet.mpg.de/projects/iconpublic/wiki/How%20to%20obtain%20the%20model%20code (besucht am 02. 12. 2019).

MPI, 2015
MPI: A Message-Passing Interface Standard – Version 3.1. Message Passing Interface Forum. 2015.

mpich.org
MPICH – High-Performance Portable MPI. Argonne National Laboratory. URL: https://www.mpich.org (besucht am 05. 12. 2019).

Müller, 2010
Müller, Peter: „Constructing climate knowledge with computer models“. In: *Wiley Interdisciplinary Reviews: Climate Change*, Nr. 1.4 (2010), S. 565–580.

Myers, Sandler und Badgett, 2011
Myers, Glenford J.; Corey Sandler und Tom Badgett: *The Art of Software Testing*. 3rd Edition. Wiley Publishing, 2011.

Neamtiu, Foster und Hicks, 2005
Neamtiu, Iulian; Jeffrey S. Foster und Michael Hicks: „Understanding Source Code Evolution Using Abstract Syntax Tree Matching“. In: *Proceedings of the 2005 International Workshop on Mining Software Repositories*. MSR '05. St. Louis, Missouri: ACM, 2005, S. 1–5.

Nebeker, 1995
Nebeker, Frederik: *Calculating the Weather: Meteorology in the 20th Century*. Bd. 60. International Geophysics. San Diego, USA: Academic Press, 1995.

Neumann, 2016
Neumann, Alexander: *Vor 60 Jahren: IBM veröffentlicht erste Sprachspezifikation für Fortran*. heise Developer, 2016. URL: https://www.heise.de/newsticker/

meldung/Vor-60-Jahren-IBM-veroeffentlicht-erste-Sprachspezifikation-fuer-Fortran-3351318.html (besucht am 03. 12. 2019).

Neumann, 2018
Neumann, Alexander: *Programmiersprache: Fortran 2018 veröffentlicht*. heise Developer, 2018. URL: https://www.heise.de/developer/meldung/Programmiersprache-Fortran-2018-veroeffentlicht-4241463.html (besucht am 03. 12. 2019).

Neumann, 1945
Neumann, John von: *First Draft of a Report on the EDVAC*. Techn. Ber. 1945.

NICAM, Collaborations
NICAM – Research Collaborations. NICAM development team, 2014. URL: http://nicam.jp/hiki/?Research+Collaborations (besucht am 03. 12. 2019).

Nishizawa u. a., 2015
Nishizawa, S.; H. Yashiro; Y. Sato; Y. Miyamoto und H. Tomita: „Influence of grid aspect ratio on planetary boundary layer turbulence in large-eddy simulations". In: *Geoscientific Model Development*, Nr. 8.10 (2015), S. 3393–3419.

Norman u. a., 2015
Norman, Matthew; Jeffrey Larkin; Aaron Vose und Katherine Evans: „A case study of CUDA FORTRAN and OpenACC for an atmospheric climate kernel". In: *Journal of Computational Science*, Nr. 9 (2015), S. 1–6.

NRC, 1979
Carbon Dioxide and Climate: A Scientific Assessment. Washington, DC: National Research Council, 1979.

Offutt, 1991
Offutt, A. Jefferson: „An integrated automatic test data generation system". In: *Journal of Systems Integration*, Nr. 1.3 (1991), S. 391–409.

Oliveira, Kanewala und Nardi, 2014
Oliveira, Rafael A.P.; Upulee Kanewala und Paulo A. Nardi: „Automated Test Oracles: State of the Art, Taxonomies, and Trends". In: Hrsg. von Atif Memon. Bd. 95. Advances in Computers. Elsevier, 2014, S. 113–199.

OpenMP, 2015
OpenMP Application Programming Interface (Version 4.5). OpenMP Architecture Review Board. 2015.

openmpi.org
Open MPI: Open Source High Performance Computing. Software in the Public Interest, Inc. URL: https://www.open-mpi.org (besucht am 05. 12. 2019).

Orso und Kennedy, 2005
Orso, Alessandro und Bryan Kennedy: „Selective Capture and Replay of Program Executions". In: *SIGSOFT Software Engineering Notes*, Nr. 30.4 (2005), S. 1–7.

Palomba, Di Nucci u. a., 2016
Palomba, Fabio; Dario Di Nucci; Annibale Panichella; Rocco Oliveto und Andrea De Lucia: „On the Diffusion of Test Smells in Automatically Generated Test Code:

An Empirical Study“. In: *Proceedings of the 9th International Workshop on Search-Based Software Testing*. SBST ’16. Austin, Texas: ACM, 2016, S. 5–14.

Palomba, Panichella u. a., 2016

Palomba, Fabio; Annibale Panichella; Andy Zaidman; Rocco Oliveto und Andrea De Lucia: „Automatic Test Case Generation: What if Test Code Quality Matters?“ In: *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. Saarbrücken: ACM, 2016, S. 130–141.

Panichella und Molina, 2017

Panichella, Annibale und Urko Rueda Molina: „Java Unit Testing Tool Competition - Fifth Round“. In: *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*. 2017, S. 32–38.

Parker, 2010

Parker, Wendy S.: „Predicting weather and climate: Uncertainty, ensembles and probability“. In: *Studies in History and Philosophy of Science Part B: Studies in History and Philosophy of Modern Physics*, Nr. 41.3 (2010). Special Issue: Modelling and Simulation in the Atmospheric and Climate Sciences, S. 263–272.

Parnas, 1972

Parnas, D. L.: „On the Criteria to Be Used in Decomposing Systems into Modules“. In: *Communications of the ACM*, Nr. 15.12 (1972), S. 1053–1058.

pasc-ch.org, ENIAC

Lohmann, Ulrike u. a.: *ENIAC: Enabling the ICON model on heterogeneous architectures*. Platform for Advanced Scientific Computing, Centro Svizzero di Calculo Scientifico (CSCS), 2018. URL: https://www.pasc-ch.org/projects/2017-2020/eniac-enabling-the-icon-model-on-heterogeneous-architectures (besucht am 03. 12. 2019).

Perry, 2006

Perry, William: *Effective Methods for Software Testing, Third Edition*. Indianapolis, USA: Wiley, 2006.

Peterson, 2009

Peterson, Pearu: „F2PY: a tool for connecting Fortran and Python programs“. In: *International Journal of Computational Science and Engineering*, Nr. 4.4 (2009).

Phillips, 1956

Phillips, Norman A.: „The general circulation of the atmosphere: A numerical experiment“. In: *Quarterly Journal of the Royal Meteorological Society*, Nr. 82.352 (1956), S. 123–164.

Pincus u. a., 2008

Pincus, Robert; Crispian P. Batstone; Robert J. Patrick Hofmann; Karl E. Taylor und Peter J. Glecker: „Evaluating the present-day simulation of clouds, precipitation, and radiation in climate models“. In: *Journal of Geophysical Research: Atmospheres*, Nr. 113.D14 (2008).

Pipitone und Easterbrook, 2012

Pipitone, Jon und Steve Easterbrook: „Assessing climate model software quality:

a defect density analysis of three models“. In: *Geoscientific Model Development*, Nr. 5.4 (2012), S. 1009–1022.

Pope und Davies, 2002

Pope, Vicky und Terry Davies: „Testing and evaluating atmospheric climate models“. In: *Computing in Science & Engineering*, Nr. 4.5 (2002), S. 64–69.

PRISM, 2015

PRISM – Programme for Integrated Earth System Modelling. European Network for Earth System Modelling (enes), 2015. URL: https://portal.enes.org/community/projects-and-initiatives/projects/prism (besucht am 03. 12. 2019).

PromO, 2012

Promotionsordnung der Fakultät für Mathematik, Informatik und Naturwissenschaftender Universität Hamburg – Vom Fakultätsrat am 01. Dezember 2010 beschlossen – mit Änderungen der Promotionsordnung der Fakultät für Mathematik, Informatik und Naturwissenschaften der Universität Hamburg vom 2. Mai 2012 und vom 10. Oktober 2012. Hamburg: Universität Hamburg, 2012.

Puri, Redler und Budich, 2013

Puri, Kamal; René Redler und Reinhard Budich: *Earth System Modelling - Volume 1. Recent Developments and Projects*. Springer Briefs in Earth System Sciences. Berlin/Heidelberg: Springer-Verlag, 2013.

pySCM, 2014

pySCM - Simple Climate Model. Bodeker Scientific, 2014. URL: https://rdc.bodekerscientific.com/docs/pySCM/index.html (besucht am 03. 12. 2019).

R-CCS, SCALE

SCALE – Scalable Computing for Advanced Library and Environment. RIKEN Center for Computational Science (R-CCS). URL: http://r-ccs-climate.riken.jp/scale (besucht am 03. 12. 2019).

redmine.org

Lang, Jean-Philippe: *Redmine*. URL: https://www.redmine.org (besucht am 03. 12. 2019).

Reichler und Kim, 2008

Reichler, Thomas und Junsu Kim: „How Well Do Coupled Models Simulate Today's Climate?“ In: *Bulletin of the American Meteorological Society*, Nr. 89.3 (2008), S. 303–311.

Richardson, 1922

Richardson, Lewis Fry: *Weather prediction by numerical process*. London, UK: Cambridge University Press, 1922.

Rilee und Clune, 2014

Rilee, Michael und Thomas L. Clune: „Towards Test Driven Development for Computational Science with pFUnit“. In: *Proceedings of the 2nd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE '14)*. New Orleans, LA, USA: IEEE Press, 2014, S. 20–27.

Ripodas u. a., 2009
Ripodas, P.; A. Gassmann; J. Förstner; D. Majewski; M. Giorgetta; P. Korn; L. Kornblueh; H. Wan; G. Zängl; L. Bonaventura und T. Heinze: „Icosahedral Shallow Water Model (ICOSWM): results of shallow water test cases and sensitivity to model parameters“. In: *Geoscientific Model Development*, Nr. 2.2 (2009), S. 231–251.

Rojas, Fraser und Arcuri, 2015
Rojas, José Miguel; Gordon Fraser und Andrea Arcuri: „Automated Unit Test Generation During Software Development: A Controlled Experiment and Think-aloud Observations“. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. Baltimore, MD, USA: ACM, 2015, S. 338–349.

Rushby, 2008
Rushby, John: „Automated Test Generation and Verified Software“. In: *Verified Software: Theories, Tools, Experiments*. Hrsg. von Bertrand Meyer und Jim Woodcock. Berlin/Heidelberg: Springer-Verlag, 2008, S. 161–172.

Saff u. a., 2005
Saff, David; Shay Artzi; Jeff H. Perkins und Michael D. Ernst: „Automatic Test Factoring for Java“. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ASE '05. Long Beach, CA, USA: ACM, 2005, S. 114–123.

Sato u. a., 2015
Sato, Yousuke; Seiya Nishizawa; Hisashi Yashiro; Yoshiaki Miyamoto; Yoshiyuki Kajikawa und Hirofumi Tomita: „Impacts of cloud microphysics on trade wind cumulus: which cloud microphysics processes contribute to the diversity in a large eddy simulation?“ In: *Progress in Earth and Planetary Science*, Nr. 2.1 (2015), S. 23.

Satoh, Matsuno u. a., 2008
Satoh, M.; T. Matsuno; H. Tomita; H. Miura; T. Nasuno und S. Iga: „Nonhydrostatic icosahedral atmospheric model (NICAM) for global cloud resolving simulations“. In: *Journal of Computational Physics*, Nr. 227.7 (2008), S. 3486–3514.

Satoh, Stevens u. a., 2019
Satoh, Masaki; Bjorn Stevens; Falko Judt; Marat Khairoutdinov; Shian-Jiann Lin; William M. Putman und Peter Düben: „Global Cloud-Resolving Models“. In: *Current Climate Change Reports* (2019).

Satoh, Tomita u. a., 2014
Satoh, Masaki; Hirofumi Tomita; Hisashi Yashiro; Hiroaki Miura; Chihiro Kodama; Tatsuya Seiki; Akira T. Noda; Yohei Yamada; Daisuke Goto; Masahiro Sawada; Takemasa Miyoshi; Yosuke Niwa; Masayuki Hara; Tomoki Ohno; Shin-ichi Iga; Takashi Arakawa; Takahiro Inoue und Hiroyasu Kubokawa: „The Non-hydrostatic Icosahedral Atmospheric Model: description and development“. In: *Progress in Earth and Planetary Science*, Nr. 1.1 (2014), S. 18.

Schmidt u. a., 2006
Schmidt, Gavin A.; Reto Ruedy; James E. Hansen; Igor Aleinov; Nadine Bell; Mike Bauer; Susanne Bauer; Brian Cairns; Vittorio Canuto; Ye Cheng; Anthony Del Genio; Greg Faluvegi; Andrew D. Friend; Tim M. Hall; Yongyun Hu; Max Kelley; Nancy Y. Kiang; Dorothy Koch; Andy A. Lacis; Jean Lerner; Ken K. Lo; Ron L. Miller; Larissa Nazarenko; Valdar Oinas; Jan Perlwitz; Judith Perlwitz; David Rind; Anastasia Romanou; Gary L. Russell; Makiko Sato; Drew T. Shindell; Peter H. Stone; Shan Sun; Nick Tausnev; Duane Thresher und Mao-Sung Yao: „Present-Day Atmospheric Simulations Using GISS ModelE: Comparison to In Situ, Satellite, and Reanalysis Data“. In: *Journal of Climate*, Nr. 19.2 (2006), S. 153–192.

Schwartz und Hetzel, 2016
Schwartz, Amanda und Michael Hetzel: „The Impact of Fault Type on the Relationship between Code Coverage and Fault Detection“. In: *2016 IEEE/ACM 11th International Workshop in Automation of Software Test (AST)*. 2016, S. 29–35.

Segal, 2009
Segal, Judith: „Some challenges facing software engineers developing software for scientists“. In: *Software Engineering for Computational Science and Engineering, 2009. SECSE '09. ICSE Workshop on*. 2009, S. 9–14.

Seung-Hee Han, 2008
Seung-Hee Han, Yong-Rae Kwon: „An Empirical Evaluation of Test Data Generation Techniques“. In: *Journal of Computing Science and Engineering*, Nr. 2.3 (2008), S. 274–300.

Shahamiri, Kadir und Mohd-Hashim, 2009
Shahamiri, Seyed Reza; Wan Mohd Nasir Wan Kadir und Siti Zaiton Mohd-Hashim: „A Comparative Study on Automated Software Test Oracle Methods“. In: *2009 Fourth International Conference on Software Engineering Advances (ICSEA '09)*. 2009, S. 140–145.

Shamshiri u. a., 2015
Shamshiri, Sina; René Just; José Miguel Rojas; Gordon Fraser; Phil McMinn und Andrea Arcuri: „Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T)“. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015, S. 201–211.

Shayma Mustafa Mohi-Aldeen, 2014
Shayma Mustafa Mohi-Aldeen Safaai Deris, Radziah Mohamad: „Systematic Mapping Study in Automatic Test Case Generation“. In: *New Trends in Software Methodologies, Tools and Techniques*. Hrsg. von Hamido Fujita; Ali Selamat und Habibollah Haron. Bd. 265. Frontiers in Artificial Intelligence and Applications. IOS Press, 2014, S. 703–720.

Shirole und Kumar, 2013
Shirole, Mahesh und Rajeev Kumar: „UML Behavioral Model Based Test Case

Generation: A Survey“. In: *SIGSOFT Software Engineering Notes*, Nr. 38.4 (2013), S. 1–13.

Shrestha und Rutherford, 2011
Shrestha, Kavir und Matthew J. Rutherford: „An Empirical Evaluation of Assertions as Oracles“. In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. 2011, S. 110–119.

Silverstein, 2003
Silverstein, Michael: „Logical Capture/Replay“. In: *STQE*, Nr. 2003.6 (2003), S. 36–42.

Sjösten-Andersson und Pareto, 2006
Sjösten-Andersson, Erik und Lars Pareto: „Costs and Benefits of Structure-aware Capture/Replay tools“. In: *Proceedings of the Sixth Conference on Software Engineering Research and Practice in Sweden (SERPS'06)*. Umeå, Schweden, 2006, S. 3–11.

Slawig, 2017
Slawig, Thomas: *Fortran im Wandel der Zeit*. heise Developer, 2017. URL: https://www.heise.de/developer/artikel/Fortran-im-Wandel-der-Zeit-3677272.html (besucht am 03. 12. 2019).

Smith u. a., 2018
Smith, Christopher J.; Piers M. Forster; Myles Allen; Nicholas Leach; Richard J. Millar; Giovanni A. Passerello und Leighton A. Regayre: „FAIR v1.3: a simple emissions-based impulse response and carbon cycle model“. In: *Geoscientific Model Development*, Nr. 11.6 (2018), S. 2273–2297.

Spillner und Linz, 2012
Spillner, Andreas und Tilo Linz: *Basiswissen Softwaretest : Aus- und Weiterbildung zum Certified Tester*. 5., überarb. und aktualisierte Aufl. Heidelberg: dpunkt-Verl., 2012.

Staats, Whalen und Heimdahl, 2011
Staats, Matt; Michael W. Whalen und Mats P.E. Heimdahl: „Programs, tests, and oracles: the foundations of testing revisited“. In: *2011 33rd International Conference on Software Engineering (ICSE '11)*. 2011, S. 391–400.

Staniforth und Thuburn, 2012
Staniforth, Andrew und John Thuburn: „Horizontal grids for global weather and climate prediction models: a review“. In: *Quarterly Journal of the Royal Meteorological Society*, Nr. 138.662 (2012), S. 1–26.

subversion.apache.org
Apache Subversion. Apache Software Foundation (ASF). URL: https://subversion.apache.org (besucht am 03. 12. 2019).

Tatebe u. a., 2019
Tatebe, Hiroaki; Tomoo Ogura; Tomoko Nitta; Yoshiki Komuro; Koji Ogochi; Toshihiko Takemura; Kengo Sudo; Miho Sekiguchi; Manabu Abe; Fuyuki Saito; Minoru Chikira; Shingo Watanabe; Masato Mori; Nagio Hirota; Yoshio Kawatani;

Takashi Mochizuki; Kei Yoshimura; Kumiko Takata; Ryouta O'ishi; Dai Yamazaki; Tatsuo Suzuki; Masao Kurogi; Takahito Kataoka; Masahiro Watanabe und Masahide Kimoto: „Description and basic evaluation of simulated mean state, internal variability, and climate sensitivity in MIROC6". In: *Geoscientific Model Development*, Nr. 12.7 (2019), S. 2727–2765.

Tavis Rudd und Bicking, 2002

Tavis Rudd, Mike Orr und Ian Bicking: „Cheetah: The Python-Powered Template Engine". In: *10th International Python Conference*. Alexandria, VA, USA, 2002.

Tebaldi und Knutti, 2007

Tebaldi, Claudia und Reto Knutti: „The use of the multi-model ensemble in probabilistic climate projections". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, Nr. 365.1857 (2007), S. 2053–2075.

Tomita, Goto und Satoh, 2008

Tomita, Hirofumi; Koji Goto und Masaki Satoh: „A New Approach to Atmospheric General Circulation Model: Global Cloud Resolving Model NICAM and its Computational Performance". In: *SIAM Journal on Scientific Computing*, Nr. 30.6 (2008), S. 2755–2776.

Tomita, Tsugawa u. a., 2001

Tomita, Hirofumi; Motohiko Tsugawa; Masaki Satoh und Koji Goto: „Shallow Water Model on a Modified Icosahedral Geodesic Grid by Using Spring Dynamics". In: *Journal of Computational Physics*, Nr. 174.2 (2001), S. 579–613.

TravisCI, MOM6

Travis-CI: MOM6. NOAA-GFDL. URL: https://travis-ci.com/NOAA-GFDL/MOM6 (besucht am 03. 12. 2019).

Tsai u. a., 2001

Tsai, W. T.; Xiaoying Bai; Ray Paul; Weiguang Shao und Vishal Agarwal: „End-to-end integration testing design". In: *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*. 2001, S. 166–171.

Unidata, NetCDF

Network Common Data Form (NetCDF). Unidata. URL: https://www.unidata.ucar.edu/software/netcdf (besucht am 03. 12. 2019).

Uppenbrink, 1996

Uppenbrink, Julia: „Arrhenius and Global Warming". In: *Science*, Nr. 272.5265 (1996), S. 1122–1122.

Utting, Pretschner und Legeard, 2012

Utting, Mark; Alexander Pretschner und Bruno Legeard: „A taxonomy of model-based testing approaches". In: *Software Testing, Verification and Reliability*, Nr. 22.5 (2012), S. 297–312.

Valcke, Redler und Budich, 2012

Valcke, Sophie; René Redler und Reinhard Budich: *Earth System Modelling - Volu-*

me 3. Coupling Software and Strategies. Springer Briefs in Earth System Sciences. Berlin/Heidelberg: Springer-Verlag, 2012.

valgrind.org

Valgrind. URL: http://www.valgrind.org (besucht am 03. 12. 2019).

Wang, Xu u. a., 2014

Wang, Dali; Yang Xu; Peter Thornton; Anthony King; Chad Steed; Lianhong Gu und Joseph Schuchart: „A functional test platform for the Community Land Model". In: *Environmental Modelling & Software*, Nr. 55 (2014), S. 25–31.

Wang, Yao und Winkler, 2016

Wang, Dali; Zhuo Yao und Frank Winkler: „Building a Function Testing Platform for Complex Scientific Code". In: Carver, Jeffrey C.; Neil P. Chue Hong und George K. Thiruvathukal: *Software Engineering for Science*. Boca Raton, USA: CRC Press, 2016, S. 135–148.

Wang und Offutt, 2009

Wang, Shuang und Jeff Offutt: „Comparison of Unit-Level Automated Test Generation Tools". In: *2009 International Conference on Software Testing, Verification, and Validation Workshops*. 2009, S. 210–219.

Watanabe u. a., 2011

Watanabe, S.; T. Hajima; K. Sudo; T. Nagashima; T. Takemura; H. Okajima; T. Nozawa; H. Kawase; M. Abe; T. Yokohata; T. Ise; H. Sato; E. Kato; K. Takata; S. Emori und M. Kawamiya: „MIROC-ESM 2010: model description and basic results of CMIP5-20c3m experiments". In: *Geoscientific Model Development*, Nr. 4.4 (2011), S. 845–872.

Weide, 2001

Weide, Bruce W.: „Modular regression testing: Connections to component-based software". In: *Proceedings Fourth ICSE Workshop on Component-Based Software Engineering*. 2001, S. 47–51.

Wieringa, 2014

Wieringa, Roel J.: *Design Science Methodology for Information Systems and Software Engineering*. Berlin/Heidelberg: Springer-Verlag, 2014.

Williamson, 2007

Williamson, David L.: „The Evolution of Dynamical Cores for Global Atmospheric Models". In: *Journal of the Meteorological Society of Japan. Ser. II*, Nr. 85B (2007), S. 241–269.

Williamson u. a., 1992

Williamson, David L.; John B. Drake; James J. Hack; Rüdiger Jakob und Paul N. Swarztrauber: „A Standard Test Set for Numerical Approximations to the Shallow Water Equations in Spherical Geometry". In: *Journal of Computational Physics*, Nr. 102.1 (1992), S. 211–224.

Wilson u. a., 1994

Wilson, Robert P.; Robert S. French; Christopher S. Wilson; Saman P. Amarasinghe; Jennifer M. Anderson; Steve W. K. Tjiang; Shih-Wei Liao; Chau-Wen Tseng;

Mary W. Hall; Monica S. Lam und John L. Hennessy: „SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers“. In: *ACM SIGPLAN Notices*, Nr. 29.12 (1994), S. 31–37.

WMO, 1992

International Meteorological Vocabulary. Genf, Schweiz: World Meteorological Organization (WMO), 1992.

WMO, 2011

Guide to Climatological Practices. Genf, Schweiz: World Meteorological Organization (WMO), 2011.

WMO, 2015

Manual on Codes - International Codes, Volume I.2, Annex II to the WMO Technical Regulations: Part B and Part C. Genf, Schweiz: World Meteorological Organization (WMO), 2015.

Xie, 2012

Xie, Tao: „Cooperative Testing and Analysis: Human-Tool, Tool-Tool and Human-Human Cooperations to Get Work Done“. In: *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Trento, Italien, 2012, S. 1–3.

Yao, Jia u. a., 2016

Yao, Zhuo; Yulu Jia; Dali Wang; Chad Steed und Scott Atchley: „In Situ Data Infrastructure for Scientific Unit Testing Platform“. In: *ICCS 2016. The International Conference on Computational Science*. Bd. 80. Procedia Computer Science. San Diego, CA, USA, 2016, S. 587–598.

Yao, Wang u. a., 2017

Yao, Zhuo; Dali Wang; Jinyuan Sun und Dong Zhong: „A Unit Testing Framework for Scientific Legacy Code“. In: *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*. 2017, S. 940–944.

Zängl u. a., 2015

Zängl, Günther; Daniel Reinert; Pilar Rípodas und Michael Baldauf: „The ICON (ICOsahedral Non-hydrostatic) modelling framework of DWD and MPI-M: Description of the non-hydrostatic dynamical core“. In: *Quarterly Journal of the Royal Meteorological Society*, Nr. 141.687 (2015), S. 563–579.

Zeller, 2005

Zeller, Andreas: *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

Zhao u. a., 2018a

Zhao, M.; J.-C. Golaz; I. M. Held; H. Guo; V. Balaji; R. Benson; J.-H. Chen; X. Chen; L. J. Donner; J. P. Dunne; K. Dunne; J. Durachta; S.-M. Fan; S. M. Freidenreich; S. T. Garner; P. Ginoux; L. M. Harris; L. W. Horowitz; J. P. Krasting; A. R. Langenhorst; Z. Liang; P. Lin; S.-J. Lin; S. L. Malyshev; E. Mason; P. C. D. Milly; Y. Ming; V. Naik; F. Paulot; D. Paynter; P. Phillipps; A. Radhakrishnan; V. Ramaswamy; T. Robinson; D. Schwarzkopf; C. J. Seman; E. Shevliakova; Z. Shen;

H. Shin; L. G. Silvers; J. R. Wilson; M. Winton; A. T. Wittenberg; B. Wyman und B. Xiang: „The GFDL Global Atmosphere and Land Model AM4.0/LM4.0: 1. Simulation Characteristics With Prescribed SSTs“. In: *Journal of Advances in Modeling Earth Systems*, Nr. 10.3 (2018), S. 691–734.

Zhao u. a., 2018b

Zhao, M.; J.-C. Golaz; I. M. Held; H. Guo; V. Balaji; R. Benson; J.-H. Chen; X. Chen; L. J. Donner; J. P. Dunne; K. Dunne; J. Durachta; S.-M. Fan; S. M. Freidenreich; S. T. Garner; P. Ginoux; L. M. Harris; L. W. Horowitz; J. P. Krasting; A. R. Langenhorst; Z. Liang; P. Lin; S.-J. Lin; S. L. Malyshev; E. Mason; P. C. D. Milly; Y. Ming; V. Naik; F. Paulot; D. Paynter; P. Phillipps; A. Radhakrishnan; V. Ramaswamy; T. Robinson; D. Schwarzkopf; C. J. Seman; E. Shevliakova; Z. Shen; H. Shin; L. G. Silvers; J. R. Wilson; M. Winton; A. T. Wittenberg; B. Wyman und B. Xiang: „The GFDL Global Atmosphere and Land Model AM4.0/LM4.0: 2. Model Description, Sensitivity Studies, and Tuning Strategies“. In: *Journal of Advances in Modeling Earth Systems*, Nr. 10.3 (2018), S. 735–769.

Zoller und Schmolitzky, 2012

Zoller, Christian und Axel Schmolitzky: „AccessAnalysis: A Tool for Measuring the Appropriateness of Access Modifiers in Java Systems“. In: *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 2012, S. 120–125.

Züllighoven, 1998

Züllighoven, Heinz: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- & Material-Ansatz*. Heidelberg: dpunkt.verlag, 1998.

Züllighoven, Bäumer u. a., 1998

Züllighoven, Heinz; Dirk Bäumer; Guido Gryczan und Thomas Slotos: „Anwendungsorientierte Softwareentwicklung“. In: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- & Material-Ansatz*. Heidelberg: dpunkt.verlag, 1998, S. 258–296.

Züllighoven und Gryczan, 1998

Züllighoven, Heinz und Guido Gryczan: „Leitbilder und Entwurfsmetaphern“. In: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- & Material-Ansatz*. Heidelberg: dpunkt.verlag, 1998, S. 147–207.

Züllighoven, Gryczan u. a., 1999

Züllighoven, Heinz; Guido Gryczan; Anita Krabbel und Ingrid Wetzel: „Application-Oriented Software Development for Supporting Cooperative Work. Human Factors and Ergonomics“. In: *Human-Computer Interaction: Ergonomics and User Interfaces*. Hrsg. von Hans-Jörg Bullinger und Jürgen Ziegler. Bd. 1. 1999, S. 1213–1217.

Eigene Veröffentlichungen

Wissenschaftliche Publikationen

Folgende Vorveröffentlichungen sind aus dieser Dissertation hervorgegangen:

Hovy und Kunkel, 2016
: Hovy, Christian und Julian Kunkel: „Towards Automatic and Flexible Unit Test Generation for Legacy HPC Code“. In: *2016 Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE)*. 2016, S. 1–8.

Hovy und Kunkel, 2017
: Hovy, Christian und Julian Kunkel: „FortranTestGenerator: Automatic and Flexible Unit Test Generation for Legacy HPC Code“. In: *ISC High Performance 2017*. Poster. Frankfurt am Main, 2017.

Folgender Artikel wurde zur Veröffentlichung eingereicht:

Hovy u. a., einger. 2019
: Hovy, Christian; Tomohiro Hajima; Luis Kornblueh; Seth Underwood und Hisashi Yashiro: „A Qualitative Study on Software Testing Practices in Climate Model Development“. 2019 eingereicht; nicht angenommen. Aktuell unveröffentlicht.

Software

Folgende Programme wurden im Rahmen dieser Dissertation entwickelt und veröffentlicht:

FortranTestGenerator	https://github.com/fortesg/fortrantestgenerator
FortranCallGraph	https://github.com/fortesg/fortrancallgraph

Zudem wurden Beiträge zu folgender Bibliothek geleistet:

Serialbox2	https://github.com/GridTools/serialbox

Eidesstattliche Versicherung

Hiermit erkläre ich gemäß §7 Abs. 4 Promotionsordnung MIN-Fakultät (2010) an Eides statt, dass ich die vorliegende Dissertation selbst verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Hamburg, den 23. Januar 2020

Christian Hovy