Functional and Structural Extensions of the AC^2 Numerics

GRS - 758

Gesellschaft für Anlagen-
und Reaktorsicherheit
(GRS) gGmbH

Functional and Structural Extensions of the AC^2 Numerics

Funktions- und Strukturausbau der AC^2-Numerik

Final Report

Tim Steinhoff
Volker Jacht
Daniel von der Cron
Dandy Eschricht
Markus Junk
Jonas Wack

September 2023

Remark

This report refers to the research project RS1593 which has been funded by the German Federal Ministry for the Environment, Nature Conversation, Nuclear Safety and Consumer Protection (BMUV).

The work was conducted by GRS.

The authors are responsible for the content of the report.

GRS - 758
ISBN 978-3-910548-49-7

Keywords

AC^2, CI/CD, coupled simulations, differential equations, GitLab, MPI, Numerical Toolkit, NuT, ODEs, PETSc, software development

Abstract

GRS develops the program package AC^2 which is employed for the safety analysis of nuclear reactors, research reactors and other nuclear facilities. Its field of application covers normal operation, transients, and accidents up to severe accidents with radioactive release at the site boundary. The main components are ATHLET, ATHLET-CD, and COCOSYS. AC^2 also includes the Numerical Toolkit (NuT), which serves as AC^2's sustainable software component to provide easy access to dedicated numerical algorithms and data structures.

The focus of this project lay on extending the AC^2 architecture to allow for an alternative and more flexible approach to coupled computations by means of introducing ODE features to NuT. This was accompanied by work on software-related aspects focusing on NuT and the development cycle of AC^2. Furthermore, ATHLET's steady state calculation has been investigated in some detail.

Within the project, the functionality of NuT was extended in order to execute ODE methods. This was implemented in terms of a per-step logic where a method performs one time integration step per given step size using auxiliary PETSc procedures. Several ODE methods are now available to NuT. The interfaces for the AC^2 codes ATHLET and COCOSYS (module THY) were implemented and tested successfully. In verification calculations the correct implementation of these ODE methods for stand-alone and coupled calculations was demonstrated. In addition, a framework for thermo-hydraulically coupled calculations between ATHLET and COCOSYS was facilitated by advanced coupling numerics provided via NuT. Different coupling approaches were discussed theoretically, and boundary conditions and constraints from ATHLET and COCOSYS-THY were analyzed. The monolithic coupling approach was chosen, where the ODE equations from each code are treated within one unified overall ODE system. NuT was extended to build up the overall Jacobian matrix from the Jacobians of ATHLET and COCOSYS and from the derivatives that represent the mutual influences to enable this approach. The consistent treatment of discontinuities and time step reductions due to code model requirements for the coupled code system was ensured. Moreover, the AC^2 communication architecture was optimized for this approach. A first functional implementation was achieved, but further work will be needed to fully establish this method within AC^2.

In parallel, the implementation of NuT was checked within the EU-funded project POP. This showed that NuT's implementation of its communication structure via MPI

is efficient. Additionally, it was found that the instruction scalability of NuT appears to be weak. This is due to long waiting times for the ATHLET process to send data to NuT. Consequently, future efforts should be focused on enhancing the speed of the system code calculations, e. g., by parallel processing via OpenMP.

The second large topic was improving the AC^2 development cycle. Importantly, substantial automation of the AC^2 build process within the GRS GitLab infrastructure was achieved enabling continuous deployment of GRS-internal code versions. In addition, a structured approach to multi-project developments within the AC^2 program landscape was established, and tools and methods for this approach were provided.

Finally, the steady state calculation of ATHLET was analyzed in depth and documented accordingly. Based on this work, some improvements could be implemented, e. g., the initialization of ATHLET zones with multiple non-condensable gases.

Overall, the main objectives of the project have been achieved. Follow-up work on improving NuT and the thermal-hydraulic coupling of ATHLET and COCOSYS within AC^2 will be necessary in future project.

Kurzfassung

Die GRS entwickelt das Programmpaket AC^2 für die Sicherheitsanalyse von Kernkraftwerken, Forschungsreaktoren und anderen kerntechnischen Anlagen /WEY 23/. Mit AC^2 können der Normalbetrieb, Transienten und Störfälle bis hin zu Unfällen mit Freisetzung von Spaltprodukten in die Umgebung mit realistischen Modellen simuliert werden. Zentrale Programme innerhalb von AC^2 sind ATHLET (Analysis of THermal-hydraulics of LEaks and Transients), ATHLET-CD (Core Degradation) und COCOSYS (COntainment COde SYStem). ATHLET dient der Simulation von Phänomenen im Primärkreislauf vor Beginn der Kernzerstörung. Mit ATHLET-CD wird dies für die Phänomene und Prozesse bei Kernzerstörung, Schmelzeverlagerung ins untere Plenum und Freisetzung von Spaltprodukten aus dem Kerninventar erweitert. COCOSYS ermöglicht die Simulation der Phänomene im Sicherheitsbehälter von Betrieb bis Unfallszenarien und erlaubt die Analyse des Containmentverhaltens unter Unfallbedingungen bis hin zur Berechnung des freigesetzten Quellterms. Für multi-physikalische Rechnungen des integralen Anlagenverhaltens bei Stör- und Unfällen und zur realistischen Berücksichtigung der Interaktion der Vorgänge im Kühlkreislauf und Reaktor können die Programme ATHLET und ATHLET-CD (ATHLET/CD) mit COCOSYS innerhalb von AC^2 gekoppelt werden.

Das Programmpaket wird durch weitere Simulationscodes und Werkzeuge ergänzt. Eine zentrale Komponente ist hierbei das Numerical Toolkit (NuT), welches für die AC^2-Programme zentral einen einfachen Zugang zu dedizierten numerischen Algorithmen und Datenstrukturen bereit stellt. Intern greift NuT auf Funktionalitäten der PETSc-Bibliothek zu. PETSc ist eine Open-Source-Bibliothek fürs effiziente Lösen numerischer Aufgaben, die von einer internationalen Gemeinschaft von Entwicklern seit vielen Jahren kontinuierlich verbessert wird /BAL 23/.

Die wesentliche numerische Aufgabe in den AC^2-Programmen ist die Zeitintegration für die thermohydraulischen Prozesse, sowohl für die Einzelprogramme als auch im Rahmen gekoppelter AC^2-Simulationen. Für die Lösung der dazugehörigen Differenzialgleichungssysteme (DGL-Systeme) wird die GRS-Numerik-Routine `FEBE` genutzt, die in den 1970er und 1980er Jahren vor dem Hintergrund der damals verfügbaren IT-Infrastruktur und Codeanforderungen entwickelt wurde. Da diese Routine nicht sinnvoll für moderne Aufgaben wie gekoppelte Simulationen erweiterbar ist und aufgrund der besseren Skalierbarkeit der numerischen Methoden in NuT, soll dieses die bestehende Numerik schrittweise ersetzen und neue Funktionalitäten ermöglichen.

Mit dem Release AC^2 2023 kann NuT sowohl im ATHLET/CD-Rechengebiet als auch im COCOSYS-Rechengebiet für die effiziente und skalierbare Bearbeitung der zur Lösung der DGL-Systeme gehörenden linearen Algebra eingesetzt werden. Die Kommunikation zwischen NuT und den AC^2-Programmen ist dabei über den Kommunikationsstandard MPI (Message Passing Interface) realisiert. Die Grundlagen für diese Funktionalitäten wurden in den Vorgängervorhaben RS1530 /STE 17b/ sowie RS1558 /STE 20/ gelegt.

Hauptziel des vorliegenden Vorhabens RS1593 war es, die Entwicklung des Numerical Toolkits fortzuführen, um weitere numerische Funktionen der AC^2-Programme durch das Toolkit verfügbar zu machen sowie die Code-Performance und Wartbarkeit zu verbessern. Dieses Hauptziel wurde in drei Einzelziele aufgegliedert, die in den drei Arbeitspaketen des Vorhabens bearbeitet wurden.

- In Arbeitspaket 1 *Erweiterung der DGL-Numerik in AC^2* sollte eine zentrale DGL-Numerik über NuT zur Verfügung gestellt werden. Insbesondere wurde die thermohydraulische Kopplung zwischen ATHLET/CD und COCOSYS betrachtet, für die eine Zeitintegration als Gesamtsystem ermöglicht werden sollte und für welche die hierfür notwendigen Datenstrukturen und Algorithmen in NuT implementiert werden sollten.
- In Arbeitspaket 2 *Softwaretechnische Verbesserungen im NuT-Kontext* sollte die weitere Codepflege, das Testen sowie die Optimierung des Programmsystems umgesetzt werden.
- In Arbeitspaket 3 *Konzeptarbeiten zur ATHLET-Startrechnung* sollte diese vertieft untersucht und dokumentiert werden. Des Weiteren sollten Verbesserungspotenziale aufgezeigt werden.

Im Folgenden werden die Ergebnisse des Vorhabens zu den einzelnen Arbeitspunkten kurz zusammengefasst.

Erweiterungen zur DGL-Numerik in AC^2

In diesem Arbeitspunkt wurde NuT derart erweitert, dass es DGL-Methoden ausführen kann. Dazu wurde eine Einzelschritt-Logik implementiert, mit der eine ausgewählte Methode einen Zeitschritt für eine gegebene Schrittweite durchführen kann. Die Implementierung in NuT greift hierzu auf Routinen der PETSc-Bibliothek zu. Die DGL-Funktionalität wurde anhand einfacher Testfälle verifiziert. Somit stehen unter NuT nunmehr verschiedene DGL-Lösungsmethoden zur Verfügung. Weiterhin wurde die Schnittstelle zwischen NuT und den AC^2-Programmen ATHLET und COCOSYS (hier

dem Modul THY) so erweitert, dass die DGL-Methoden aus ATHLET bzw. COCOSYS angesprochen werden können. Weiterhin wurde die Kontrolllogik für die numerische Zeitintegration in ATHLET und COCOSYS derart ergänzt, dass die Zeitintegration alternativ mit NuT statt rein mit FEBE durchgeführt werden kann. Damit sind zunächst Simulationen mit den Einzelprogrammen ATHLET/CD und COCOSYS unter Nutzung dieser neuen Funktionalität von NuT möglich. Diese wurde anhand von theoretischen Testfällen verifiziert.

Als nächster Schritt wurde die thermohydraulische Kopplung zwischen ATHLET und COCOSYS angegangen. Hierfür wurden zunächst die Modellterme (Erhaltungsgleichungen und Lösungsvariablen) von ATHLET und COCOSYS und der Einfluss der ATHLET-Parameter auf die COCOSYS-Parameter an der Kopplungsschnittstelle und umgekehrt theoretisch untersucht. Auf dieser Grundlage wurden verschiedene mögliche Kopplungsansätze diskutiert und ihre Eignung für eine Bestimmung numerisch konsistente Informationen über die Ableitungsterme an der Kopplungsschnittstelle bewertet. Dabei stellte sich heraus, dass ein monolithischer Kopplungsansatz für die Zwecke des Vorhabens geeignet ist. Bei diesem Ansatz werden die DGL-Systeme im ATHLET- bzw. COCOSYS-Rechengebiet zu einem übergreifenden Gesamtsystem zusammengefasst. Da in ATHLET bzw. COCOSYS die Ableitungsterme, d. h. die Jacobi-Matrizen, für das jeweilige Rechengebiet bereits ausgerechnet werden, erfordert dieser Ansatz zusätzlich die Bereitstellung der Ableitungsinformationen, welche den gegenseitigen Einfluss der Systeme aufeinander charakterisieren. Der wesentliche Vorteil dieses Ansatzes ist, dass aus numerischer Sicht ein Gesamtsystem wie ein Einzelsystem behandelt werden kann und daher die notwendigen Erweiterungen der Funktionalitäten von NuT überschaubar bleiben. Weiterhin bietet der monolithische Ansatz eine gute Basis für mögliche zukünftige Verbesserungen. Angesichts der softwaretechnischen Komplexität der Realisierung einer Kopplungsnumerik wurde dieser Ansatz für eine Umsetzung im aktuellen Vorhaben ausgewählt.

Entsprechend wurde in NuT die Möglichkeit geschaffen, die Jacobi-Matrix des Gesamtsystems aus den Einzelmatrizen der Rechengebiete und der Ableitungsinformationen an der Schnittstelle aufzubauen, zu speichern und für die Ausführung von DGL-Methoden zu nutzen. Weiterhin wurden die AC^2-Programme ATHLET und COCOSYS so erweitert, dass diese NuT alle Daten für die Jacobi-Matrix des Gesamtsystems zur Verfügung stellen können. Dazu wurden entsprechende Kommunikations- und Datenstrukturen in NuT sowie in den AC^2-Programmen bereitgestellt und verifiziert. Weiterhin wurde die Kontrolllogik für die Zeitintegration in einer gekoppelten AC^2-

Simulation derart erweitert, dass Daten und Zustände zwischen den AC^2-Programmen geeignet synchronisiert werden. Damit ist sichergestellt, dass von NuT genutzte Informationen sowie die AC^2-Kontrolllogik konsistent sind. Ein weiterer wichtiger Schritt war die Berücksichtigung von Unstetigkeiten und Schrittweitenbegrenzungen, die durch Modellanforderungen in den Einzelcodes erforderlich sind. Zusätzlich wurde die Möglichkeit implementiert, einen gekoppelten Zeitschritt mit reduzierter Schrittweite konsistent zu wiederholen. Um das Integrationsverfahren effizient zu gestalten, wurde die Kommunikationsarchitektur innerhalb von AC^2 so optimiert, dass der ATHLET-Prozess und der COCOSYS-THY-Prozess direkt Informationen austauschen können, ohne die allgemeine Prozess-Steuerung in AC^2 nutzen zu müssen.

Dies hat es erlaubt, erste Testrechnungen für einfache gekoppelte Systeme durchzuführen. Bei einem Vergleich der Ergebnisse der neuen Kopplungsmethodik mit denen der vorhandenen Kopplungsmethodik und mit Einzelcode-Ergebnissen zeigte sich, dass der monolithische Kopplungsansatz deutlich besser mit den Ergebnissen des Einzelcodes übereinstimmt und damit eine höhere Vorhersagequalität zeigt. Weiterhin zeigen die Ergebnisse, dass der monolithische Kopplungsansatz bessere numerische Stabilitätseigenschaften aufweist. Gleichzeitig bleiben die Zeitschrittweiten für den monolithischen Ansatz auch nach Ende des transienten Prozesses deutlich unter denen des Einzelcodes und der bisherigen Kopplungsmethodik. Hier besteht Verbesserungspotenzial, insbesondere bei der Zeitschrittweitensteuerung für gekoppelte Simulationen. Des Weiteren wurden im Zuge der Arbeiten einige offene Fragen zur Konsistenz der thermohydraulischen Modellierung zwischen ATHLET und COCOSYS identifiziert, die für eine Verbesserung der Kopplung in AC^2 weiter bearbeitet und gelöst werden sollten.

Softwaretechnische Verbesserungen im NuT-Kontext

In diesem Arbeitspaket wurden eine Reihe von Verbesserungen und Optimierungen an NuT umgesetzt. Um die Nachvollziehbarkeit der von NuT ausgeführten Operationen während einer Simulation zu erhöhen, wurde das sogenannte Logging, also das Aufzeichnen von Statusinformationen zur Laufzeit, weiter verbessert. Dies ermöglicht es Entwicklern und Anwendern, mögliche Probleme bzw. Datensatzfehler während einer Simulation besser zu identifizieren. Hierbei wurde bereits ausgenutzt, dass im Rahmen dieses Arbeitspaketes die Kommunikationsschnittstelle von NuT flexibler gestaltet wurde. Es ist nun möglich, neue Entitäten-Klassen einzuführen, ohne die grundsätzlichen Mechanismen der Kommunikationsschnittstelle anpassen zu müssen. Um die Erzeugung von Jacobi-Matrizen in den AC^2-Codes effizienter

zu gestalten, wurde das sogenannte Seeding weiter optimiert. Dadurch sind insbesondere bei realistischen Anwendungsrechnungen weniger Funktionsauswertungen in den AC^2-Programmen notwendig, was die Rechenläufe beschleunigt. Zusätzlich wurde das Speichermanagement von NuT weiter verbessert, um interne Prozesse in NuT effizienter zu gestalten.

Schließlich hat die GRS hinsichtlich NuT ein Angebot des von der EU geförderten POP-Vorhabens (Performance Optimisation and Productivity) wahrgenommen. Hierbei wurde die Effizienz der Implementierung der MPI-basierten Kommunikations- und Rechenstrukturen in NuT durch die Experten des POP-Vorhabens untersucht und bewertet. In den Metriken des POP-Vorhabens zeigte sich, dass die Kommunikationseffizienz von NuT sehr gut mit der Anzahl der Prozesse skaliert. Dies galt allerdings nicht für die Anzahl der in NuT ausgeführten Anweisungen. Mit steigender Prozessanzahl steigt auch die Anzahl der Anweisungen. Dies liegt daran, dass NuT auf Daten vom Simulationsprogramm – hier ATHLET – warten muss. Dies geschieht über regelmäßiges Abfragen auf dem Kommunikationskanal pro genutzten Prozess. Im Rahmen der POP-Metrik gilt dies als zu zählende Operation. Aus diesen Resultaten lässt sich folgern, dass zur Beschleunigung einer Simulation mit AC^2-Programmen primär die Simulationscodes zusätzliche Rechenressourcen nutzen sollten, und nicht zusätzliche Rechenressourcen für Aufgaben von NuT zur Verfügung gestellt werden sollten. Eine weitere Parallelisierung der AC^2-Simulationscodes, z. B. unter Nutzung von OpenMP, ist daher sinnvoll.

Das zweite wichtige Thema in diesem Arbeitspaket war die weitere Verbesserung des Entwicklungsprozesses für die AC^2-Programme. Hierfür wurden Verfahren und Werkzeuge geschaffen, so dass eine AC^2-Distribution automatisiert unter GitLab generiert, getestet und an die GRS-internen Nutzer verteilt werden kann. Dies erforderte erhebliche Änderungen an einer Vielzahl von Einzelprogrammen zur Vereinheitlichung von softwaretechnischen Abhängigkeiten und Build-Prozessen innerhalb des AC^2-Systems. Dies hat die Voraussetzungen für eine effizientere Nutzung von automatisierten Verfahren zu Regressionstests und Validierungsrechnungen im AC^2-Kontext geschaffen. Dies wird ergänzt durch einen strukturierten Ansatz für die Multi-Projekt-Entwicklungsarbeiten, die innerhalb von AC^2 notwendig sind. Auch hierfür wurden geeignete Werkzeuge zur Verfügung gestellt. Dabei haben sich die Arbeiten zur Automatisierung der Build-Prozesse für die AC^2-Distribution und die verstärkte Nutzung der Continuous-Integration- und Continuous-Deployment-Infrastruktur unter GitLab bereits für den Release von AC^2 2023 bewährt.

Konzeptarbeiten zur ATHLET-Startrechnung

In diesem Arbeitspaket wurde die Startrechnung von ATHLET vertieft analysiert. Die Routinen der Startrechnung und der Datenfluss zwischen ihnen wurden aufgearbeitet und dokumentiert. Ebenso wurden die in der Startrechnung genutzten Modelle und Algorithmen untersucht. In diesem Zuge wurden auch Tests der Funktionalität und Korrektheit der Startrechnung durchgeführt. Dabei konnten einzelne Probleme und Verbesserungspotenziale identifiziert werden. Eine unmittelbar umgesetzte Maßnahme war es, die Möglichkeit zu schaffen, ein Gemisch nicht-kondensierbarer Gase bereits in der Startrechnung zu initialisieren und bei der Bestimmung des Anfangszustands einer Simulation (der steady state calculation) konsistent zu berücksichtigen. Diese Verbesserung ist Bestandteil von AC^2 2023. Zusätzlich wurden eine Reihe von Vorschlägen für eine weitere Verbesserung der Startrechnung für ATHLET abgeleitet, die in zukünftigen Arbeiten aufgegriffen werden können.

Fazit und Ausblick

Insgesamt hat das vorliegende Vorhaben seine Ziele erreicht. In NuT konnten DGL-Methoden erfolgreich implementiert werden. Weiterhin konnte eine erste funktionierende Implementierung einer konsistenten Kopplungsnumerik über einen monolithischen Kopplungsansatz bereitgestellt werden. Aufgrund der Komplexität dieser Aufgabe konnten jedoch nicht alle wünschenswerten und ursprünglich anvisierten Funktionalitäten innerhalb des Vorhabens bereitgestellt werden. Insbesondere fehlen noch die Möglichkeit zu partiellen Updates der Jacobimatrizen im NuT-DGL-Kontext sowie ein konsistenter Restart über entsprechende Ausgabedateien von ATHLET und COCOSYS. Weiterhin sollte auch ein Kontraktivitätstest für den Newton-Prozess innerhalb des Integrationsverfahrens bereitgestellt werden.

Nachfolgende Arbeiten zur Verbesserung von NuT und der thermohydraulischen Kopplung zwischen ATHLET und COCOSYS in AC^2 sollten insbesondere eine Verbesserung der Zeitschrittweitensteuerung anstreben. Dabei sollte vor allem das Problem gelöst werden, dass sich zurzeit die Zeitschrittweiten nicht wie im Einzelcode nach Ende eines transienten Prozesses wieder schneller auf ihren ursprünglichen, größeren Wert erholen. Derzeit sind hier noch die Lösungsvariablen an der Kopplungsschnittstelle limitierend. Weiterhin sollte die Kopplungsschnittstelle ausführlich auf ihre Robustheit und Leistungsfähigkeit getestet werden. Dies sollte insbesondere auch mehrfache Kopplungsschnittstellen umfassen, wie sie in realistischen Datensätzen vorkommen. Zudem sollte die Leistungsfähigkeit und Robustheit für Zwei-Phasen-Strömungen, auch bei Gegenströmung, und auch in parallel gekoppelten Objekten

genauer analysiert werden. Weiterhin erfordert das Gemischspiegelmodell in den Simulationscodes zusätzliche Analysen und Weiterentwicklungen an der Schnittstelle.

Eine weiter anzustrebende Weiterentwicklung ist die Nutzung von sogenannten Multiraten-Methoden in der Kopplung. Da sich die von ATHLET bzw. COCOSYS für das jeweilige Rechengebiet vorgeschlagene Zeitschrittweite zum Teil deutlich unterscheidet und Funktionsauswertungen der Simulationscodes numerisch teuer sind, wäre es hilfreich, jedes Rechengebiet mit einem eigenen Zeitschritt rechnen zu lassen. Dies würde eine deutliche Leistungssteigerung für gekoppelte Simulationen ermöglichen. Die Nutzung von Multiraten-Methoden erfordert sowohl vorherige theoretische Untersuchungen zu den Grundprinzipien und der Stabilität solcher Integrationsverfahren als auch deutliche Erweiterungen in der Funktionalität von NuT und den AC^2-Programmen.

Hinsichtlich des Softwareentwicklungsprozesses für die AC^2-Programme sollten die Vorgehensweisen und Werkzeuge für die Entwicklungs- und Validierungsarbeiten, welche mehrere Projekte gleichzeitig umfassen, unter GitLab weiter verbessert werden. Dabei sollten auch weitere Optimierungen in der Projektstruktur für die AC^2-Programme untersucht werden. Ein generell wichtiger Punkt ist, dass sichergestellt und überprüft werden muss, dass Änderungen an einem AC^2-Code nicht zu Problemen oder zum Verlust der Vorhersagefähigkeit in einem anderen AC^2-Code führen und dass die AC^2-Programme als Gesamtsystem weiterhin ablauffähig bleiben.

Contents

1 Introduction

GRS develops the program package AC^2 which is employed for the safety analysis of nuclear reactors, research reactors and other nuclear facilities. Its field of application covers normal operation, transients, and accidents up to severe accidents with radioactive release at the site boundary. The main components are ATHLET (Analysis of THermal-hydraulics of LEaks and Transients), ATHLET-CD (Core Degradation), and COCOSYS (COntainment COde SYStem) /WEY 23/. These can be used in a coupled way to cover complex simulation scenarios. Also, single code executions are possible to allow for the investigation of specific phenomena and processes.

The corresponding software suite comes with several further tools and programs to complement the main components. This includes the Numerical Toolkit (NuT), which serves as AC^2's sustainable software component to provide easy access to dedicated numerical algorithms and data structures.

A major application of numerical algorithms in AC^2 is the advancement of the time-integration of the thermo-hydraulic processes, both for single code simulations and coupled computations. In AC^2 2023, NuT can be used to support the linear algebra operations of the time-integration in ATHLET and in COCOSYS's thermo-hydraulic module THY. NuT's interface is based on the communication standard MPI (Message Passing Interface). The fundamentals of this approach were established in RS1530 /STE 17b/. AC^2-wide access and further refinements were taken care of in RS1558 /STE 20/.

Thermo-hydraulic computations in AC^2 are based on classical conservation laws for momentum, mass, and energy. Using a finite-volume staggered-grid approach plus further suitable transformations lead to a system of ODEs (Ordinary Differential Equations) in ATHLET and THY, respectively.

The numerical focus of this project lay on extending the AC^2 architecture to allow for an alternative and more flexible approach to coupled computations by means of introducing ODE features to NuT. This covers single code simulations too.

The numerically motivated work was accompanied by work on software-related aspects focusing on NuT and the development cycle of AC^2. With a growing number of components and interactions between these components the software project AC^2 becomes more and more complex. Hence, special emphasis was put on extending automation means to improve the overall AC^2 build process.

Furthermore, this project included some conceptual work as well. The compatibility of thermo-hydraulic modeling within AC^2, and also the details of the workflow of ATHLET's steady state calculations have been investigated.

Accordingly, the work packages of the project and the corresponding chapters of this report are given as follows.

- Ch. 2 – WP1 *Extending AC^2's ODE-Numerics*
- Ch. 3 – WP2 *Improving NuT and AC^2 on the Level of Software Engineering*
- Ch. 4 – WP3 *Reviewing ATHLET's Steady State Calculation on a Conceptual Level*

Finally, in Chapter 5 the results of this project are summarized and a brief outlook on open issues and potential follow-up work is given.

1.1 General remarks

- Not all basic aspects of the discussed numerics or software development processes can be explained in detail within the constraints of this report. The interested reader is advised to consult the following sources:
  - general and ODE-related numerics
    /COR 13/, /HAI 93/, /HAI 96/, /BUT 16/
  - software development in C++ and MPI
    /LIP 12/, /MAR 17/, /MPI 23/
  - software development with GitLab
    /GIT 24a/
- Several thousand lines of code were written for this project. These cannot be reproduced in this report. However, plenty of diagrams and charts are provided to explain the purpose and structure of the aforementioned changes.
- The refactoring of the NuT code as well as the established AC^2 development cycle as discussed in Chapter 3 were taken into account for the release of AC^2 2023. Especially the substantial CI/CD improvements proved very beneficial for its release procedure. The ODE-related features require further development and are scheduled for a future major release of AC^2.

2 WP1 – Extending AC^2's ODE-Numerics

From a numerical point of view, a thermo-hydraulic coupling between ATHLET/CD and COCOSYS results in a combined system of ODEs introducing mutual dependencies, see (2.3) below. In this project, two possible solution approaches were initially considered: On the one hand, the simultaneous solution of a unified large system via some (linearly) implicit method – the so-called *monolithic* approach – and on the other hand the solution of intertwined subsystems by means of multi-rate methods. Though multi-rate methods would be the superior approach with regards to numerical performance, it was decided to implement the monolithic approach – once thermal-hydraulic investigations confirmed that it is a feasible concept in the given context, see Section 2.2.1.

The monolithic approach treats the coupled system as one single system. This makes it numerically easier to handle, but comes with penalties w. r. t. performance (no tailored step sizes for the subsystems; one large system instead of two small ones to work with). However, it is a robust approach due to the implicit coupling that is realized. It is to be expected that this is reflected in the quality of the results as well. Hence, it is well suited to produce reference solutions for test problems future work on the coupling can be compared with for the sake of verification.

In order to realize a monolithic approach, several modifications of the existing codes of ATHLET, COCOSYS, and NuT were required. Details are given in Section 2.3. Running a test problem is included. Additionally, the coupling was investigated on the level of modeling. Results are presented in Section 2.2.

First, however, in Section 2.1 some terms are defined which are used in the subsequent sections.

2.1 Terminology and basic relations

In this section frequently used terms are described and some basic relations between solution variables of AC^2 simulations are explained. The following sections will make use of these terms and relations.

COCOSYS and THY

The AC^2 program COCOSYS (COntainment COde SYStem) is a code for the simulation of all relevant processes and plant states during operation, transients up to

severe accidents in the containments of light water reactors /KLE 23; ARN 23/. The code consists of several modules which cover different physical phenomena (such as fission product behavior or core concrete interaction) and which may run as separate processes on the computer. THY is the thermo-hydraulic module of COCOSYS. This means, for example, that the terms *thermo-hydraulic ATHLET-COCOSYS coupling* and *ATHLET-THY coupling* are used synonymously in the text at hand. Furthermore, a COCOSYS data set contains THY-related input data and the applied thermo-hydraulic model equations belong to the THY module.

CV, zone, and junction

Both ATHLET and THY use a staggered grid for storing their solution variables and solving their balance equations. In ATHLET, the scalar solution variables which relate to mass and energy conservation for the liquid and the vapor phase are stored in so-called control volumes (CVs) whereas the momentum-related variables are stored on junctions, which are offset by – roughly said – half a CV. Basically, the same holds true for THY, just with a few differences: In THY, the CVs are called *zones* and, as opposed to their ATHLET counterparts, these zones can be further subdivided in so-called parts which are either mainly gaseous, liquid or solid. Moreover, while ATHLET accounts for convective momentum transport through its junctions and deploys elaborate models for the dynamic calculation of friction and form losses, the THY junction model is rather simple as it only considers friction loss of laminar pipe flow and, moreover, does not thoroughly ensure momentum conservation.

Function routines, solution variables, and coupling variables

The mathematical problem to be solved during a stand-alone ATHLET run is the solution of an initial value problem (IVP) for a large ODE system:

$$y' = f(t, y), \quad \text{with initial values } y(t_0) = y_0. \tag{2.1}$$

In (2.1), y is the solution vector, its elements are the so-called *solution variables* which define the state of the simulated system at a given time t. The solution variables of ATHLET are:

- In CVs: total pressure in CV, liquid temperature, vapor and gas temperature, vapor mass fraction, ratio of partial pressure of all non-condensable (NC) gases to total pressure in CV, and mass ratio of one single NC gas component to all NC gas components in CV.
- On junctions: total mass flow rate (if the five-equation model is applied), *or* liquid

velocity times cross-sectional area of the junction and vapor velocity times cross-sectional area of the junction (if the six-equation model is applied).

The right-hand-side of the ODEs, i. e. f, is the *function routine* which contains the physical models applied in ATHLET. In the source code, f is named `AFK`.
The mathematical problem for THY is similar. To distinguish it from ATHLET, the THY IVP is written as

$$u' = g(t, u), \quad \text{with initial values } u(t_0) = u_0. \tag{2.2}$$

The routine name of g in the source code is `FKTFE`. The solution variables of THY are:

- In `NONEQUILIB` zones: component masses in the zone parts, temperatures in the zone parts.
- On junctions of type `INST`: total mass flow rate.

In coupled calculations, data – *coupling variables* – have to be exchanged between both codes at the coupling interface in order to capture the influence one system has on the other. Denoting by α the data which are derived from THY solution variables and act as boundary conditions for ATHLET, and – vice versa – denoting by β the data which are derived from ATHLET solution variables and act as boundary conditions for THY, the coupled ODE system to be solved looks as follows:

$$\begin{aligned} y' &= f(t, y, \alpha) \\ u' &= g(t, u, \beta). \end{aligned} \tag{2.3}$$

The vector α comprises per coupling CV[1]

- the total pressure,
- the liquid temperature,
- the vapor and gas temperature,
- the ratio of partial pressure of all NC gases to total pressure,
- the mass ratio of each single NC gas component to all NC gas components.

Since these quantities are derived from the solution variables of THY, it holds that

$$\alpha = \alpha(u). \tag{2.4}$$

On the other hand, the vector β comprises per coupling junction

- the mass flow rate of each component,

[1] See the paragraph after the next one for the terms *coupling CV* and *coupling junction*.

- the energy flow rate of each component.

Since these quantities are derived from the solution variables of ATHLET, this implies

$$\beta = \beta(y) \tag{2.5}$$

Note that this is a simplified description which aims at clarifying the terminology. Actually, $\beta = \beta(y, \alpha)$ holds true. The implications of this fact are described in Section 2.2.1.

Jacobian information and `FTRIX` blocks

The initial value problems (2.1) and (2.2) are considered as stiff. See Section 2.3.1 for more details on that topic. A consequence of stiffness is that Jacobian information $\partial f/\partial y$ or $\partial g/\partial u$, respectively, has to be available. Consequently, this holds for the coupled system (2.3) as well, see (2.6) below.

`FTRIX` is the sparse matrix package used in both ATHLET and THY for handling the Jacobian. The package exploits the fact that the thermo-hydraulic networks (consisting of CVs/zones and junctions), which are built-up in the data sets as well as through the model equations of the codes, usually lead to diagonally dominant Jacobian matrices (pattern-wise) with lots of zero entries on the off-diagonals. Internally, `FTRIX` transforms the structure of the thermo-fluid dynamic network into a so-called block structure. In ATHLET, for every network element like CV or junction, an `FTRIX` block is defined and the solution variables are collected in it. On the contrary, in THY, the equations of a zone together with those of the junction(s) defined as leaving this zone are collected in a common `FTRIX` block. See Fig. 2.1 for a simple explanation of the differences between the `FTRIX` blocks in ATHLET and THY.

If support by NuT is activated for ATHLET or THY the Jacobian is stored and set in NuT. For this, the block structure information is exploited. Setting values is done by a dedicated host-sided routine that substitutes the default.

Coupling interface

The ATHLET/THY coupling interface is depicted in Fig. 2.2. The figure shows four zones/CVs, connected by three junctions. The blue-colored CVs and junctions are dynamically calculated by ATHLET while the green-colored zones and junctions are calculated by THY. One of the zones is marked as the coupling CV/zone. It is dynamically calculated by THY, but exists in the ATHLET network as well and its function is to provide the above mentioned α boundary conditions to ATHLET. The ATHLET junction entering the coupling CV is the coupling junction which provides the β boundary conditions to THY. Hence, domain-overlapping as a coupling approach is utilized within AC^2.

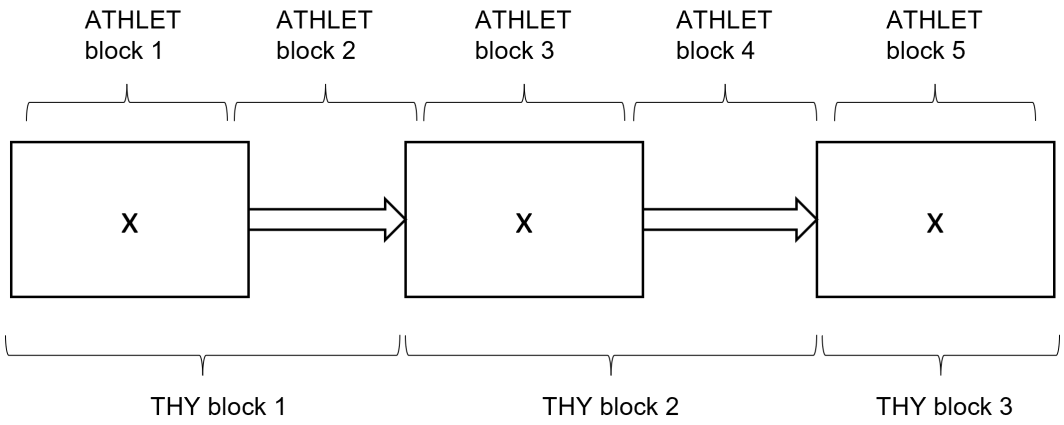

Fig. 2.1 An arrangement of three CVs/zones and two junctions.
In ATHLET, this would result in five `FTRIX` blocks.
In THY, this would result in three `FTRIX` blocks.

Remark 2.1. The definition of coupling interfaces above has already been used in earlier coupling approaches – such as the one pursued in the project RS1535A (EASY) /BUC 18/. The basic concept of the approach hasn't been altered for this project. However, see Section 2.2 for a discussion on what improvements may prove beneficial to be introduced in future work.

Overall Jacobian matrix, matrices LL and UR

Considering a monolithic approach, the complete Jacobian matrix of the coupled ATHLET/THY system must be composed of the "pure" ATHLET and THY Jacobian matrices plus two off-diagonal matrices that reflect the mutual influences. Following the notation from (2.3) it holds that

$$J = \begin{pmatrix} \frac{\partial f}{\partial y} & \frac{\partial f}{\partial u} \\ \frac{\partial g}{\partial y} & \frac{\partial g}{\partial u} \end{pmatrix}. \tag{2.6}$$

A demonstration of the general structure is given in Fig. 2.3. As can be seen in the figure, the matrices of the single systems – as well as the complete Jacobian – are square matrices whereas the off-diagonal matrices LL and UR are rectangular. The dimensions of LL and UR are symmetric and determined by the dimensions of A and T. The UR matrix describes the linear influence of perturbations in the THY solution variables on the ATHLET system (e. g. $\frac{\partial f_9}{\partial u_3}$). Analogously, the UR matrix represents the linear influence of perturbations in the ATHLET solution variables on the THY system (e. g. $\frac{\partial g_4}{\partial y_2}$). While the "pure" ATHLET and THY Jacobian matrices are usually diagonally dominant, this is not the case for the LL and UR matrices.

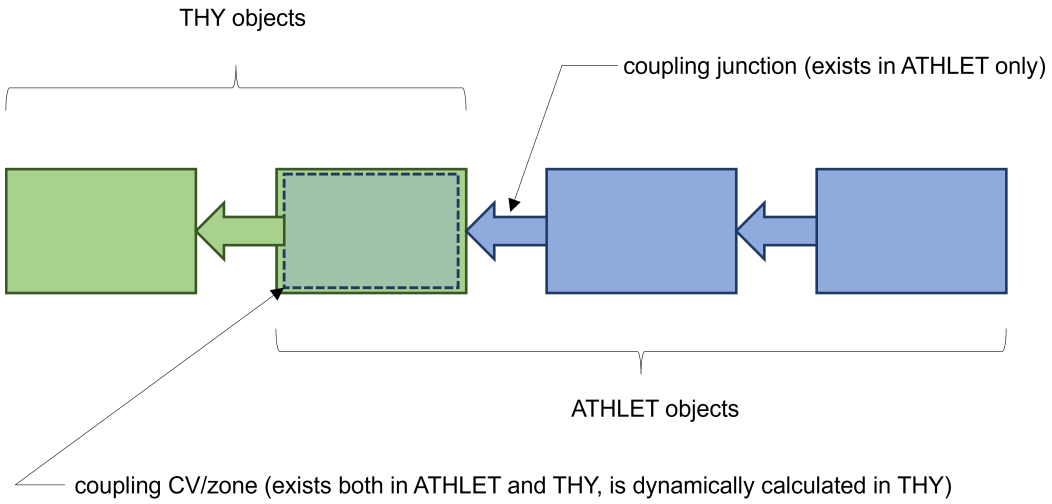

Fig. 2.2 ATHLET/THY coupling interface consisting of a coupling CV (zone), which is modeled both in the ATHLET and COCOSYS input decks, and a coupling junction, which is modeled in the ATHLET input deck only

Nevertheless, since the number of coupling interfaces is usually small compared to the total number of computational nodes in the ATHLET and THY domains, LL and UR are typically sparse. This fact is exploited in the current approach by storing the non-zero elements (structurally speaking) in the so-called compressed sparse column (CSC) and compressed sparse row (CSR) formats.

Remark 2.2. For the sake of demonstration, the sketched matrix in Fig. 2.3 is shown in *element format*, i. e., each small square inside the matrix represents the dependence of a function component on a solution variable. However, when collecting information to build the overall Jacobian matrix as described in the next section, the *block format* for the matrix structure is exploited. A block comprises several elements representing a certain network object within the context of the Jacobian matrix.

Seed matrix

A seed matrix S, also abbreviated as *seed*, contains elements $S_{i,j} \in \{0, 1\}$ such that (formally) multiplied with a Jacobian J, i. e. $J \cdot S$, all non-zero elements of J are contained in the resulting matrix and no linear combinations of multiple non-zero elements occur. The trivial seed is given by the identity matrix. The idea of seeding is to find a matrix S with above properties and with as few numbers of columns as possible. This is motivated by the fact that the number of columns directly relates to the number of function evaluations that are required to determine J by means of finite differences. In order to save computation time fewer evaluations are preferable.

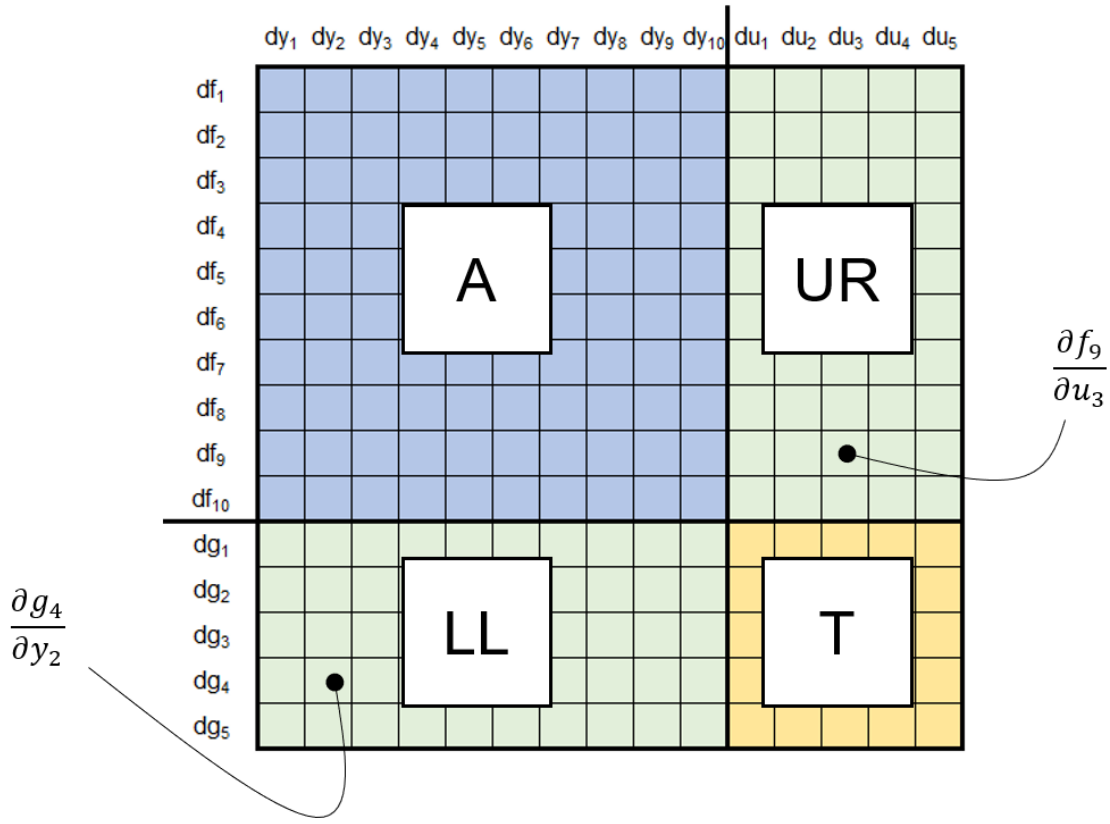

Fig. 2.3 Schematic sketch of the Jacobian matrix of the coupled system. The blue-colored square matrix A belongs to the ATHLET system. The yellow-colored square matrix T belongs to the THY system. The green-colored off-diagonal matrices LL (meaning *lower left*) and UR (meaning *upper right*) represent the mutual influences.
The noted derivatives are just illustrating examples of the matrix elements.

2.2 Preparation of ATHLET and THY to establish a physically consistent coupling within the context of doable numerics

This section comprises discussions of two aspects of the coupling between ATHLET and THY. The first one is related to the feasibility of the monolithic approach in the given context. This directly translates to the question whether a Jacobian for the overall system can be made available. Details are given in Section 2.2.1. The actual implementation is discussed in Section 2.3.2. Furthermore, several samples for testing the ATHLET-COCOSYS coupling have been developed, see Section 2.2.3.

The second aspect considers the coupling between ATHLET and THY on the level of thermo-hydraulic modeling. Both in ATHLET and COCOSYS, various models for CVs/zones and junctions can be applied (dependent on user input). For the coupling approach pursued within RS1593

- the homogeneous CV model in ATHLET,
- the `NONEQUILIB` zone model and `INST` junction model in COCOSYS

were used, because these models are broadly applicable in data sets and seem to be the best compatible ones. Nevertheless, with regard to further extensions of the monolithic approach, the coupling of other zone/CV and junction models is expected to be feasible as well. A discussion on how to improve the AC^2-internal coupling is carried out in Section 2.2.2.

2.2.1 Building a Jacobian matrix for the overall system – feasibility considerations

Before any work on determining an overall Jacobian matrix could be initiated, the general feasibility of this endeavor had to be investigated. Specifically, it was analyzed whether the mutual influence of the ATHLET and THY solution variables is *differentiable*, which is a necessary prerequisite for constructing a Jacobian matrix for the unified system.

As it is not unusual in the context of numerical programming, the use of functions like $\min$, $\max$, $\sqrt{\cdot}$, or $|\cdot|$ as well as the appearance of numerous conditional statements make it difficult to provide meaningful derivative information. These difficulties are present in ATHLET's and THY's codes as well. To overcome these problems the so-called internal numerical differentiation, /BOC 83/, /HAI 93, Sec. II.6/, may be applied to make the desired information available. The basic idea is to follow the same

lines of code for actual evaluations of functions as well as for derivative computations.

An analysis of the thermal-hydraulic models implemented in ATHLET and THY revealed that the influence of the ATHLET solution variables on the THY system – as well as, conversely, the influence of the THY solution variables on the ATHLET system – cannot be reasonably determined analytically. Hence, finite differences were applied to receive derivative information as it is done for each single code already.

The concept of finite differences comes with the amiable property to already follow the lines of code that a usual function evaluation takes, hence, following the idea of internal numerical differentiation. Corner cases may appear due to the required perturbations that are added to the input values of a function evaluation. However, finite differences can be interpreted as approximations to directional derivatives, which are usually available. As a last resort, both function routines, `AFK` and `FKTFE`, can throw an error if an evaluation of their respective argument is not possible. Fall-back strategies kick in, see also Section 2.3.3, to overcome these problems.

In order to make the finite difference approach capable of determining a Jacobian matrix for the unified system, the following tasks had to be taken care of:

- determine suitable perturbation values,
- determine which solution variable has an influence on the other system,
- make perturbation values available to the code that triggers the required function evaluation.

For this project it was decided that the code, which holds a solution variable, applies its usual routine to provide a corresponding perturbation value. Hence, the already existing code could be used. The mutual influence of the systems is directly related to the structure of the matrices LL and UR in Figure 2.3. How to get hold of the matrix structure is discussed in the next paragraph. The last bullet point is part of the discussion given in Section 2.3.2 where the actual computation of the overall Jacobian matrix and corresponding validations are considered.

Remark 2.3. In the course of the project, another approach was discussed to make the information of the Jacobian matrix of the coupled system available. That approach is based on exploiting the chain rule of differentiation to determine the data for UR and LL. The idea is to split the derivatives $\partial f/\partial u$ and $\partial g/\partial y$ up, each into a product of (sub)derivatives where each factor can be calculated locally, either in ATHLET or THY. The ATHLET/CD-Driver would then combine the (sub)derivatives. However, that would have come with a significantly more complex implementation than the

chosen ansatz. Hence, the chain rule approach was discarded eventually. For a more detailed explanation see Appendix A.1.

Furthermore, considering the monolithic approach the corresponding quantities and linear systems to solve scale with the sum of the dimensions of both involved ODE systems. Regarding the linear systems, an alternative approach was discussed to solve only systems of dimensions dictated by either of the ODE systems. Unfortunately, this ansatz becomes quickly inefficient when the number of coupling quantities rises. Hence, it was discarded as well in favor of the obvious choice to simply solve systems of the dimension that reflects the overall system. A brief description of the alternative can be found in Appendix A.2.

Determination of the matrix structure

First, the structure of the complete Jacobian matrix (2.6) was determined in the AC^2-specific block format. As described in Section 2.1, the matrix was decomposed into four submatrices (A, UR, LL, T), where A and T represent the square Jacobian matrices of the individual systems, and LL and UR represent the additional matrices that capture the mutual influences. Thus, the problem of determining the structure of the overall Jacobian matrix was transferred to the determination of the structure of the LL and UR matrices. Furthermore, a source code analysis revealed that the matrices LL and UR are structurally symmetric. Hence, the problem could be reduced to the determination of the LL matrix structure.

Due to the realization of the coupling interface as described in Section 2.1, an `FTRIX` block for the coupling zone is not only created in THY – where the zone is dynamically calculated – but also in ATHLET. This fact was exploited to determine the position and size of the nontrivial blocks, i. e., the structure of the LL matrix, using data tables (`ITABs`) already available in ATHLET and THY. In order to save memory space, the structure of the LL matrix is sent to NuT in CSR format by means of suitable routines provided by NuT's communication interface, see Section 2.3.2.1. The UR matrix, which is structurally symmetric to LL, is sent analogously in CSC format to NuT, where finally the overall Jacobian matrix is assembled from the four individual matrices by invoking the interface of the corresponding assembling routine from the host side. This is done in the ATHLET/CD-driver.

Restriction of parallel execution of function evaluations

In order to save computation time, the evaluation of the function routines in ATHLET and THY ought to be in parallel as much as possible. As an analysis of the thermo-hydraulic model equations of both codes revealed, a completely independent

evaluation of both routines is not always possible because the exchanged coupling variables do not necessarily depend on the state of only *one* system (ATHLET or THY), but may also be determined by the state of the respective other system. Specifically, using the notation from Section 2.1 the relationship $\alpha = \alpha(u)$ and $\beta = \beta(y, \alpha)$ could be demonstrated. So, while α solely depends on the state u, the input β is determined not only by y, but – indirectly – by u as well. For a concrete example see Appendix A.3. From this finding, it was concluded that a partially sequential execution of the ATHLET and THY function routines is inevitable. This required some specific means of synchronization. Details are given in Section 2.3.2.

2.2.2 Derivation of a concept to achieve physical consistency

The thermo-hydraulic models applied in ATHLET and THY differ in various aspects. This is due to the fact that the two codes have been developed with different objectives and requirements for model and code capabilities and have been independent of each other for several decades in the past.

By means of reviewing the coupling-related code sections and by consulting the related literature (especially the models and methods manuals of the codes /KLE 23/, /SCH 23a/), several tasks have been identified as required to foster a physically consistent coupling. These tasks are described below and cast into a realization concept.

Remark 2.4. During the code review, two errors with an impact on physical consistency were detected. One error concerned the calculation of the component mass fractions. The other error concerned the transformation of the ATHLET component `AIR` to the COCOSYS components N_2 and O_2. Both errors have been fixed and the fixes are included in AC^2 2023.

Remark 2.5. So far, the monolithic coupling approach has only been applied to a simple test problem. A description is given in Section 2.3.4. This was done late in the project timeline due to the time-consuming work regarding the overall Jacobian matrix and the adaptation of the ODE control logic. Hence, the considerations below do not include any practical experience based on the new approach. It is expected that future simulations with the monolithic coupling approach will result in additional insights which may lead to further improvements fostering a physically consistent thermo-hydraulic coupling. A set of test cases that would likely provide more insights is discussed in Section 2.2.3.

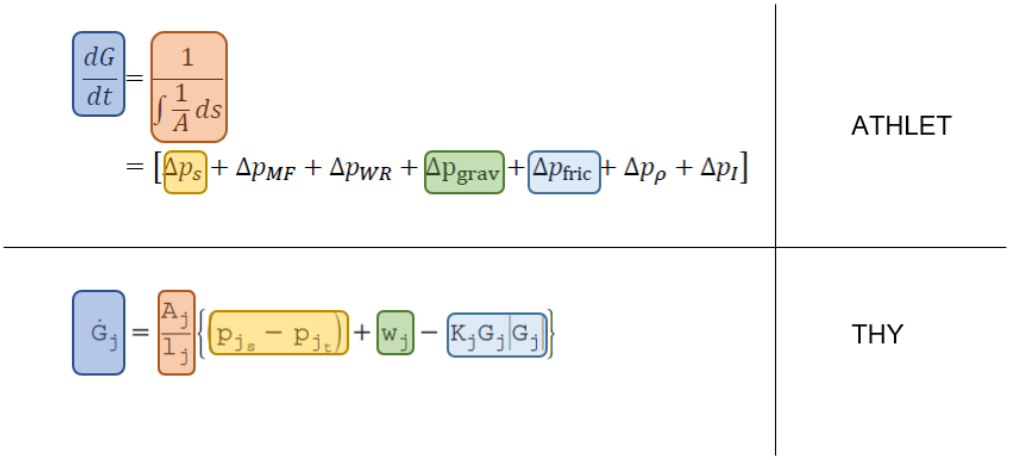

Fig. 2.4 Comparison of the momentum balances in ATHLET (5-equation model, upper part) and THY (junction type `INST`, lower part). Equal terms are marked with boxes of the same color. Dark blue: Time derivative of the total mass flow rate; Orange: Inertia term; Yellow: Gradient of static pressure; Green: Geodetic pressure term; Light blue: Friction and form losses. References: /SCH 23a/, /KLE 23/.

2.2.2.1 Identification of recommended tasks

It is fundamental to ensure that the balances of the conserved quantities mass, energy, and momentum are not violated and that the geometry of the coupling zone, see Fig. 2.2, is the same in both codes.

As opposed to ATHLET, the implementation of the momentum balance in THY is quite rudimentary and, among other terms, does not consider the convective momentum flux Δp_{MF}, as can be seen in Fig. 2.4. If the momentum transport in THY should be calculated similarly to ATHLET, one would have to extend the momentum equation of the junction model accordingly. However, as the overlapping domain of both codes comprises only the coupling CV/zone, but not the coupling *junction*, see Fig. 2.2, an extension of the THY momentum equation is not essential to obtain a consistent code coupling.

Associated with the momentum transport on the junctions, kinetic energy is transported from one control volume into another (respectively from one zone to another, in COCOSYS terminology). When a component (such as nitrogen) flows from the ATHLET into the THY simulation domain, the mass flow rate $\dot{m}$ is accompanied by an energy flow rate $\dot{E}$. The energy flow rate leaving the ATHLET simulation domain is composed of an enthalpy flow rate $\dot{H}$ and a flow of kinetic energy: $\dot{E}_{ATHL} =$

$\dot{H}_{ATHL} + \frac{1}{2}\dot{m}v^2_{ATHL}$. However, since the THY zone model NONEQUILIB does not consider kinetic energy /KLE 23/, the entire inflowing energy for the COCOSYS zone is treated as enthalpy: $\dot{E}_{ATHL} = \dot{E}_{THY} = \dot{H}_{THY}$. Even though the amount of kinetic energy flow is usually small compared to that of enthalpy flow rate, it is obvious that this is an inconsistency which should be corrected by including kinetic energy in the NONEQUILIB zone model.

In reality, kinetic energy is not completely transported, but also dissipates and leads thereby to an increase of internal energy. In contrast to this, dissipation is neglected, both in ATHLET and THY, see for example /SCH 23a/: "in the energy balance equations, [...] the dissipation energy are neglected". The latter applies to the dissipative terms at the phase interface and due to wall friction and form losses. Thus, both codes are consistent regarding their models, but simplify the physics. While the contribution of dissipation to the energy balance turned out to be rather small compared to other terms in the simulations performed up to now, this does not generally have to be the case. Therefore, fixing this error in the energy balance is considered a necessary task, although of lower priority.

Another rather minor issue is the diffusive flux of gas mass and energy. Model terms describing this phenomenon are included in the THY junction model INST by default whereas they have to be activated for ATHLET by the user via an input under the control word MISCELLAN in the data set. The task identified here would be to compare the diffusion models of both codes and unify them if necessary. Nevertheless, this is not a necessary requirement for preparing a consistent coupling since even if there are differences between the diffusion models, these will not have an effect on the coupling itself because the mass and energy flow rates are calculated solely in ATHLET and are provided to THY as boundary conditions. The same reasoning applies to other models such as flow regime calculation.

In addition to the correct balancing of the physical conservation quantities, the usage of uniform fluid properties as well as of uniform numerical values for mathematical and physical constants in ATHLET and COCOSYS is a fundamental prerequisite to reach a physically consistent coupling. For example, using different numerical values for the gravitational acceleration in the two codes could result in artificial pressure differences and corresponding nonphysical mass flows across the coupling interface. Using different fluid property correlations could cause errors in the energy balance, as is shown by the example in Fig. 2.5: The mass flow $\dot{m}$ which is directed from the THY into the ATHLET domain is calculated by ATHLET. Associated with this mass

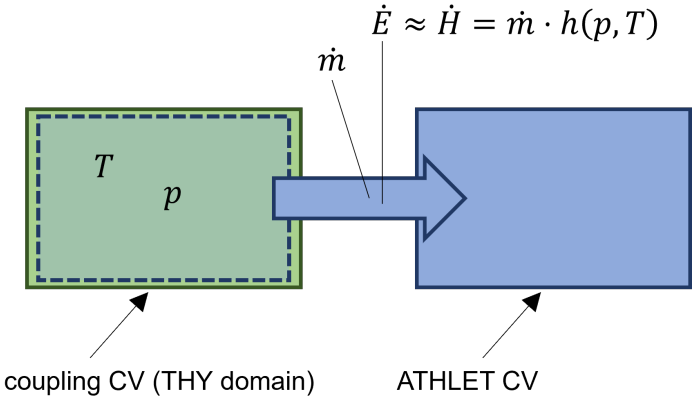

Fig. 2.5 Mass and energy flow rates

flow is an energy flow $\dot{E}$ from one domain into the other. Neglecting the kinetic energy for the sake of simplicity, this energy flow corresponds to an enthalpy flow which can be expressed as the product of the mass flow and the specific enthalpy h of the fluid mixture in the upstream zone:

$$\dot{E} = \dot{H} = \dot{m} \cdot h. \tag{2.7}$$

In both ATHLET and THY, the specific enthalpy is calculated as a function of pressure and temperature

$$h = h\left(p, T\right). \tag{2.8}$$

If the calculation instructions for h of both codes do not match, i. e. $h_{ATHLET}\left(p,T\right) \neq h_{THY}\left(p,T\right)$ for a given pair of values (p,T), the simulation will be physically inconsistent because even though the transported energy flow rate is calculated only once – namely in ATHLET – and is therefore equal for both codes, it does not match the transported mass flow for one of the codes.

For example, if ATHLET calculates a mass flow rate of 5 kg/s and an accompanying enthalpy flow rate of $\dot{H} = 5\,\mathrm{kg/s} \cdot h_{ATHLET}\left(10\,\mathrm{bar}, 80\,^{\circ}\mathrm{C}\right) = 5\,\mathrm{kg/s} \cdot 365\,\mathrm{kJ/kg} = 1825\,\mathrm{kJ/s}$, obviously 5 kg/s and 1825 kJ/s are withdrawn from the THY domain.[2] Now, if THY uses different fluid property correlations than ATHLET and therefore associates an enthalpy flow rate of $\dot{H} = 5\,\mathrm{kg/s} \cdot h_{THY}\left(10\,\mathrm{bar}, 80\,^{\circ}\mathrm{C}\right) = 1820\,\mathrm{kJ/s}$ with the given mass flow rate, too much energy (namely $(1825 - 1820)\ \mathrm{kJ/s} = 5\,\mathrm{kJ/s}$) is withdrawn from the THY domain. The best way to avoid this problem is the usage of a unified fluid property library.

[2]The calculated specific enthalpy corresponds approximately to that of nitrogen at 10 bar and 80 °C.

2.2.2.2 Realization concept

Based on the tasks from the previous section that were identified as required to foster a physically consistent coupling within the context of the provided numerical environment, a concept for their realization was derived. The tasks are sorted by urgency in descending order and divided into working steps.

An implementation of the concept within the current project was not feasible. Instead, it is advisable to define it as a separate working package within the framework of a subsequent project.

Unification of mathematical and physical constants

This is a necessary task to achieve a physically consistent coupling. Working steps are as follows:

- Identification of all modeling-related constants used by ATHLET and COCOSYS (this comprises both global module variables and local variables in subroutines)
- Definition and implementation of a container structure for the constants (programming library, plugin, etc.) that can be accessed from ATHLET as well as from COCOSYS
- Moving the constants from ATHLET and COCOSYS into the container structure
- Code modifications of ATHLET and COCOSYS to use the container structure
- Verification calculations to check the implementation

Current activities within the research project RS1604 (SIWAP) aim for a unification of the heat transfer models of ATHLET and THY. Since these models also access physical constants, some constants have already been included in a static program library (`ac2shared`) which can be integrated in ATHLET and COCOSYS. Whether the chosen container structure is ultimately the most suitable one will become clear in the further course of RS1604. Definitely, there are synergy effects with the working steps listed here.

Unification of fluid properties

This is a necessary task to achieve a physically consistent coupling. Concerning liquid water and vapor, physical consistency has already been reached: As an external AC^2 developer, the Zittau/Görlitz University of Applied Sciences (HSZG) developed a new fluid property package based on the Spline-Based Table Look-up (SBTL) method which can already be included both in ATHLET and COCOSYS in the form of a plugin. A fluid property plugin for non-condensable gases is currently being worked on by

HSZG. Furthermore, HSZG is developing mixing models for real gases and adapting the ATHLET and COCOSYS implementations accordingly. These developments are expected to be completed in early 2024. After that, the following work is planned to be accomplished by GRS:

- Transfer of models from external to internal GitLab server
- Code review and smaller refactoring where necessary
- Transfer of test cases to GitLab CI
- Updating the AC^2 model manuals

Consideration of kinetic energy in the `NONEQUILIB` zone model of THY

In principle, this is a necessary task to achieve a physically consistent coupling, albeit of rather minor importance since the kinetic energy is usually small compared to enthalpy (for example, nitrogen that flows with a velocity of 100 m/s at a pressure of 10 bar and a temperature of 80 °C has a specific kinetic energy of 5 kJ/kg and a specific enthalpy of 365 kJ/kg). Working steps are:

- Based on the energy balance: Derivation of a THY zone model equation which includes kinetic energy terms
- Implementation of the derived kinetic energy terms in the zone models
- Verification calculations to check the implementation
- Updating the AC^2 model manuals

Consideration of momentum flux in the `INST` junction model of THY

Including the momentum flux in the `INST` junction model would lead to a harmonization of the physical models in ATHLET and THY and, moreover, would enable momentum transfer from one simulation domain to the other. However, it is not essential for a physically consistent coupling. Working steps are:

- Based on the momentum balance: Derivation of a THY junction model equation which includes the momentum flux
- Consideration of additional dependencies by extension of the `FTRIX` block structure
- Implementation of the momentum flux term
- Verification calculations to check the implementation
- Updating the AC^2 model manuals

Compared to the other tasks, this is a more complex one which requires a comparatively higher amount of resources.

Consideration of dissipation in the energy balances of both ATHLET and THY
Since its absolute value is rather small in many simulation cases, the consideration of the dissipation in the energy balances of ATHLET and THY is not a high-priority task. Nevertheless, it is essential for correctly capturing the physics. Working steps are:

- Modification of the ATHLET and THY energy balances to include dissipation terms
- Modeling and implementation of the dissipation terms
- Verification calculations to check the implementation
- Updating the AC^2 model manuals

2.2.3 Samples for testing the ATHLET-COCOSYS coupling

Within the framework of this project, various samples for testing the newly developed thermo-hydraulic ATHLET-COCOSYS coupling have been created. Accompanied by a description, these samples are stored on the GRS GitLab server for internal verification purposes. The samples can be subdivided into two types. First, purely "technical" test cases were developed which can be used for verification of the coupling. Second, physics-oriented test cases were considered whose results can be reasonably interpreted (at least qualitatively) and which thus can be used for further verification of the physics of the coupling, e. g., by comparison with reference runs, such as ATHLET stand-alone simulations. The focus was on creating samples which provide the necessary insight while being as simple as possible. Since they are developed for coupled calculations, each of the samples consists of both an ATHLET and a COCOSYS data set.
Note that due to the time-consuming work involved in creating the overall Jacobian matrix, only the simple example described in Section 2.2.3.1 was calculated with the final program version. The findings and conclusions from the test calculation are presented in Section 2.3.4.

2.2.3.1 Zone structure samples for verification purposes

Small samples of varying complexity were created, the main purpose of which is to check the correctness of the identified functional dependencies in the Jacobian matrix

of the integral system, i. e., of the combined ATHLET/THY Jacobian matrix. On the contrary, no emphasis was put on physical significance.

Two of these examples are presented here. One is on the lower end of complexity while the second one covers the area of high complexity for the purpose of verification. The others – which are not described here – are in the range between these two.

Simple sample

The simulation domain of the simple sample consists of three thermo-fluid dynamic objects (TFOs) in the ATHLET data set of the sample. The accompanying COCOSYS data set defines only one zone for simplicity. The topology of the setup is shown in Fig. 2.6. The graphic is based on an ATHLET Input Graphic (AIG) of the geometry defined in ATHLET, i. e. the COCOSYS zone is not shown, but has the same volume and position as `COCOZONE`.

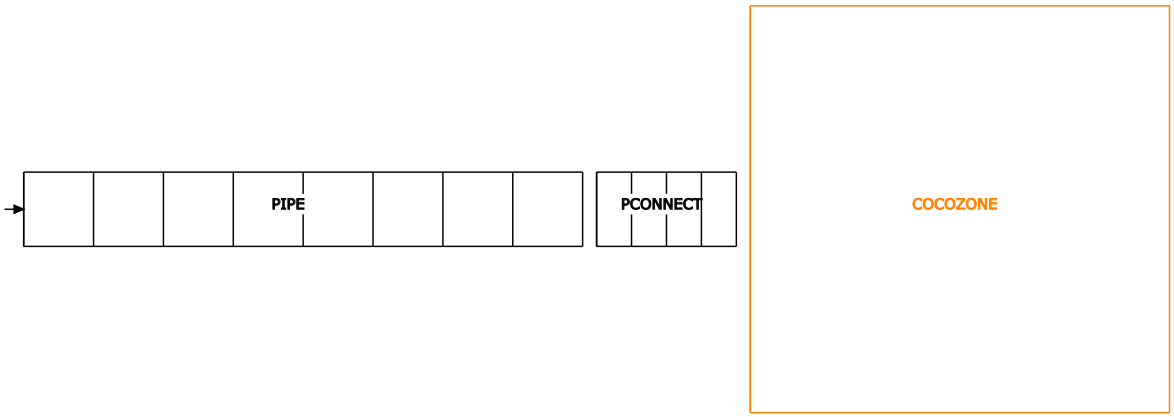

Fig. 2.6 AIG of the simple sample

The COCOSYS and ATHLET domain are coupled using the external coupling data interface of ATHLET. The coupling interface for zone quantities (= CV-related quantities) is the TFO `COCOZONE`; the interface for flow quantities is the junction connecting `PCONNECT` and `COCOZONE`. Those TFOs which are dynamically calculated in ATHLET are drawn in black in Fig. 2.6. They are calculated using ATHLET's 6-equation model. The coupling TFO `COCOZONE` (drawn in orange in the figure) is dynamically calculated by the module THY. Its thermodynamic state is transferred to the ATHLET-CV in `COCOZONE` via the coupling interface. The applied zone model in THY is the `NONEQUILIB` model.

At the beginning of the simulation, all TFOs and the COCOSYS zone are filled with a mixture of 10 Vol–% vapor and 90 Vol–% air at 60 °C and 1 bar. The calculation domain is thus in an equilibrium state. After a short zero-transient phase, vapor of

300 °C is injected into the bottom part of object PIPE (the injection point is marked by the small arrow in Fig. 2.6). This makes the setup leave the equilibrium state as the pressure rises in the ATHLET part, which leads to a mass flow to COCOZONE.
Simulation results for this configuration can be found in section 2.3.4.

Complex sample

The sample shown in Fig. 2.7 is clearly more complex compared to the previous one. Instead of one, there are two coupling interfaces for the zone quantities – namely CVs 18 and 33, both drawn in orange color, belonging to the TFOs COCOEXT1 and COCOEXT2, respectively. These zones as well as the zones and junctions drawn in green color are dynamically calculated by THY. The objects drawn in black are dynamically calculated by ATHLET. Junctions (14), (18), (23), (24), and (22) are the interface junctions which provide the flow quantities calculated by ATHLET as boundary conditions for THY. Apart from these thermo-hydraulic linkages of both calculation domains, a further linkage exists in the sample: As indicated by the red dashed arrow in the figure, the GCSM controller of the valve on junction (14) is designed in such a way that the opening degree of the valve is influenced by the thermo-hydraulic state in CV 4 (which is from a topological point of view rather distant from the coupling interfaces). The complexity of the sample is further increased by the fact that the modeled TFOs are initially filled with hydrogen, nitrogen, and helium in varying mixing ratios. Coupling the ATHLET and THY simulation domains as shown in Fig. 2.7 might be a rare case for the simulation of Gen. II or III light-water reactors. However, future application of AC^2 to advanced reactor concepts, such as SMRs or pool-type reactors, will presumably lead to even more complex models. In any case, the rather complex sample is suitable to verify the implementation of the coupling – especially the calculation of both the Jacobian matrix and the seed matrix of the integral system.

Remark 2.6. In the course of creating the complex sample, a general bug in ATHLET (which also concerned stand-alone simulations) regarding the FTRIX links of GCSM-controlled valves was detected and thereafter fixed. The fix has already been included in AC^2 2023.

2.2.3.2 Physics-oriented samples for verification purposes

The main focus of the physics-oriented samples is to obtain reasonable and physically meaningful simulation results. Especially the second physics-oriented sample (*water pool heat-up*, see below) has its focus on the use case of a thermo-hydraulic coupling

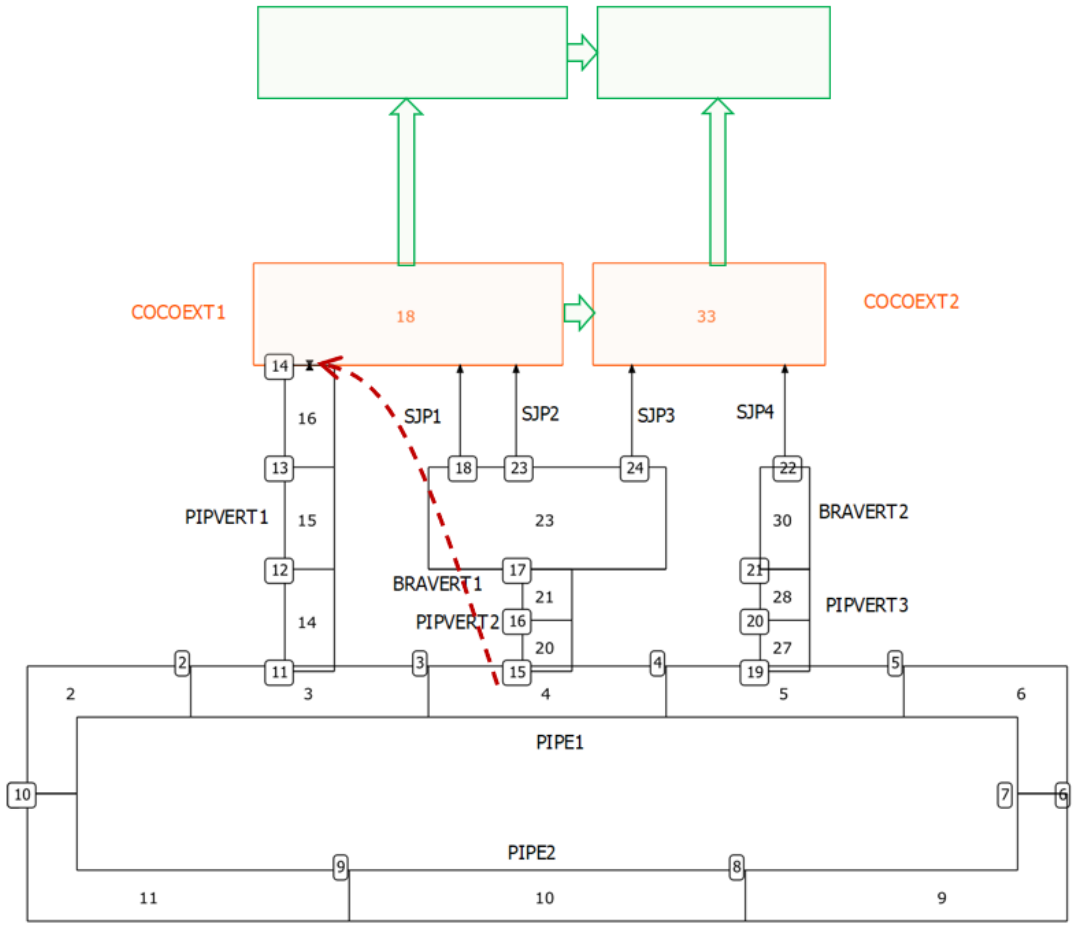

Fig. 2.7 AIG of the most complex technical sample

of ATHLET and COCOSYS in the atmosphere above a water pool inside a reactor containment as shown in Fig. 2.8.

Helium injection

This sample is of a simple geometry as shown in Fig. 2.9. The system is initially filled with pure nitrogen and becomes filled with helium during a transient according to the characteristic shown in Fig. 2.10. Both, a one-channel and a two-channel model of the system were prepared, see Fig. 2.11.

- **1-channel model**
  The ATHLET data set comprises a simple pipe `HOMOPIPE` with three homogeneous CVs and a fill at its bottom as well as a branch TFO `DOME_ATH`. The latter is the coupling TFO, i. e. it is dynamically calculated by THY.
- **2-channel model**
  To enable convection loops, both `HOMOPIPE` and `DOME_ATH` are subdivided into two

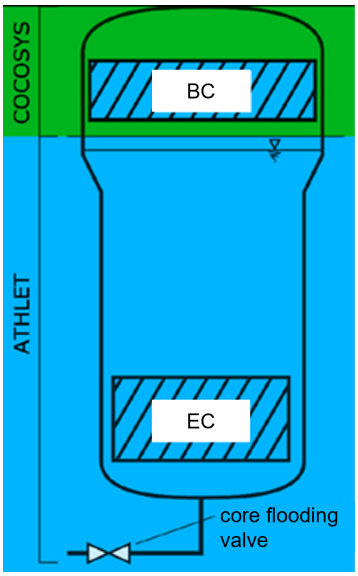

Fig. 2.8 ATHLET-THY coupling interface above water pool (as an example, the INKA flooding pool vessel with emergency condenser (EC) and building condenser (BC) is shown). Adapted from /BUC 18/.

parallel channels. As indicated by the vertical arrow in Fig. 2.11 (right side), the helium is injected into the bottom right CV. As a consequence of the subdivision into parallel channels, two ATHLET-THY interfaces have to be defined as indicated by the color coding in the figure.

Water pool heat-up

This sample comes with a mixture level in the lower part of the system, see Fig. 2.12. Initially, there is pure liquid water below the mixture level and a pure nitrogen atmosphere above it. The temperature of the fluids is 20 °C, the initial pressure is at 1 bar. After a short zero-transient phase of 10 seconds, heat is added to the liquid below the mixture level in the transient calculation so that the liquid temperature rises and vapor is produced and released into the atmosphere where it mixes with the nitrogen. Note: While gas, vapor and liquid water can be transported from one code domain to the other across the coupling interface (both, one- and two-phase flow is possible), the *mixture level tracking information* is not transferred via the interface. Therefore, the coupling CV/zone has to be always homogeneous; this is ensured by input checks, if the interface via CW `COCOSYS` PW `COUPLING` is used (ATHLET 3.4). For a CW `EXT` coupling, this constraint has to be checked by the user; however, this type of interface will be obsolete soon for ATHLET-COCOSYS couplings.

Again, both a one-channel and a – more realistic but more complex – two-channel model of the system were prepared. The AIGs of both models are shown in Fig. 2.13.

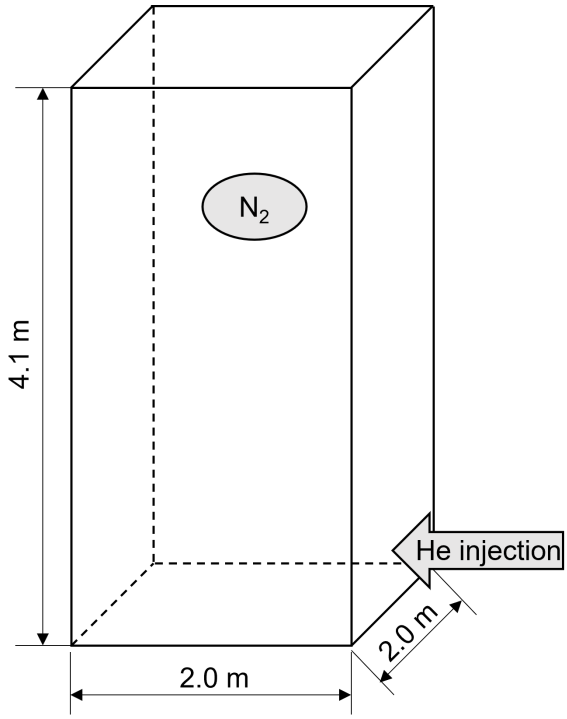

Fig. 2.9 Dimensioned sketch of the helium injection sample

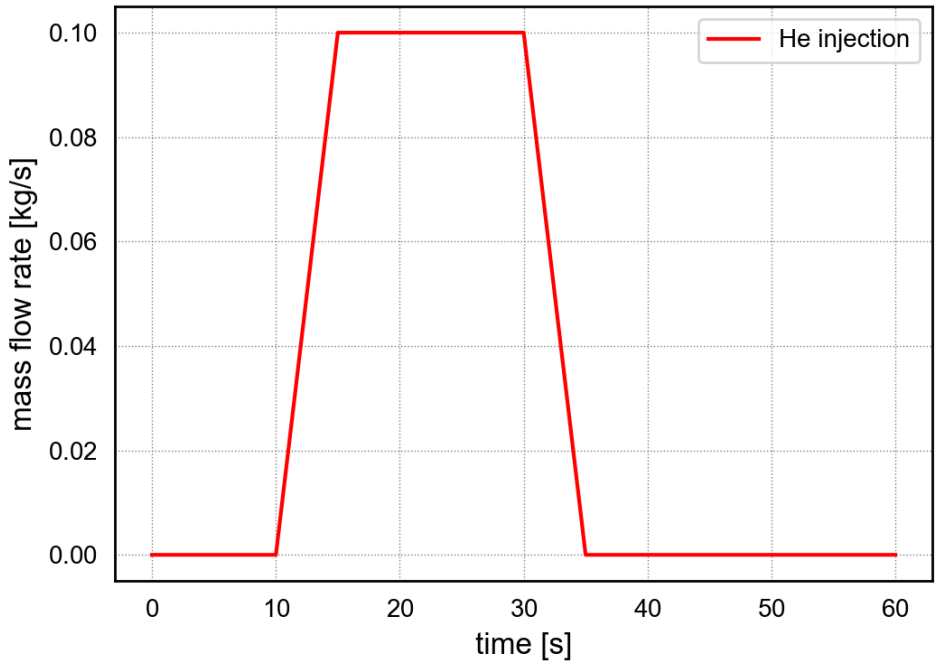

Fig. 2.10 Between 10 s and 35 s, helium is injected at the lower part of the system

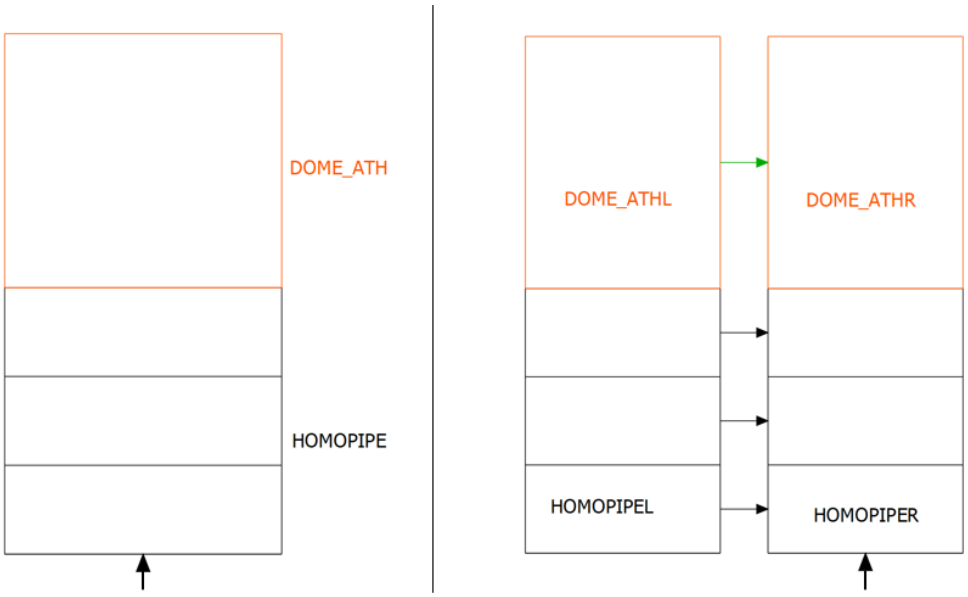

Fig. 2.11 AIG of the He injection (left: 1-channel model; right: 2-channel model). Black: Network objects of the ATHLET domain. Orange: Coupling CVs (dynamically calculated by THY). Gray: Junction of the THY domain.

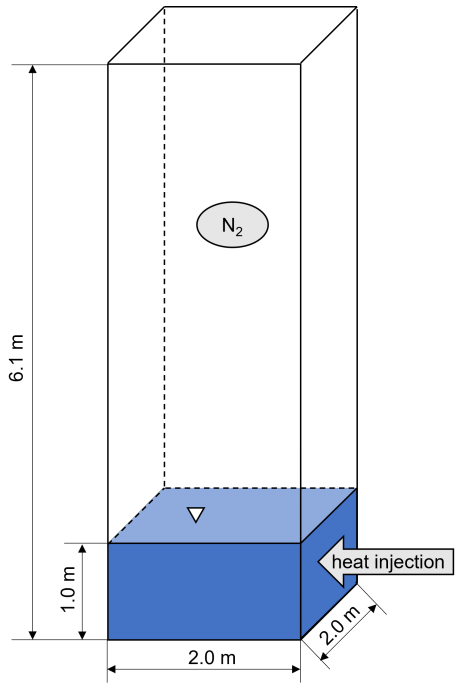

Fig. 2.12 Dimensioned sketch of the water pool heat-up sample

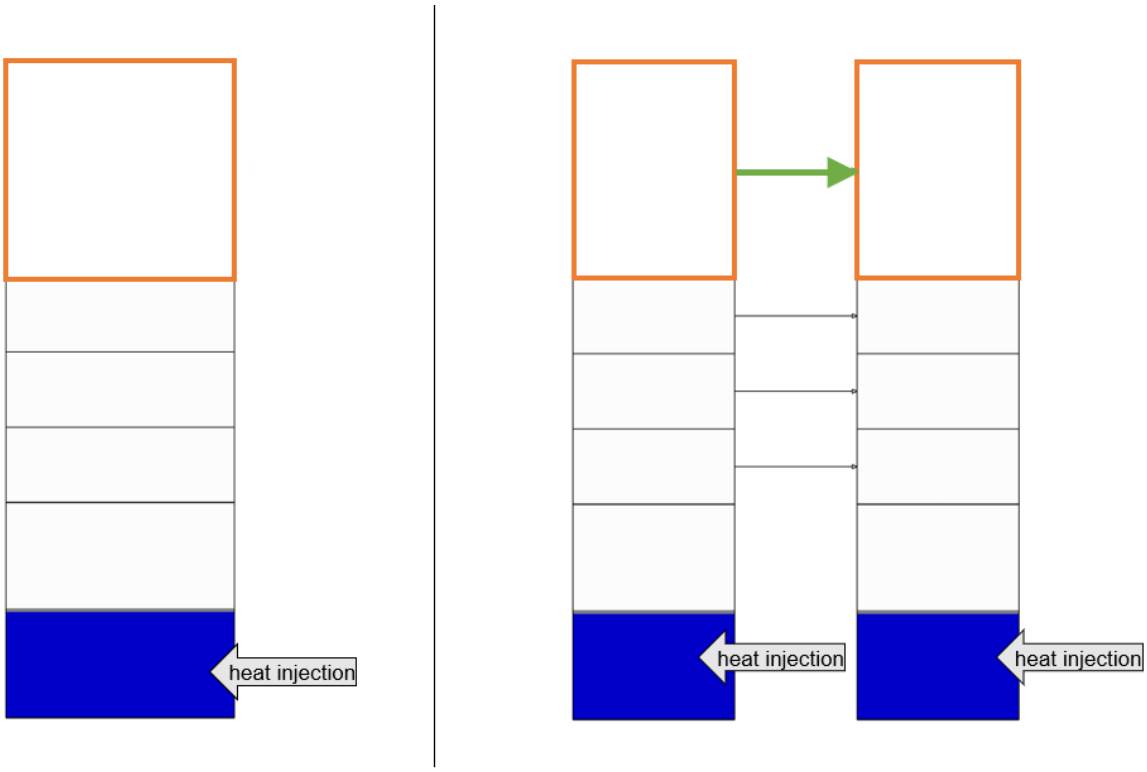

Fig. 2.13 ATLAS picture of the water pool (left: 1-channel model; right: 2-channel model). Black: Network objects of the ATHLET domain. Orange: Coupling CV (dynamically calculated by THY). Gray: Junction of the THY domain. In the 2-channel case, heat is added equally to both parallel "sumps".

2.3 ODE-numerics for single and coupled systems

Establishing a new approach to ODE numerics in AC^2 in order to handle single and also coupled systems via a monolithic ansatz required considerable modifications in the system codes ATHLET and COCOSYS (module THY) as well as in the ATHLET/CD-driver and NuT. The approach separates method logic from control logic. Method logic is handled by NuT whereas the existing control logic in ATHLET and THY was adapted to work with the new methods. A common endeavor that required the cooperation of all listed components was the determination of a Jacobian matrix for the overall system. The accessibility of such Jacobians is key for the monolithic approach to work. This section gives details on the several implementation tasks.

The implementation was accomplished to a degree that a test problem could be run. Further modifications are necessary, though. See also the conclusions in Chapter 5 for further information.

2.3.1 Establishing a feature in NuT to execute certain ODE-methods

Work for this task was done on two levels. First, NuT was extended internally to provide the means to store and execute certain ODE methods for a given time-step. Second, this new ODE logic was incorporated in NuT's interface architecture to allow for suitable access.

2.3.1.1 ODE methods in NuT

A monolithic approach to coupled computations interprets the combined subsystems as one large but single system. Hence, the problem to solve is given in the form of a classical initial value problem. Considering each subsystem on its own (pure ATHLET or COCOSYS computations) the corresponding mathematical problem is also an initial value problem as it is described in Section 2.1. This comes with the advantage that the implemented logic in NuT to execute certain ODE methods can be used for single code simulations as well as for coupled computations.

In order to keep the notation simple in this section, a given initial value problem is denoted by

$$y' = f(t, y), \quad y(t_0) = y_0, \tag{2.9}$$

may it arise from a monolithic approach to coupled computations or from single code calculations.

Opting for one-step methods

The methods that can be employed via the new implementations in NuT belong to the class of so-called one-step methods. This is in line with the default approaches implemented in ATHLET and THY respectively. The schemes there can be interpreted as one-step methods as well. For further details, see the paragraph *AC² and the requirement for implicit methods* below.

In the given context of AC² it makes sense to focus on such methods since the codes may encounter discontinuities in the right hand side f of the ODE system in (2.9) during the time integration process. One-step methods, in contrast to multi-step methods, do not rely on information from previous time steps to ensure a certain order or degree of stability. Discontinuities require dedicated handling, see Section 2.3.3 below. After this is done, a one-step method can simply proceed as usual whereas a multi-step method requires a build-up phase since the smoothness assumptions

regarding the previous time steps are violated by the discontinuity. The build-up phase is usually done by means of one-step methods anyway.

Basic Runge–Kutta scheme

The ODE methods that can be invoked via NuT are one-step methods that may be of different types but they are all derived from the same general Runge–Kutta (RK) scheme: For a given time step h from some t_0 to t_0+h the general s-stage RK scheme is described via

$$k_i = hf\Big(t_0 + c_i h, y_0 + \sum_{j=1}^{s} a_{ij} k_j\Big), \quad i = 1, \ldots, s, \qquad y_1 = y_0 + \sum_{j=1}^{s} b_j k_j, \tag{2.10}$$

where the choice of coefficients $c := (c_1, \ldots, c_s)^T \in [0,1]$, $\mathcal{A} := (a_{ij})_{i,j=1,\ldots,s} \in \mathbb{R}^{s\times s}$, and $b := (b_1, \ldots, b_s)^T \in \mathbb{R}^s$ uniquely determines a specific method. A method is said to be of order p if for sufficiently smooth f it holds that

$$y(t_0 + h; y_0) - y_1 \in \mathcal{O}(h^{p+1}). \tag{2.11}$$

The k_i are called stage derivatives whereas the stage values Y_i are defined by

$$Y_i = y_0 + \sum_{j=1}^{s} a_{ij} k_j, \quad i = 1, \ldots, s. \tag{2.12}$$

Throughout, it is assumed that $c_i = \sum_{j=1}^{s} a_{ij}$. Hence, Y_i approximates $y(t_0 + c_i h; y_0)$ at least up to order one. Introducing the stage shifts $z_i = Y_i - y_0$ it obviously holds that

$$Z = (\mathcal{A} \otimes I)K \qquad \text{where} \quad Z := \begin{pmatrix} z_1 \\ \vdots \\ z_s \end{pmatrix}, \quad K := \begin{pmatrix} k_1 \\ \vdots \\ k_s \end{pmatrix}. \tag{2.13}$$

This relation comes in handy when an implementation of methods is considered, see the paragraph *Working with* z_i below.

If it holds that $a_{ij} = 0$ for all $j \geqslant i$ the scheme is explicit. Thus, the k_i can be computed directly from (2.10) in a successive manner. If there is an $a_{ij} \neq 0$ with $j \geqslant i$ the scheme becomes implicit. In such a case, an implementation has to employ some Newton-type process in order to determine (approximations for) the stage quantities in an iterative way.

AC² and the requirement for implicit methods

As it is described in /SCH 23a, Ch. 6/ ATHLET's system in (2.9) must be considered as stiff. In the given case, this is due to the specific structure of the spectrum of the Jacobian $\partial f/\partial y$. A general consequence of stiff problems is that they lead to poor

performance of explicit methods. Hence, some implicit treatment is necessary. See /HAI 96/ for a thorough discussion on that topic.

To counter stiffness, ATHLET employs an extrapolation ansatz based on the linearly implicit Euler method, see /SCH 23a, Ch. 6/. Since it is mainly the spectrum of the Jacobian that is responsible for the stiffness, employing a linearly implicit method is appropriate, see the discussion on the stability of linearly implicit methods in /HAI 96, Sec. IV.7/. Linearly implicit methods come with the advantage that the related Newton-type process is reduced to a single iteration per stage, see (2.15) below. This may save computational time. On the other hand, the quality of the involved Jacobian approximation must be monitored closely in order to ensure stability. ATHLET provides several means to support such monitoring. For further details see Section 2.3.3.

THY uses the same basic ansatz as ATHLET but the order of the extrapolation can be adapted dynamically during the time integration process up to very high orders. ATHLET uses a fixed order of three. The decision for the latter is based on empirical data, providing a compromise between performance and stability. ATHLET's fixed order justifies to also work with methods of fixed order in the context of a monolithic approach to coupled computations. Hence, no extended order adaptions are considered for the implementations in this project. Exceptions are discussed in Section 2.3.3.

Types of methods in NuT

In order to be useful within the context of AC^2 the framework of ODE methods in NuT must allow for the execution of (at least) linearly implicit methods. The implementation goes a step further to cover methods that execute a predefined number of Newton-type iteration steps where each stage may define its own number which can be greater than one. Hence, linearly implicit methods are included but more sophisticated ones are possible as well. The following types of methods are supported:

- Linearly implicit methods – /HAI 96, Sec. IV.7/
  These methods are also called Rosenbrock–Wanner (ROW) methods. They are derived from diagonally implicit RK methods, i. e. in (2.10) the matrix $\mathcal{A}$ is a lower left triangular matrix. Accordingly, the stages are diagonal stages, there is no dependency on later stages. As mentioned before, a single Newton-type step is performed per stage. Because of that, initial guesses k_i^0 come into play for the definition of the scheme. For $i = 1, \ldots, s$ let

$$\gamma_{ii} k_i^0 = \sum_{j=1}^{i-1} -\gamma_{ij} \cdot k_j^1, \quad \gamma_{ii} := a_{ii}, \quad \text{and} \quad \alpha_{ij} := a_{ij} - \gamma_{ij}. \tag{2.14}$$

Then one Newton-type step applied to the system (2.9) at (t_0, y_0) results in

$$\begin{aligned}\Big(I - h\gamma_{ii}J\Big)\delta k_i^0 &= -k_i^0 + hf\Big(t_0 + h\cdot\sum_{j=1}^{i-1}\alpha_{ij}, y_0 + \sum_{j=1}^{i-1}\alpha_{ij}k_j^1\Big) \\ &\quad + h^2\frac{\partial f_0}{\partial t}\cdot\sum_{j=1}^{i}\gamma_{ij}, \\ k_i^1 &= k_i^0 + \delta k_i^0, \qquad\qquad i = 1,\ldots,s, \\ y_1 &= y_0 + \sum_{j=1}^{s} b_j k_j^1.\end{aligned} \tag{2.15}$$

The matrix J makes the linear system in (2.15) nontrivial, and for the sake of stability of the above scheme it is crucial that J provides a decent approximation to the Jacobian $\partial f/\partial y$ at (t_0, y_0). The vector $\partial f_0/\partial t$ stands for $\partial f/\partial t$ at (t_0, y_0). Some generous approximation may be used as well. Because usually, $\sum_{j=1}^{i-1}\alpha_{ij}$ is close to or equal to c_i, and hence $\sum_{j=1}^{i}\gamma_{ij}$ is close to or equal to zero. Combined with the factor h^2 the impact of $\partial f_0/\partial t$ on (2.15) is comparatively low.

The default extrapolation ansatz in ATHLET and THY falls into this category of linearly implicit methods for any fixed extrapolation order. According to the above discussion on dynamic changes of the order it suffices to support fixed order in the context of monolithic computations. Hence, by including linearly implicit methods in NuT the default method in AC² can be taken into account as well.

- FiterRK methods of high stage order – /STE 17b, Sec. 3.3/
  These methods extend the finite iteration idea of ROW methods to RK schemes that come with a matrix $\mathcal{A}$ of the shape

$$\mathcal{A} = \begin{pmatrix} a_{11} & \cdots & a_{1m} & & & \\ \vdots & \ddots & \vdots & & & \\ a_{m1} & \cdots & a_{mm} & & & \\ a_{m+1,1} & \cdots & \cdots & a_{m+1,m+1} & & \\ \vdots & & & & \ddots & \\ a_{s1} & \cdots & \cdots & \cdots & \cdots & a_{ss} \end{pmatrix}, \quad \begin{aligned} &A_m := (a_{ij})_{i,j=1,\ldots,m}, \\ &a_{ii} \neq 0, \\ &\quad i = m+1,\ldots,s. \end{aligned} \tag{2.16}$$

  The block A_m allows for taking methods into account that are of high stage order, i. e. methods for which each stage value Y_i, $i = 1,\ldots,s$, fulfills the order condition (2.11) w. r. t. its corresponding true value $y(t_0 + c_i h; y_0)$. Such methods do not suffer from certain order reduction phenomena, see /HAI 96, Sec. IV.15/ and especially /HUN 03, Sec. II.2/. The benefits of an extension of A_m by means of subsequent diagonal stages are discussed in /BUT 90/. In order to be efficient, a single point

spectrum of $\mathcal{A}$ and therefore of A_m is to be preferred. This can be ensured by the techniques discussed in /HAI 96, Sec. IV.8/ and /STE 17a/. In order to avoid linear systems of dimension m-times the dimension of the system in (2.9) a Schur decomposition of A_m is applied. This way, m successive linear systems like in (2.15) are solved to implement one iteration step for the block stages. A Schur decomposition is the result of a similarity transformation with orthogonal matrices. Hence, the transformation is robust and does not introduce any additional error.

The finite iteration idea in this context is based on /STE 17b, Satz 3.26/. Depending on the quality of the initial guesses k_i^0, $i = 1, \ldots, m$, the block stages may require more than one Newton-type iteration step to produce sufficient results. However, the required number is still predefined. Also, the subsequent diagonal stages can usually be computed within a single iteration and still ensure high stage order.

- Explicit RK methods – /HAI 93, Sec. II.1/
  Explicit RK methods are naturally supported by the scheme (2.15) via setting $k_i^0 = 0$, $J = 0$, and discarding the $\partial f_0/\partial t$ term. The implementation takes care of the modified linear algebra in a transparent way, see also Fig. 2.17. Though the systems in AC^2 generally require an implicit approach, explicit methods come in handy for exception handling in case of discontinuities. Details are given in Section 2.3.3.

Remark 2.7. An implementation of an implicit scheme based on (2.10) also uses a finite number of iteration steps per stage to produce approximations to the implicitly defined quantities. However, the number of steps is controlled by an error monitor. Hence, it is not predefined. Such an approach can be approximated by above method types by simply raising the number of predefined iteration steps per stage to some relatively large number like five or six. At the current stage of development this suffices to get an idea of what an implicit scheme could provide. Further work may be pursued in later projects if considered suitable. For this project the focus was on providing linearly implicit methods and as a natural extension to it FiterRK methods of high stage order. This way, the default method of AC^2 can be resembled (for fixed extrapolation order) and some more complex methods can be tested as well.

Remark 2.8. Technically, it is also possible to execute FiterRK methods of high stage order in an explicit way. Simply set $J = 0$ in (2.15) and discard the $\partial f_0/\partial t$ term if $\sum_{j=1}^{i} \gamma_{ij}$ isn't zero anyway. The order doesn't change as long as f stays sufficiently smooth. In case of discontinuities it makes more sense, though, to switch to a method of order one, see Section 2.3.3.

Working with z_i

Regarding an implementation of methods it is advantageous to work with the stage shifts z_i instead of the stage derivatives k_i or values Y_i. Stage shifts are easy to interpret since they directly reflect the change of solution variables for a given (sub-)time step. Also, they are less prone to rounding errors than stage values due to their incremental nature.

Usually, $\mathcal{A}$ is nonsingular. In such a case, (2.13) can equivalently be stated as

$$(\mathcal{A}^{-1} \otimes I)Z = K.$$

By means of the above relation, (2.10) can easily be rewritten to work with the stage shifts z_i instead of the derivatives k_i. The calculations in (2.15) become

$$\begin{aligned}(-h^{-1}a_{ii}^{-1}I + J)\delta z_i^0 &= h^{-1}k_i^0 - f(t_0 + \sum_{j=1}^{i-1}\alpha_{ij}h, y_0 + z_i^0) - h\partial f_0/\partial t \cdot \sum_{j=1}^{i}\gamma_{ij} \\ z_i^1 &= z_i^0 + \delta z_i^0, \quad i = 1, \ldots, s, \\ y_1 &= y_0 + (b^T\mathcal{A}^{-1} \otimes I)Z^1\end{aligned} \tag{2.17}$$

where Z^1 comprises $z_1^1, \ldots, z_s^1$. The stage derivatives k_i^0 are determined by means of the previous z_j, $j < i$. In the above description k_i^0 is used for the sake of a compact notation. In contrast to (2.13) the linear system already includes a scaling by $-h^{-1}a_{ii}^{-1}$. This saves computation time since J isn't scaled anymore. Simply its diagonal is altered by $-h^{-1}a_{ii}^{-1}I$.

In case of an explicit first stage, i. e. $c_1 = 0$ and hence $z_1 = 0$, $\mathcal{A}$ is singular. However, the same ideas can be applied if $\mathcal{A}$ is substituted by $\mathcal{A} + e_1e_1^T$ and z_1 by $k_1 = f(t_0, y_0)$ in (2.13).[3]

For an explicit RK method $\mathcal{A}$ is a strictly lower triangular matrix and therefore singular too. To work with the stage shifts z_i in this case the following trick can be applied:

1. Add some nonsingular diagonal matrix D to $\mathcal{A}$, resulting in a nonsingular matrix.
2. Apply the transformations that lead to (2.17), just like it is done for the regular nonsingular case.
3. Set $J = 0$, $k_i^0 = 0$, discard $\partial f_0/\partial t$, and compute (2.17).

Details on the transformations to get from (2.15) to (2.17) are thoroughly discussed in /STE 17b, Subsec. 3.3.4/. There, handling of block stages is comprehensively covered as well.

[3]The vector e_1 denotes the first unit vector in $\mathbb{R}^s$.

Remark 2.9. It is part of the implementation that NuT does not determine y_1 but $\Delta y_0 := y_1 - y_0$. This comes with the advantage that NuT does not require the knowledge of y_0 to execute (2.17) since f-evaluations are done by the host. It is the host's responsibility to calculate the final value y_1 by adding Δy_0 to y_0.

The pre method concept

In case of FiterRK methods with block stages, i. e. nontrivial A_m, it makes sense to ask for start approximations z_i^0, $i = 1, \ldots, m$, better than zero. Setting $z_i^0 = 0$ is a valid and always available option but also requires the most iteration steps. In order to provide better z_i^0, the concept of a pre method was implemented. See also Fig. 2.15. To be precise, a cascade of pre methods can be used where each pre method feeds data to its direct successor till finally the actual method receives its data. Practically though, a single pre method usually suffices.

The implementation is done in a way that any method can be used as a pre method. The developer decides what combination is suitable. To transfer data from one method to another, interpolation may become necessary since the c_i-values are not necessarily the same. Currently, linear interpolation of the following kinds is supported:

- $\lambda \cdot z_\mu^{pre}, \quad \mu := \arg\max_i c_i^{pre},\ \lambda \in [0, 1]$,
- $z_{i-1}^{pre} + \lambda \cdot (z_i^{pre} - z_{i-1}^{pre}), \quad i = 1, \ldots, s^{pre},\ z_0^{pre} := 0,\ \lambda \in [0, 1]$.

Such interpolation leads to approximations z_i^0 of first order. If deemed necessary more sophisticated interpolation concepts can be added.

Support of host-sided control logic

The implemented ODE feature is focused on executing certain methods. There is no control logic like step size selection, redoing of steps or error calculation involved. These are things that are left to the host application(s) since they may include decisions that are particular to the application. However, support for certain features is provided:

- Error estimation

  As it is explained in Remark 2.9 NuT computes $\Delta y_0 = y_1 - y_0$ as an approximation for $y(t_0 + h; y_0) - y_0$. Additionally, an error vector err_{est}^{loc} is determined. Formally, the latter is constructed like Δy_0 but based on some suitable $\delta b = b - \hat{b}$ instead of b. Via $\hat{b}$ a method of higher order is realized, based on the same stages as Δy_0. This way, the error of y_1 can be estimated. NuT solely provides the vector information err_{est}^{loc}. The host applies an appropriate error norm $\|\cdot\|_{err}$ to it. See Section 2.3.3 for details.

- Contraction checks for the Newton-type process
  The supported methods perform a predefined number of Newton-type iteration steps. Nonetheless, it is a valid question whether the iteration would actually converge. In this context it appears reasonable to check the iterates for contraction, i. e., $\|\delta z_i^{\ell+1}\|_{err}/\|\delta z_i^{\ell}\|_{err} < 1$. If such a contraction check fails the host may reduce the step size and/or update the Jacobian.

  NuT comes with the option to provide input vectors $\delta z_i^{\ell+1}$ and δz_i^{ℓ} for a contraction check to be done by the host. If no stage goes for a second iteration step, e.g. if a linearly implicit method is considered, an additional iteration step may be defined for a stage, solely for the sake of producing contraction information. Such a strategy is also applied in ATHLET's default algorithm. NuT may provide contraction information based on block stages as well. Due to the intertwining of the block stages, $2 \cdot m$ vectors instead of just two have to be taken into account. In order to process the information the host must provide a suitably adapted error norm.

- First order approximations
  The host may demand some of its solution variables to be computed by a low approximation scheme. This may be combined with generous error bounds in order to counteract highly oscillating solution parts. This is a technique which is applied by the default algorithm in both ATHLET and THY. Accordingly, NuT supports it as well.

Concrete methods

Several methods were implemented in NuT. All are implicit schemes but can be executed in an explicit way as well, see Remark 2.8.

- `T11` and `Tvar`
  `T11` denotes the linearly implicit Euler method. The label `T11` is motivated by the close connection to the extrapolation scheme in Fig. 2.14. Practically, `T11` is the simplest linearly implicit method. Following the notation from (2.17) it can be written as

$$\begin{aligned} (-h^{-1}I + J)\delta z_1^0 &= -f(t_0, y_0) - h\partial f_0/\partial t \\ z_1^1 = \delta z_1^0, \qquad & y_1 = y_0 + z_1^1. \end{aligned} \tag{2.18}$$

  `T11` does not serve as a stand-alone method for practical use. But due to its very simple nature it helped in the development of the code.

  Of a more practical use is the extension `Tvar` which requires a parameter $\lambda \geqslant 1$. For $\lambda \in \mathbb{N}$ `Tvar` resembles $T_{\lambda,1}$, i. e., a λ-time consecutive application of (2.18)

for the step size $h_\lambda := h/\lambda$. Naturally, this leads to a linearly implicit method with $s = \lambda$ stages. The relations in (2.17) become

$$\begin{aligned}(-h_\lambda^{-1} I + J)\delta z_i^0 &= -f\Big(t_0 + (i-1)\cdot h_\lambda, y_0 + z_{i-1}^1\Big) - h_\lambda \partial f_0/\partial t \\ z_i^1 &= z_{i-1}^1 + \delta z_i^0, \quad i = 1, \ldots, \lambda, \qquad T_{\lambda,1} := y_1 = y_0 + z_\lambda^1.\end{aligned} \tag{2.19}$$

Due to its flexibility `Tvar` is a good candidate to be used as a pre method. Especially in the context of FiterRK methods of high stage order where $\mathcal{A}$ has a single point spectrum with real eigenvalue γ. The parameter λ should be chosen as $\lambda = \gamma^{-1}$ in that case. This way, `Tvar` and the main method use the same matrix for the linear systems. An additional decomposition can be avoided. The method `block1` below comes with the option to employ `Tvar` in the described way.

The eigenvalue γ is not necessarily the reciprocal of a natural number. The choice $\lambda = \gamma^{-1}$ remains the same but s becomes $s = \mathrm{ceil}(\lambda)$. This leads to a $c_s > 1$. Therefore, the developer is given the option to compute less than s stages. It is the developer's responsibility to tune the methods in a way that less than s stages still suffices to produce all desired initial guesses for the main method via NuT's cross mapping feature, see Fig. 2.15.

- `T33`

  The basic extrapolation ansatz in ATHLET is resembled by `T33`. This method uses (2.19) for $\lambda_i = i,\ i = 1, 2, 3$, generating the corresponding $T_{\lambda_i,1}$ resulting in six stages. Approximations of higher order are calculated according to the recursion

$$T_{j,k+1} = T_{j,k} + \frac{T_{j,k} - T_{j-1,k}}{(\lambda_j/\lambda_{j-k}) - 1}, \quad j > k,\ k \geqslant 1, \tag{2.20}$$

  and as depicted in Fig. 2.14. For any fixed j and k the resulting $T_{j,k+1}$ can be expressed as a linear combination of the $z_{\lambda_i}^1$ values. This is done for $T_{3,3}$ which is used to proceed the time integration. For an error estimate either $T_{2,2}$ or $T_{3,2}$ is employed. The first one is in line with the basic scheme in THY, whereas the second one relates to the ATHLET scheme. Adaptive order is not covered by this method. Extrapolation with adaptive order would require additional development of the ODE mechanism in NuT.

  An extension of `T33`, i. e., `T33extra` was created to cover special treatment of discontinuities in line with the default approach in ATHLET and THY. Details are covered in Section 2.3.3.

- `block1`

  The method `block1` is an FiterRK method of stage order two which is realized by

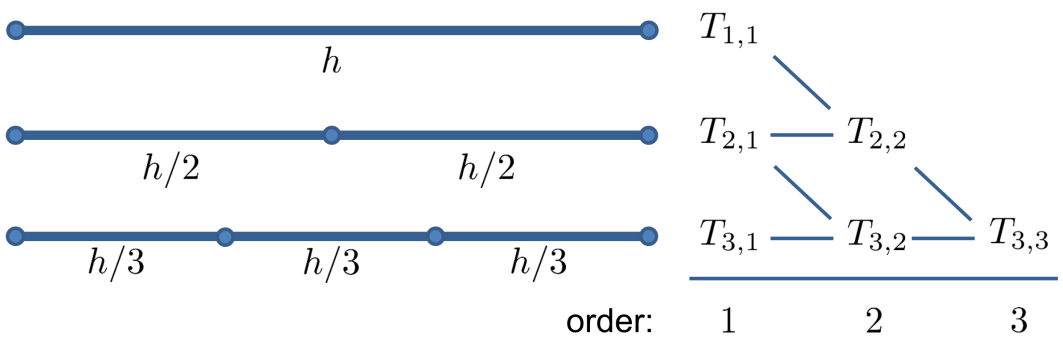

Fig. 2.14 Extrapolation scheme based on the linearly implicit Euler up to order three. The $T_{j,k}$ are calculated according to (2.20).

a block of size three using the techniques from /STE 17a/ to ensure a single point spectrum with eigenvalue $\gamma = \frac{1}{2}$. It comes with an additional diagonal stage of order three for the sake of error estimation and local extrapolation. Also, an additional diagonal stage of order one is provided. Optionally, `Tvar` can be employed with $\lambda = \gamma^{-1} = 2$ to deliver initial guesses of order one for the block stages. The main purpose of `block1` is to test the implementation for block stages and pre methods. Further, more application related, FiterRK methods can be added later.

Computing initial guesses z_i^0

The computations in (2.17) require an initial guess z_i^0. Analogously, this holds true for potential block stages. The implementation of ODE methods in NuT allows for a variety of options how to provide initial guesses. The main motivation is to avoid additional f-evaluations wherever possible. Fig. 2.15 gives an overview of the different options. What option(s) are in use is decided by the developer of a method. It is a static information that is determined while defining the method.

- The inbox concept

  Initial data for a certain stage or a set of block stages may be available or suitable to compute *before* the actual stage(s) are considered. Hence, an inbox concept was implemented. Previous stages or an optional pre method may store the desired data in the inbox. These data comprise the z_i^0-value as well as its related f-evaluation. When it comes to computing the actual stage(s), the inbox is checked for valid data. No other stage or other method is allowed to directly write into variables that belong to a given stage. Initial data are either picked up from the inbox or they are computed by the stage itself. Thinking of a stage as an object this encapsulation concept ensures data safety, avoids unnecessary inter-dependencies, and allows for an easily traceable flow of data. Most of the labels

on the inbox side of the graph in Fig. 2.15 are rather self-explanatory. Further information is given for the following two labels:

- `inter-stage`
  When a stage i triggers an f-evaluation for some input z_i^ℓ, $\ell \geqslant 0$, in order to get on with its own computations *and* these are exactly the same data a later stage j requires, inter-stage mapping to the inbox of j can be used. In the case of a pre method as source, an intermediate outbox comes into play as an interface for the actual method to access the data.
- `piggyback`
  Piggyback mapping complements inter-stage mapping. It is the same basic idea but kicks in when the initial value plus corresponding f-evaluation for a given stage are set by any *external* means. Piggyback mapping allows the same data to be used for any subsequent stages as well and, hence, avoiding an f-evaluation at exactly the same input data.

- Direct computation
  Direct computations are triggered when the inbox for a given stage (or a set of block stages) is empty. In such a case the stage has to take care of producing initial data itself. For block stages there's only the option of a zero guess as they are the first stages of a method, see (2.16). A diagonal stage i may exploit information from previous stages. Following the basic relation between the stage shift and stage derivative quantities as given by (2.15) this is formally done by means of a linear combination of stage derivatives $k_j^{\ell_j}$, with $\ell_j \geqslant 1$ for $j < i$, yielding $z_i^0 = \sum_{j=1}^{i-1} \alpha_{ij} k_j^{\ell_j}$. This motivates the `alpha` notation in Fig. 2.15. For the actual computations, equation (2.13) is exploited once more to express the $k_j^{\ell_j}$ by linear combinations of $z_j^{\ell_j}$, $j < i$. Hence, only a z-quantity (and an accompanying time value) need to be stored to represent the information of any given stage during the stage computations of a given method.

  The sub-options `pick` and `zero` are special cases of `alpha` in the sense that they provide shortcuts to avoid calculating a linear combination if it's clear from the definition of the method that either a single previous $z_j^{l_j}$ is opted for or that $z_i^0 = 0$ holds due to $\alpha_{ij} = 0$ for all $j = 1, \ldots, i-1$.

  Finally, the label `expl1st` is reserved for the first stage of methods that utilize an explicit first stage, hence the name. Recall from above that in this case the method works with $k_1 = f(t_0, y_0)$ instead of z_1.

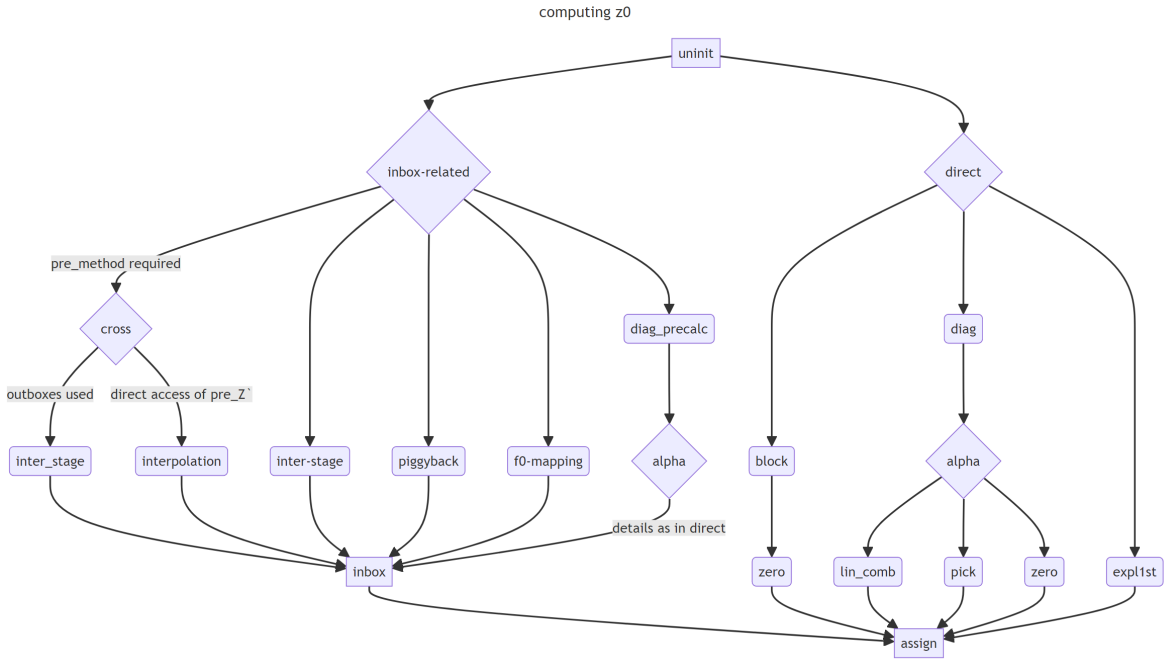

Fig. 2.15 Calculation options for z_i^0

Process flow of executing an ODE method

In contrast to NuT's linear algebra features the execution of one step of an ODE method requires interaction with the host *in between* the calculations. This is due to the demand to evaluate f for input values that are generated during the execution of a method.[4] This is not an information that can be given in advance. Hence, the implementation has to take a state mechanism into account in order to proceed computations at the correct position after f-evaluations have been processed. Furthermore, waiting for data from the host should be non-blocking.

Executing a method is handled by an ODE entity, see also Section 2.3.1.2. The above requirements have been fulfilled by the combination of two means:

- introducing a two-tuple `[section, code]` which defines the state of the execution of the method and therefore of the entity,
- providing a main routine `compute_stages` to be used by the host which triggers or continues calculations and which tells the host of the entity's status on return via the value of `code`.

It is the host's responsibility to process the status information given by NuT between two invocations of `compute_stages`. NuT provides several further routines to let the host read or write information. E. g., if the status code that is returned from

[4] Practically, the only exception is the explicit Euler method $y_1 = y_0 + hf(t_0, y_0)$.

`compute_stages` reads as `getF`, the host can use NuT's `f_get_input` routines to read the input that is required to execute an f-evaluation. Afterwards one of NuT's `set_f_output` routines can be made use of to give NuT the desired data. When `compute_stages` is invoked again, NuT assumes (after some internal checks) that the host provided the requested data. Based on `[section, code]` NuT can navigate through the ODE code to resume computations. Since this information is stored in the ODE entity, NuT is not blocked for requests during the host computations.

The general process flow in NuT is shown in Fig. 2.16. The `internal` labels refer to the possible values of `section` (except for `proceed` which is a possible `code` value). Additionally, the `tell host` labels are covered by `code`. Final approximations $\Delta y_0 = y_1 - y_0$ are computed in the `end` section. From a numerical point of view the implementation is inspired by the Algorithms 3.2-3.4 in /STE 17b/ to handle block data, diagonal data, and the determination of Δy_0, respectively.

Verification

Implementing the execution of ODE methods of the presented kind including possible block stages and sophisticated ways to produce initial guesses z_i^0 was a complex endeavor. Therefore, verification was done on two levels. Both means also serve as regression tests for future modifications.

- A significant amount of `assert`-statements were placed in the corresponding C++ code to check for valid states and data accessibility. These statements are processed only if the code is compiled in debug mode. Hence, performance does not suffer. NuT's CI takes care of running the code in debug mode as well for NuT's default CI pipeline. See also Section 3.2.3.
- The implemented methods were run for a test problem. It reads as

$$\begin{aligned} y'_{(1)} &= y_{(1)} \cdot (p_1 - p_3 y_{(2)}) \\ y'_{(2)} &= y_{(2)} \cdot (p_2 - p_4 y_{(1)}), \end{aligned} \qquad y(t_0) = \begin{pmatrix} y_{0,1} \\ y_{0,2} \end{pmatrix}, \tag{2.21}$$

  where the parameters and initial values are chosen as $p_1 = 0.08$, $p_2 = -0.2$, $p_3 = 0.002$, $p_4 = -0.0004$ and $y(t_0) = (400, 5)^T$, respectively. Above system describes a classical prey/predator interaction of Lotka-Volterra type /LOT 98/. Time integration is considered for the interval $[0, 500]$. Step sizes are fixed and chosen as $h = 0.1$. This problem has several amiable properties.
  - The problem is small in size but not scalar. Nontrivial matrix and vector operations occur while running a method for the problem. This helped tremendously to identify dimension mismatches during development.

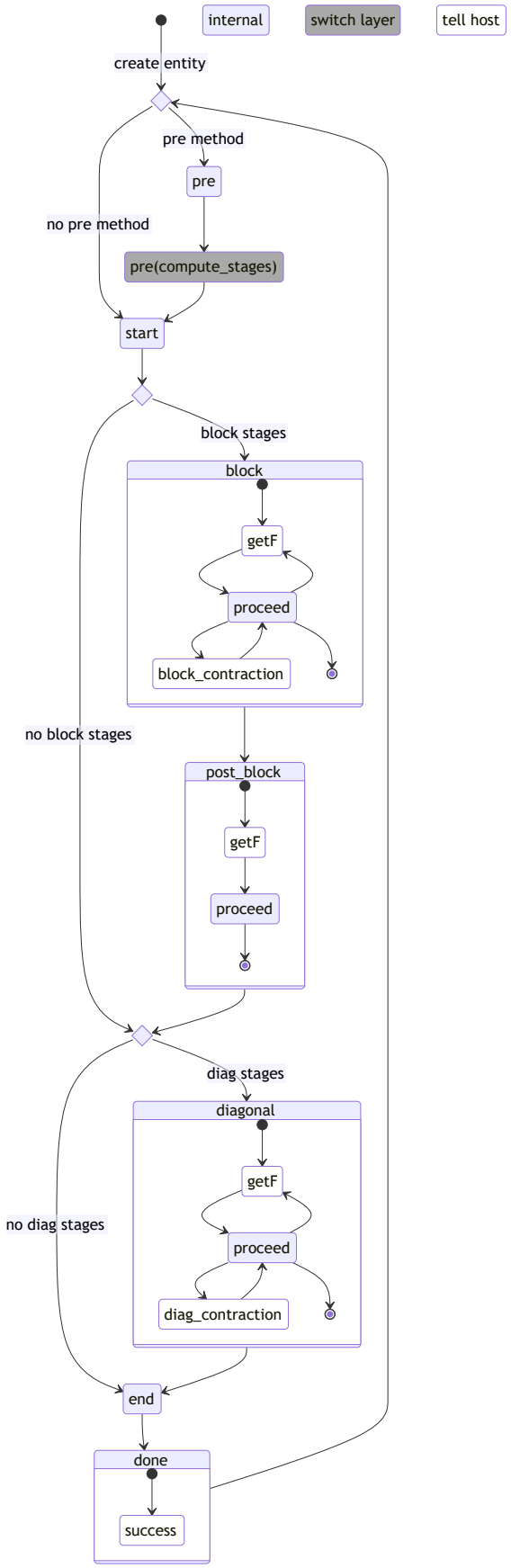

Fig. 2.16 Process flow in NuT while executing an ODE method via invocations of `compute_stages`

– The problem has a unique solution that is periodic. The periodicity has to be reproduced by a method up to a certain degree depending on the order of the method. If a method fails (but others of the same type do not) this is an indication for erroneous method parameters.

– The problem is not stiff. It is no problem to set $J = 0$ and check for the correct execution of explicit type computations.

After running the methods the resulting trajectories were manually checked. If considered of sufficient quality the final approximation values for $y(500)$ were stored in the test program to be compared with future computed values. This makes sense since such values are highly sensitive to any perturbations of the time integration process. Any modifications that are not supposed to alter the behavior of a certain method should leave the final approximations unaltered too. This testing concept came in handy while the handling of block stages was implemented. Diagonal stages were already implemented at that time. Thus, the behavior of a method that is solely based on diagonal stages should not change at all.

2.3.1.2 Accessing ODE features via NuT's interface

The ODE features described in the previous section are encapsulated in an ODE entity data structure (of the C++ type `struct`). Each ODE entity serves the execution of one method. The entity can be accessed without any further means if NuT is linked directly to a given C++ program. However, in order to make the execution of ODE methods available in the context of AC^2 a wrapper class `TS` was developed.

Like it was done for other NuT entity structs or classes before, the generator concept that was introduced in /STE 20/ was employed to create Fortran, C, and C++ interfaces for the `TS` class based on a json-input. This includes library and plugin use. Hence, the MPI communication layer is already covered as well. The generator automatism helps tremendously to ensure robust and consistent interfaces.

The entity class `TS` covers access to ODE entities as well as linear algebra entities. This helps to keep the number of required entity references on the host sides (ATHLET and THY) small. There is no need for bloated if/else constructs when one reference can be used for ODE related task or maybe only for linear algebra related ones.

The wrapping `TS` provides is often simple, forwarding the input parameters to the invocation of the corresponding ODE or linear algebra procedure, respectively. To comply to the data types that are supported by the automatically generated interfaces

certain data processing may be included too, though.

Another benefit of the TS wrapper is the handling of combining an ODE entity with a linear algebra entity. This is done in a transparent way. Per se, ODE entities don't require access to sophisticated linear algebra. For example, if a method is explicit or if it is executed with a Jacobian approximation $J = 0$ the ODE entity's solve operation simply performs some scaling. An overview of the NuT internal relationship between the `TS` wrapper and the ODE and linear algebra classes are given in Fig. 2.17.

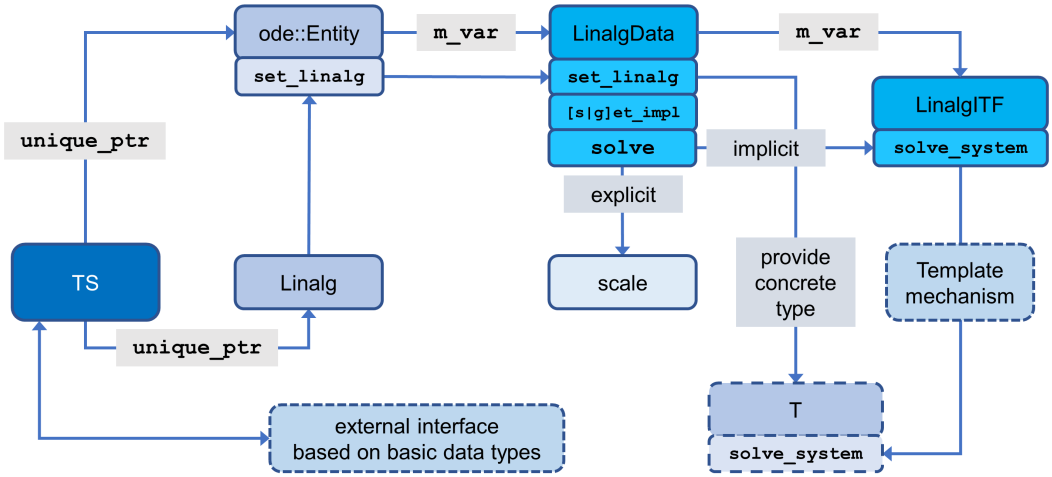

Fig. 2.17 Overview of the relationship between the `TS` wrapper and the ODE and linear algebra classes in NuT. The label `m_var` identifies the object which is pointed to as a member variable of the higher-level object.

2.3.2 Building a Jacobian matrix for the overall system – implementation

If NuT is activated, both ATHLET and THY handle the determination of Jacobian information by invoking their respective version of the procedure `FMANUT`. In case of ATHLET, an approximation of $\partial f/\partial y$ is computed, whereas for THY the derivative $\partial g/\partial u$ is approximated. In terms of an overall system, these are the matrices A and T in Fig. 2.3. For Jacobian information w. r. t. the overall system also the submatrices UR and LL, i. e. approximations of $\partial f/\partial u$ and $\partial g/\partial y$ are required. Hence, the `FMANUT` routines have been extended by synchronization means and additional code to handle the calculation of the relevant elements of UR and LL as well.

Since the cross dependencies of the solution variables require the evaluation of the individual time derivatives in parallel, see Section 2.2.1, the evaluation of f via `AFK` and that of g via `FKTFE` must always be done at the same time. This also holds true in the context of Jacobian calculations. The necessity to recalculate the Jacobian

matrix is flagged by each code individually based on the respective control logic of the time integration process. Examples for triggering a recalculation are missed error bounds or an activation or deactivation of equations. The flags used in each code are synchronized, to make sure that both codes enter their respective `FMANUT` procedure when at least one of them requires an update of the Jacobian matrix.

The principal sequence of operations and the exchange of information between the three participants is given in Fig. 2.18. In general, the determination of the Jacobian matrix consists of two main parts.

The first main part is the setup phase in which local pattern information about the equation system is evaluated and transferred to NuT, i.e., the `nut_worker` process (steps 1 to 8). In case of an overall system, this step includes updating the structure of the submatrices UR and LL. Based on the new matrix information, the seed matrix is determined by NuT and sent to the requesting code (step 9). The sparse matrix format CSR (Compressed Sparse Row) is used.

The second main part is the processing of the seed matrix which contains the actual calculation of the elements of the Jacobian matrix. Inside the loop `[seed vector]`, elements of the seed are processed by both codes in parallel. In each cycle, solution variables are perturbed, and the resulting time derivatives are calculated. The Jacobian matrix entries are computed with this information via finite differences for affected equations and transferred to NuT in steps 21 and 22 in each cycle.

The individual steps in each phase that are required to obtain the elements of the Jacobian matrix in the monolithic approach will be described in the following paragraphs in detail.

Setup phase

After evaluating its individual pattern information, each code informs the `nut_worker` in steps 1 and 2 about the updated shape of the purely local Jacobian information given by A or T, respectively. This step would also occur when the codes use NuT individually. The transfer of the submatrices A and T utilizes the extended NuT functionality as described in Section 2.3.2.1. This also applies to the later transfer of LL and UR.

Both codes then enter their respective procedure `build_matrix_structure()` to determine the submatrices LL and UR. In case of ATHLET this procedure is one of the procedures imported from the ATHLET/CD-driver, see also Section 2.3.3.3. It first receives information about the current structure of the matrix T from THY. Combined

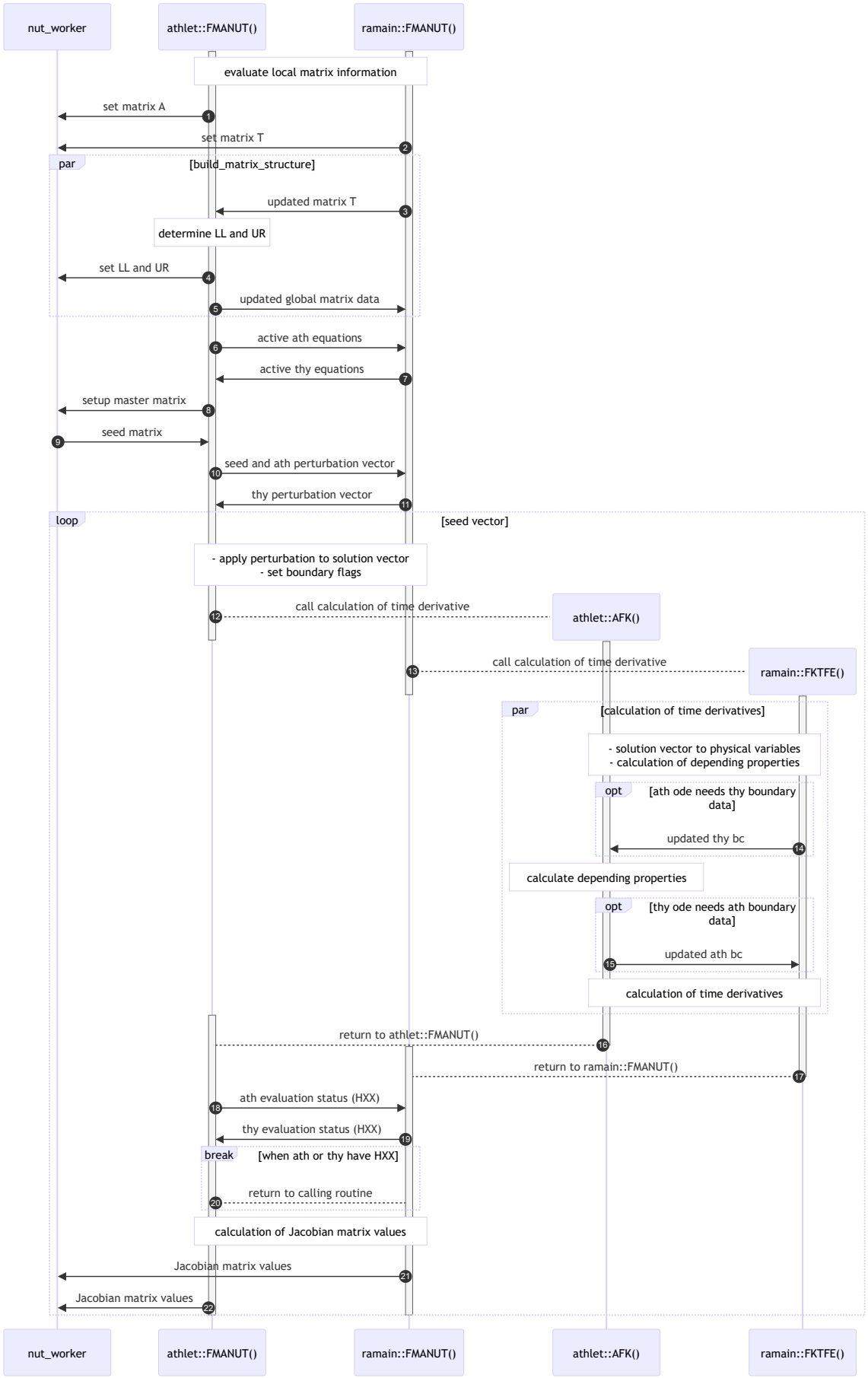

Fig. 2.18 Overview of the Jacobian matrix calculation process

with data about the structure of the submatrix A that is directly read from ATHLET, the new structure of the LL matrix can be determined. The structure of the submatrix UR then follows from symmetry assumptions as described in Section 2.2.1.

This code section also includes the generation of equation-wise vectors, that store information about the (mutual) dependence of ATHLET and THY equations. These vectors are used in the later calls to `AFK` and `FKTFE` to determine whether it is necessary to update boundary conditions.The updated structure of LL and UR is transferred to the `nut_worker` process in step 4. The updated information about the global matrix is shared with THY in step 5 along with the dependency vectors. The inclusion of the latter ensures that send/receive operations for boundary data are triggered for the same equations during the seed application.

To finalize the setup of the global matrix, NuT requires the information which equations are activated. This information is collected by steps 6 and 7 on the ATHLET side and transferred to NuT in step 8. This finishes the preparation of the global matrix inside NuT, and in step 9 ATHLET requests and receives the seed matrix. In step 10, the seed matrix is shared with THY along with the perturbation vector of ATHLET. The latter is needed in THY to apply the finite difference scheme to ATHLET perturbations (i. e. Δp_j in eq. (2.22) below). The THY perturbations are needed in ATHLET for the same reason and thus received by ATHLET in step 11.

Calculation phase

With the setup phase finished, both codes enter the loop `[seed vector]`, in which one column of the seed matrix S is processed in each loop cycle. The non-zero elements of the current column define a set of equation numbers $\mathbb{S}$ from the global equation system that are to be perturbed during this cycle. Both codes process all elements of $\mathbb{S}$. For each element that belongs to the code itself, the corresponding value from the perturbation vector is added to the solution variable. This leads to the perturbed solution vectors $\tilde{u}$ and $\tilde{y}$ in THY and ATHLET, respectively. The flags that will trigger exchange of boundary data later in `AFK` and `FKTFE` are set according to the dependency vectors.

The evaluation of the time derivatives is then started in parallel in steps 12 and 13. ATHLET enters `AFK`, while THY calls `FKTFE`. Both codes begin their evaluation by transferring the values from their solution vector to physical variables. In case of THY for instance, the solution vector variables associated with the component masses in the zones are copied to the matching `ZMASS(icomp,ipart,izone)` to which later evaluations inside `FKTFE` refer. This is followed by the calculation of additional depending

variables that are not part of the solution vector, e. g., the calculation of the pressure in gas zone parts as function of the respective ZMASS values in THY .

Hence, all possible values $\alpha(\tilde{u})$ from equations (2.3) and (2.4) are determined. When at least one element of $\mathbb{S}$ has made the respective boundary flag to be set, the transfer of boundary data from THY to ATHLET is triggered in step 14. This usually means that a THY solution variable was perturbed that affects a coupling variable, e. g., the component mass or temperature of a coupled zone. On the ATHLET side the received values are applied to the appropriate coupling interface, and depending variables are calculated, e. g., component mass flow rates, that depend on the solution variable *total mass flow rate* (or *volume flow rate*), and the gas mixture in THY should the flow direction point from THY to ATHLET. With this, all possible $\beta(\tilde{y}, \alpha(\tilde{u}))$ are determined. In this scenario, the boundary values β from eq. (2.3), for instance the component mass flow rates, are transferred to THY in step 15. There, they can be used in the further calculation of the time derivatives.

Once all time derivatives are calculated, both processes return to their respective FMANUT procedure. In case of problems during the calculation of the time derivatives both AFK and FKTFE set their respective HXX flag to true. These values are exchanged between the codes in steps 18 and 19, and logically combined with the OR operation. This ensures that both routines return the same HXX state to the controller which has to initiate appropriate actions when HXX = true is returned.

When the evaluation of the time derivatives was successful, both codes compute approximations J_{ij} to the elements of the Jacobian matrix by means of finite differences. For this purpose, both codes process all elements of $\mathbb{S}$ again to determine the required values.

$$\begin{aligned} &\text{ATH: } J_{ij} = \frac{f_i(t, \tilde{y}, \alpha(\tilde{u})) - f_i(t, y_0, \alpha(u_0))}{\Delta p_j} \\ &\text{THY: } J_{ij} = \frac{g_i\Big(t, \tilde{y}, \beta(\tilde{y}, \alpha(\tilde{u}))\Big) - g_i(t, y_0, \beta(y_0, \alpha(u_0))}{\Delta p_j} \end{aligned} \qquad \text{with } j \in \mathbb{S}. \tag{2.22}$$

In the above y_0 and u_0 denote the unperturbed solution vector of the respective code. The vectors $\tilde{y}$ and $\tilde{u}$ are the perturbed solution vectors, that take the perturbation Δp_j into account. The value of the latter depends on the physical meaning of equation j and is maintained by each code. Since these values were exchanged during the setup phase (steps 10 and 11), both codes can select the value of Δp_j depending on which system the equation j belongs to.

The elements of $\mathbb{S}$ are processed in a loop, processing one j in each cycle. The

relevant i, for which eq. (2.22) is to be evaluated, is determined by means of j and the pattern information of the submatrices A, T, UR, and LL.

Remark 2.10. Considering actual computations and for a given $\mathbb{S}$ the perturbed solution vectors $\tilde{y}$ and $\tilde{u}$ take all corresponding perturbations at once into account. This is a valid approach since the algorithm that generates the seed matrix ensures that each equation i is only affected by exactly one j from $\mathbb{S}$. Thus the obtained change in the time derivative for element i can be attributed to the perturbation of solution variable j with certainty. Hence, the number of required function evaluations, and therefore, the number of invocations of `AFK` and `FKTFE` can be reduced.

The codes collect their calculated matrix entries J_{ij} and transmit them to NuT for every set $\mathbb{S}$ (steps 21 and 22). This limits the size of the vectors that are required to store the values plus column and row indices of the calculated entries, since the maximum size of these vectors is the total number of equations belonging to ATHLET and THY, respectively. This can be deduced from Remark 2.10. The actual transfer contains the values plus column and row indices whic are collected for the current $\mathbb{S}$. NuT checks whether the column and row indices of the submitted entries are in line with the matrix definitions done during the setup phase. This helped a lot to identify index issues during development.

The loop `[seed vector]` ends when all columns of the the seed matrix S have been processed. Both `FMANUT` routines return to their caller.

2.3.2.1 NuT extensions to handle matrices composed of submatrices

In order to provide NuT with the feature to store and work with a Jacobian of an overall system of the form (2.6) two mechanisms were implemented:

- create and fill four submatrices organized by index,
- compose the submatrices in a 2×2 pattern to create an overall matrix.

With an overall Jacobian at hand, a monolithic approach to handle a coupled system can make use of the ODE concepts and routines in Section 2.3.1 exactly the same way as a single system would do. Furthermore, the implementation is done in a way that the usual linear algebra support for the host codes is still available. Existing interface routines haven't changed but were complemented by new ones to handle the requirements of a unified system.

The overall Jacobian holds information from both involved systems as well as information of the mutual impact. Hence, the NuT entity that manages the access to the

Jacobian matrix in NuT must be available on both communicators, the one shared with ATHLET and the one shared with COCOSYS. NuT provides such support by default. It is only required that the MMA subcommunicators of both communicators cover the *exact same* NuTprocesses. This is the default anyway if NuT support for both codes is considered. No additional settings are required.

As it turned out during the feasibility studies of Section 2.2.1, the structural data to build the UR submatrix (upper right one, see Fig. 2.3) is given by the hosts in a way that a column-oriented representation comes natural. This justifies to provide a corresponding interface routine. Also, the row-oriented version comes in handy. Hence, the procedures

```
ts_submaster_setup_bcsc,
ts_submaster_setup_bcsr.
```

were established. The pattern of the LL submatrix (lower left one) is symmetric to the one of UR. Exploiting that symmetry plus the fact that the CSR and CSC formats (compressed sparse row|column) are symmetric to each other too, i. e. CSR$(A) =$ CSC(A^T), the calls of above routines are as follows:

```
ts_submaster_setup_bcsc(ts_entity, 1, /
  ePtrTHY, ePtrATH, colPtr, rowIndx),
ts_submaster_setup_bcsr(ts_entity, 2, /
  ePtrTHY, ePtrATH, colPtr, rowIndx).
```

The indices `1` and `2` identify the UR and LL submatrix, respectively. The arrays `colPtr` and `rowIndx` describe the nonzero *block* pattern of the matrices UR and LL in compressed form. To map from block pattern to element pattern the arrays `ePtrTHY` and `ePtrATH` are utilized. The procedures are invoked in the ATHLET/CD-driver. There, all information is available. The patterns of the single system Jacobians are build in each respective code *beforehand* via a suitable invocation of `ts_submaster_setup_bcsr` each. The pattern of the overall Jacobian matrix is build via the invocation

```
call ts_master_setup_submaster(ts_entity)
```

which is done right after building the patterns of UR and LL. There is no specific parameter beyond the related entity required since NuT's Linalg class defines a member variable `submaster` via

```
std::vector<Mat> submaster = std::vector<Mat>(4, Mat{context});
```

If a single system is considered, i. e. no coupling, `submaster` remains in its initial state. No assignments occur and no memory demands arise. Hence, performance is not affected.

The actual composition of matrices is done in NuT's `Mat` class by means of invoking the PETSc functions

```
MatCreateNest,
MatConvert
```

in a consecutive way. The first function converts an array of matrices to a matrix of PETSc type `MATNEST`. To receive a matrix of the default type `MATAIJ` the second function is used. Converting the matrix to `MATAIJ` comes with the benefit that the existing algorithms for seeding and solving can be applied as usual. For further information on the PETSc functions and types consult the online manual at /BAL 23/.

2.3.3 Adapt control logic in ATHLET and THY

Background

In ATHLET as well as in THY the Fortran routine `FEBE` is responsible for executing one step of the time integration process a simulation run is based upon. `FEBE` is invoked in a consecutive manner to get from some t_{start} to t_{end} via a finite sequence of discrete time steps $\{h_i\}$. `FEBE` implements an extrapolation scheme based on the `F`orward `E`uler / `B`ackward `E`uler methods. Backward Euler is done in its linearly implict form (2.18). Forward Euler simply refers to the explicit counterpart. Technically, there's the option to split the system in two parts where one is treated explicitly and the second one linearly implicit. This is a mode, however, that hasn't seen use in years, presumably due to performance issues. This wouldn't be a surprise since stiffness usually evolves in sub-spaces that are not along the dimensions of Euclidean space, see /HAI 96, Sec. IV.10/.

The default scheme is implicit in order to tackle the stiffness of the underlying problem. Explicit calculations come into play when discontinuities are encountered or when the Jacobian matrix cannot be determined by finite differences or when the implicit algorithm predicts too small time steps where model problems are encountered. The flag `HXX` must be set for one of these exceptions to occur.

While THY-`FEBE` is still rather close to the description of the algorithm in /BAR 89/, ATHLET-`FEBE` deviates from it in some areas. Especially due to extensions like contraction checks for the Jacobian, support of partially updating it, or special treatment

of the steam and gas quality, which is a solution variable in ATHLET. Also, in both codes certain components of the solution vectors are (optionally) handled by first order approximations (e. g. pressure in ATHLET or the mass flow rate in THY). The idea is to use the damping effect of the linearly implicit Euler method to get rid of highly oscillating parts of the solution variable.

The FEBE routine does not only include the execution of an extrapolation method but its accompanying controls for step sizes, error handling and Jacobian monitoring as well. Furthermore, FEBE is embedded in an architecture of supporting routines. Some of them purely take care of data management, others are responsible for certain sub-tasks, see Fig. 2.19. Most of the data are represented in terms of global variables that are stored in Fortran modules.

Adaption

In the given context, it was no easy task to define an alternative ODE control. FEBE's control logic and method execution are stored in the *same* Fortran file and they are tightly entangled over the length of about one thousand lines of code (THY) or two thousand lines (ATHLET), respectively. Furthermore, different coding styles, working with jump labels, anachronistic naming schemes and using global variables to define the state of the algorithm posed a considerable challenge. A very careful approach was necessary to adapt the basics in order to make the execution of another method possible by means of an alternative control logic. First, THY was taken care of due to its less complicated FEBE version. Then ATHLET followed based on the work done in THY. Further work on the logic is strongly advised to improve its overall performance and to better meet the host's requirements, see also Sections 2.3.3.2 and 2.3.4 below.

Regarding the implementation it was decided to exploit the embedding of FEBE for the newly developed alternative as well. The new control logic is stored in a separate Fortran module nut_ts. It is accompanied by several auxiliary routines to make the embedding work and to improve readability. Details on the embedding for ATHLET are shown in the lower half of the diagram in Fig. 2.19. For THY an analogous approach was followed. Still, global variables are in play, but embedding comes with the advantage that the code beyond the actual ODE logic in nut_ts can behave as usual. The scope of modifications did not expand to unrealistic levels. Considering the complexity of the matter at hand it is reasonable to pursue a step-by-step approach where future work can benefit from the basics that were established in this project.

Since NuT takes care of executing a method, including the necessary linear algebra, FIMP and FTRIX are mainly used for providing the correct data to build derivatives,

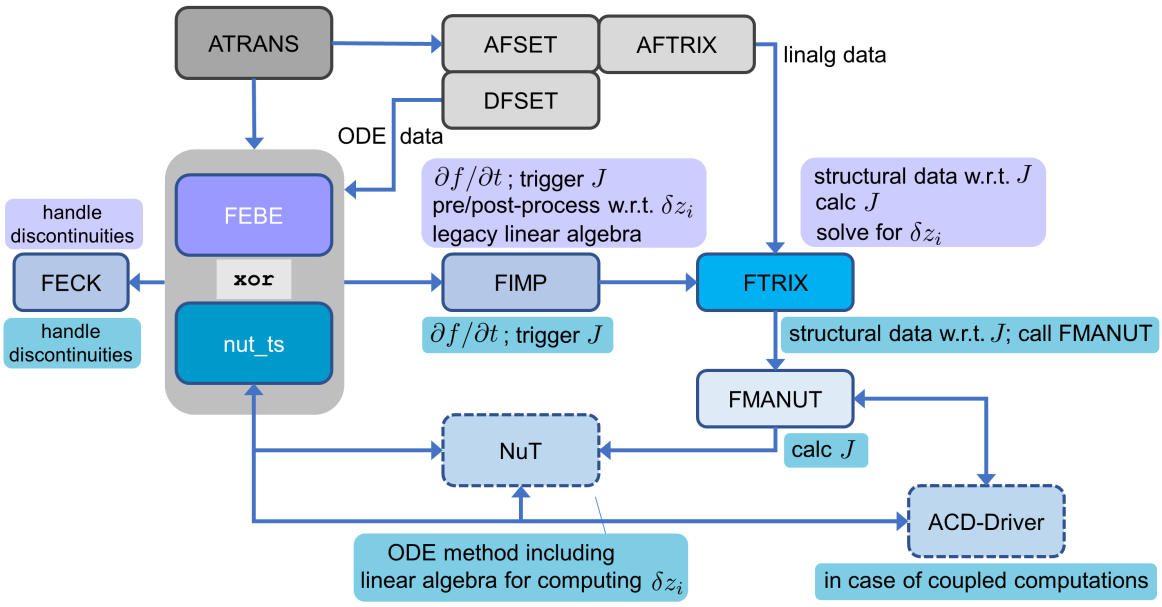

Fig. 2.19 Embedding NuT's ODE feature in ATHLET. The THY case is analogous

mainly the Jacobian which is handled by `FMANUT`. This is true for single code computations as well as coupled computations. In the latter case the `FMANUT` routines from both codes, ATHLET and THY, are run in parallel. The ATHLET/CD-driver comes into play as well, see also Section 2.3.2. Like it is done for `FEBE`, the routine `FECK` takes care of exception handling in case of discontinuities. Some minor modifications were necessary to be compatible to the new approach.

2.3.3.1 Overview of supported control aspects

The following aspects are taken care of by the alternative control logic. If no specific code is mentioned, ATHLET or THY statements about `FEBE` are related to both codes.

Execution of single code or coupled simulations

Both codes, ATHLET and COCOSYS-THY, can use NuT's ODE feature separately and also together in the context of a monolithic approach to coupled computations. For the latter case the general flow of execution is the same. Still the time step procedure of `nut_ts` is invoked. Additional synchronization measures take care of the exchange of coupling data. Further details are discussed in Section 2.3.3.3.

Explicit and (linearly) implicit methods

The control logic in `nut_ts` supports the type of methods discussed in Section 2.3.1.1. Technically, an arbitrary amount of methods can be initiated. Switching between methods is possible too. This allows for special treatments, see the paragraph on

discontinuities below. When computations are done a single method is in charge.

Error control and step size control

These two control types go hand in hand since the error has a direct impact on the step size. The implementation of both controls are inspired by the discussion in /STE 17b, Subsec. 3.5.1/ on that matter. A consistent error norm is used that takes default absolute and relative tolerances into account. Weights for the relative error can be computed based on a mix of y_0 and y_1 but also by resorting to the stage values Y_i, which is a generalization to FEBE's choice. Also, special error bounds can be considered as the host sees fit. The default base norm is the maximum norm. This comes in handy for coupled computations since each code can compute its own part of the error. The two results are compared to each other by a final $\max(\cdot, \cdot)$ invocation. This is equivalent to applying the maximum norm to a vector that is related to the overall system.

For the step size control a correction (when redoing a step) is done by the classical dead-beat controller H_0110 /HAI 96, Sec. IV.8/. FEBE's step size prediction is based on this controller as well. The NuT counterpart can offer two additional step size selections based on control theoretic considerations /GUS 94/, /SÖD 03/. See also /STE 17b, Tab. 3.18/. Additional absolute and relative limiters are applied. The values for that are taken from FEBE. Note that especially in ATHLET FEBE sometimes appears to be inconsistent in measuring the influence of the error for step size control. Further investigation is required.

Turned-off equations

In both codes the logical array TOP tracks which equations are active (`false`) and which are turned-off (`true`) for the current time step. This control is supported by the alternative control in `nut_ts` analogously to FEBE. Explicitly sharing the TOP information with NuT is only required for constructing the Jacobian matrix. For other parts of the ODE method calculations it suffices to simply set any component of the function evaluation to zero that are turned-off according to TOP. A simple induction argument shows that this is then also true for the stage shifts z_i. Hence, no change of the corresponding components appear.

Calculation of derivatives

As for FEBE this is done by the help of FIMP and FTRIX. The actual routine to calculate the Jacobian in tandem with NuT is FMANUT. See also Fig. 2.19. Derivatives are taken care of if implicit calculations are opted for (the default) and right before the

stage evaluations by means of NuT are executed. A new Jacobian is evaluated if it is demanded from outside nut_ts, if a change from explicit to implicit calculations occurs, or if the age counter exceeds its limit. Partial updates are not supported yet.

Handling of HXX-cases

It is not a rare case that function evaluations report that a HXX-event occurred. To handle the situation, the routine FECK is invoked, just like it is done for FEBE. The original idea of FECK is to solely handle discontinuities. Either a switch to explicit calculations is initiated or a bisection-based algorithm is used to reduce the step size. This kind of treatment proves beneficial for the other types of HXX-events as well, hence, the more general application of FECK is given.

Following its original intention, FECK's bisection ansatz approaches the time value a discontinuity arises at in an iterative way, since often it is not known in advance. If a certain tolerance is met, dedicated explicit calculations kick in to bridge the discontinuity. This is done like in FEBE where a first order explicit Euler scheme with additional smoothing for the last sub-step is applied. These special computations are handled by a second TS entity. After the discontinuity is bridged the logic in nu_ts switches back to the main TS entity that covers working with the actual method.

Handling HXX-events and its aftermath comes with some influence on the step size selection process. This is adapted from FEBE.

Support of forward flow evaluations in ATHLET

Function evaluations in ATHLET are not to be made in an arbitrary fashion. It matters at what time value the previous evaluation took place. The diagram in Fig. 2.20 shows the dos and don'ts. Basically, the flow of evaluations has to be forward and has to proceed in not too large sub-steps. If an evaluation for an earlier time value is required, a new flow from the start t_0 is initiated. The main motivation for this special treatment of function evaluations comes from the ATHLET-internal weak coupling between thermohydraulics and heat transfer: thermohydraulics move forward by a sub-step and heat transfer calculations catch up. Several other models, including parts of ATHLET/CD, have similar constraints due to their loose coupling.

The control logic in nu_ts generalizes FEBE's handling of the situation in accordance with the dos and don'ts mentioned above. This allows for other methods than Euler to be executed. However, the given flow constraints can pose a massive limitation on any method that is not as simple as the Euler method. In any case, it has to be taken into account when creating new methods. The rather flexible approach to initial

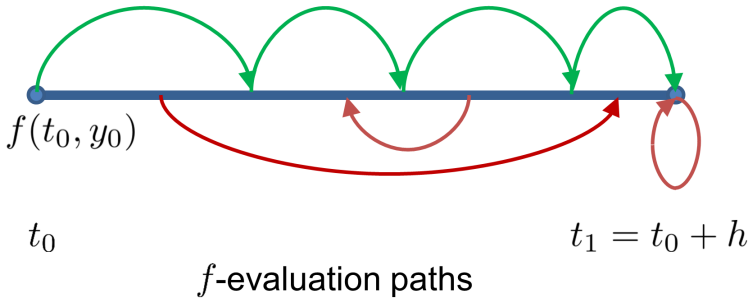

Fig. 2.20 f-evaluation paths in ATHLET. Valid choices (green) and to be avoided ones (red).

guesses z_i^0 as shown in Fig. 2.15 may help in this regard.

For the flow control the same flag `nfkey` as in `FEBE` is used to define the current evaluation state. In THY such a control logic is not required since the default mode of computations is given by a strong coupling between thermohydraulics and heat transfer. Weak coupling is possible too but it is considered a legacy mode. Hence, it is not taken into account by `nu_ts`.

Compatibility with ATHLET's monitor w. r. t. the steam and gas quality

The steam and gas quality is a solution variable in ATHLET. By definition it is a ratio with values in $[0, 1]$. Furthermore, it is not allowed to change too drastically when close to the endpoints of the interval. ATHLET uses the routines `DXMLIM` and `DXNUPO` to monitor the situation and if necessary to apply corrections. Redoing a step with a smaller step size is an option as well (the `HXX` flag is set). The two routines show a close connection to the Euler method that is applied in `FEBE`. Of importance are the latest function evaluation and its time value (to give an idea of how the quality evolves). The same variable as in `FEBE` is used for the function evaluation. However, the corresponding time value required adaption, since not necessarily any given method uses fixed sub-step sizes as it is done by `FEBE`. Accordingly, the generalization in `nut_ts` explicitly sets the time value of the latest function evaluation according to the method in use. It is to be seen if this suffices. Basic compatibility is ensured, though.

2.3.3.2 Missing features

In the given context the priority was on establishing the basics of an ODE control logic which includes support for a monolithic approach to coupled computations. This endeavor already was of considerable complexity and led to constraints on what

could be achieved in this project. The following features in FEBE weren't taken into account and may be the content of future tasks.

- Contraction check of the Newton-type process
- Partial updates of the Jacobian, both for single code and coupled computations
- Ensuring consistency for restarts
- Explicit mass correction in ATHLET
- Adaptive order control in THY

2.3.3.3 Add synchronization means in order to handle an overall system

In the COCOSYS context multiple modules exist that communicate with the COCOSYS main driver. The latter controls the order in which individual modules execute their part of the computation in the current time step, collects results and plot data and distributes boundary conditions. The individual module processes communicate only with the main driver and not among each other for keeping the communication structure clear and avoiding deadlocks. To minimize the necessity to handle specifics of the library used for inter-process communication (currently Intel® MPI) within the main codes, communication is done via functions from the `coco_system` library. It provides the means to setup communication via MMA in the beginning of the calculation and provides functions with minimal interfaces to send, receive and unpack messages. The receiving functions are specially crafted to allow for a serial execution of COCOSYS modules. Due to the fact that the typically used MPI-library function `MPI_Receive` polls actively for the arrival of a message, a process that has called this function will require 100% CPU time until the message arrives. A typical COCOSYS run contains multiple processes that are in this state for a considerable amount of time until their next task is provided by the main driver. Active polling would make all these processes stress the CPU while they are not calculating. The receiving functions inside the `coco_system` library avoid this by regularly calling `MPI_Iprobe` instead. If no message has arrived yet, the processes sleep for small time period and check again. While sleeping, the processes appear to be idle, i.e., a process will not require much CPU time while waiting for a message.

This existing communication functionality is not well suited for coupled numeric. While in a COCOSYS run exchanges occur typically once per time step, the coupled numeric requires a large amount of exchanges per time step. An exchange of messages

between THY and ATHLET via the main driver would introduce an unwanted overhead. Thus a new set of communication functions was implemented that allow direct communication between THY and ATHLET. The only additional parameter these functions require – when compared to the already existing communication functions – is of a Fortran derived type named `t_interface`. The parameter contains all the information required for the communication and makes the new functions usable for future other module to module communications. A predefined instance of this type can be accessed by both THY and ATHLET/CD-driver via the `coco_system` library.

While the `MPI_Iprobe` mechanism reduces the CPU load during serial execution of processes, THY and ATHLET will work on the overall numerics in parallel. When a process enters a message receive function, and the expected message has not yet arrived, the `MPI_Iprobe` cycle will enter the sleep state at least once. Entering the sleep state for numerous messages in a time step would thus introduce a considerable time delay, especially under Microsoft® Windows, where the minimal sleep time is 1000 µs (the default sleep time under Linux is set to 50 µs). The new set of communication functions therefore uses the `MPI_Receive` and `MPI_Send` functions directly to avoid any sleep cycles and process messages as fast as possible.

The ATHLET library itself is not aware of any communication routines. In case of using NuT these are provided by the NuT plugin. In case of coupling with COCOSYS, ATHLET calls external procedures from the ATHLET/CD-driver, which are linked against the `coco_system` library and thus have access to communication functions. To be able to call external procedures, ATHLET calls `_callHook("<hook-name>")` at specific code locations. External codes can add one of their procedures to the hook `<hook-name>` by calling `connectCallback(...,"<hook-name>",...)`. In most cases, this methodology is used by the ATHLET/CD-driver, for instance to receive new boundary conditions from COCOSYS at the begin of a time step via the hook `ATRANS_NewTimeStep` or, at the end of a time step, to send ATHLET results to COCOSYS via the hook `ATRANS_ExtDataDone`. The external procedures access ATHLET data by importing pointers from scopes that are exposed by ATHLET.

However, this methodology has certain drawbacks:

- For the coupled numeric numerous exchanges are required, requiring numerous additional hooks.
- Parameters can not be passed to a hook procedure in a simple manner, making the reuse of functions difficult.

- All ATHLET data accessed by a hook procedure have to be exported via a scope, making it ambiguous for the developer which values are involved or changed by the procedure.
- While the developers do see hooks in the main code, it is not immediately clear which procedure is attached to the hook or whether there is a procedure attached at all.

Thus, it was decided, not to follow this methodology for the coupled numerics. Instead, the ATHLET/CD-driver exports a set of procedure pointers with well-defined interfaces to the ATHLET scope `nut_jac` once at the beginning of the simulation. ATHLET imports these pointers and is then able to directly call the procedures while passing data unknown to the ATHLET/CD-driver. For instance, the procedure `sync_real8(r8, operation)` is one of the procedures imported by ATHLET. The procedure parameters are a floating point number and an operation specifier. The procedure is used in different contexts during the joint time step integration. It is used for instance before a check whether the current estimated error in the global solution is acceptable by providing the maximum of the error norms estimated in each code:

```
err_norm_ac2 = err_est_norm
if (nut_mode == nut_ode_coupled) then
   call sync_real8(err_norm_ac2, 'max') ! get global error norm
endif
```

Additionally, it is used in multiple places in the code after new step sizes were calculated by the individual codes:

```
if (nut_mode == nut_ode_coupled) then
   call sync_real8(step_size_new, 'min') ! get minimal time step size
endif
```

Other exported procedures are only called once. They do a specific task at their call point, usually exchanging multiple data in one message to reduce the number of messages. The procedure `exchange_pre_jac_data(...)` for instance is called when both codes have finished their preparation for the calculation of the Jacobian matrix and the global seed matrix has been obtained from NuT by ATHLET. From the ATHLET side the perturbation vector and the seed matrix is sent to THY, while THY sends the perturbation vector of his equations to ATHLET.

2.3.4 Running a test case

Description

To test the new ODE control logic with the overall equation system, the sample setup *Simple Sample* was run. For a detailed description of the sample refer to section 2.2.3.1 and Fig. 2.6. The results from the coupled run using the overall equation system and the ODE solver from NuT, applying the method `T33`, are compared to runs using the traditional coupling method with two independent equation systems and using the `FEBE/FTRIX` package, as well as stand-alone ATHLET calculations applying the NuT solution methods or the `FEBE/FTRIX` package.

ATHLET's external coupling interface is only activated when an external code, here the ATHLET/CD-driver, sets the necessary flags before ATHLET starts. Input concerning the external coupling interface is otherwise ignored, and ATHLET dynamically calculates the respective TFOs by itself. This allows us to run the *Simple Sample* also uncoupled, i. e., using only ATHLET, for comparison while using the same ATHLET input file. Since the activation of NuT features can also be achieved with command line flags, that same ATHLET input file was used for the application of NuT and for the run using the traditional `FEBE/FTRIX` package.

When using the traditional coupling method, ATHLET calculates a variable number of time steps first, then COCOSYS follows and calculates the same time interval. The maximum number of time steps ATHLET is allowed to do is controlled via the input. There are means in place that limit the number of steps ATHLET does, which are based on the amount of mass or energy transferred. The values of the latter are integrated by the ATHLET/CD-driver for that purpose. For the comparison done here, the settings are such that ATHLET is allowed to do exactly one time step. This represents the closest numerical coupling achievable with the traditional coupling methodology. A time step cycle thus runs as follows. ATHLET can adjust the time step size for its step according to its own demands. COCOSYS will take over the time step size used by ATHLET for its own step. When ATHLET has finished the time step calculation, its results, here mass and energy flow rates for the junction between `PCONNECT` and `COCOZONE`, are transferred to THY, where they are used in THY calculations for this step. After COCOSYS has calculated its time step, the THY results of this step, here pressure, temperature and gas mixture, are transferred to ATHLET, where they are applied at the beginning of ATHLET's next time step.

The maximum time step size was set to 0.5 s. The injection of steam starts at a time of 5 s to linearly increase from 0 kg s^{-1} to 0.1 kg s^{-1} until 6 s. Then it remains constant.

The initial gas mixture in the domain consists of air with a steam fraction of 10 Vol−%.

Results

Some results of the four runs are depicted in Fig. 2.21 and Fig. 2.22. The curves are named according to the methodology used. The name AC2 is used for coupled simulations using the traditional coupling methodology using the FEBE/FTRIX package. AC2+NuT was run with the new ODE capabilities of NuT and the monolithic equation system. ATHLET and ATHLET+NuT represent stand-alone ATHLET calculations, where the first uses the traditional FEBE/FTRIX package, and the latter uses the new ODE capabilities of NuT.

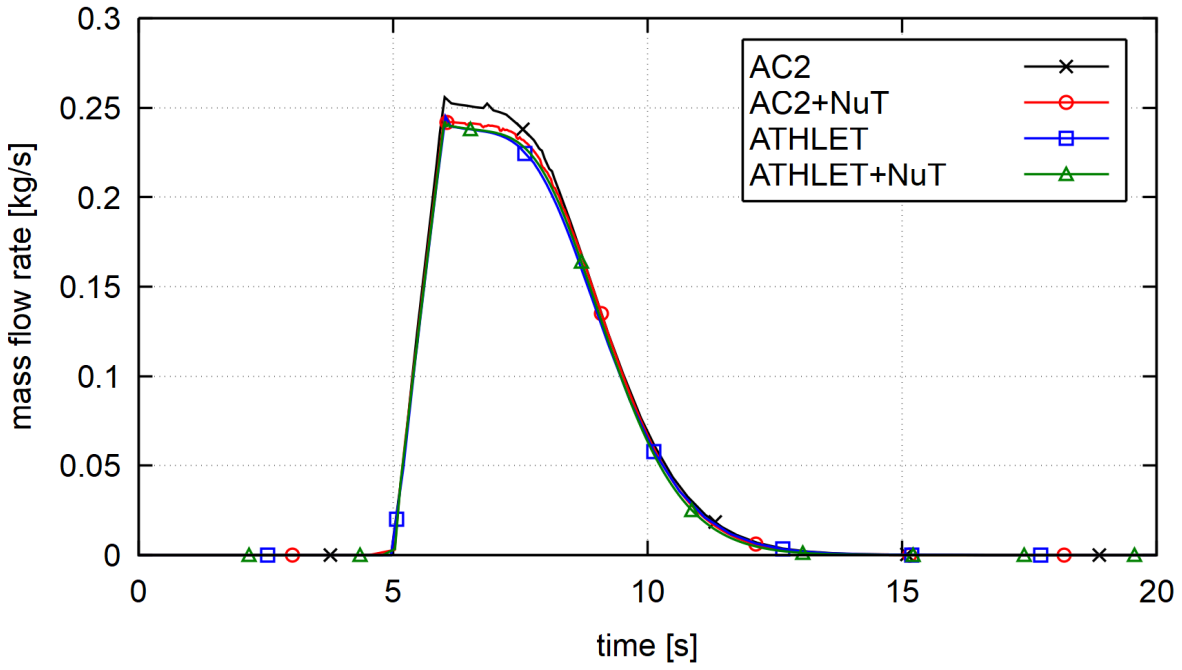

Fig. 2.21 Simple sample: Mass flow rate of non-condensable gases over time

The mass flow rate of non-condensable gases (air) depicted in Fig. 2.21 clearly shows the onset of the steam injection by a rapid rise of the mass flow rate. The mass flow rate shown belongs to the ATHLET junction that connects PCONNECT and COCOZONE. The highest mass flow rate reached is larger than the injected 0.1 kg s^{-1}, since the injected steam has a lower density than the gas mixture initially present in PIPE and PCONNECT. And because the volume of the displaced air-steam mixture corresponds to that of the injected steam, the mass flow rate is larger. As more and more air in PCONNECT is replaced by the incoming steam, the air mass flow rate decreases until it reaches about 0 kg s^{-1} eventually.

While the results for the ATHLET stand-alone calculations and the coupled AC2+NuT

agree well with one another, the run using the traditional coupling AC2 shows a higher peak value of the mass flow rate. The reason can be found in the coupling methodology applied. ATHLET predicts a rise of the mass flow rate until the pressure on the opposite side rises, which slows or even reverts the rise. Since the information about a pressure rise in the COCOSYS domain is only transferred back to ATHLET at the beginning of a new time step, ATHLET increases the mass flow rate for the full time step. With a constant pressure as boundary condition, ATHLET sees no numerical problems, and thus no reason to reduce the time step size any further. The exact peak value therefore highly depends on where the time step ends in relation to the end of the mass flow rise in this case. This behaviour illustrates the potential for numerical instability at the coupling junction, should this overshooting in time step size and physical quantities (here mass flow) lead to oscillations that cannot be controlled any more. The coupled calculation AC2+NuT is much more in line with the stand-alone ATHLET runs and shows much less overshooting, indicating better stability properties.

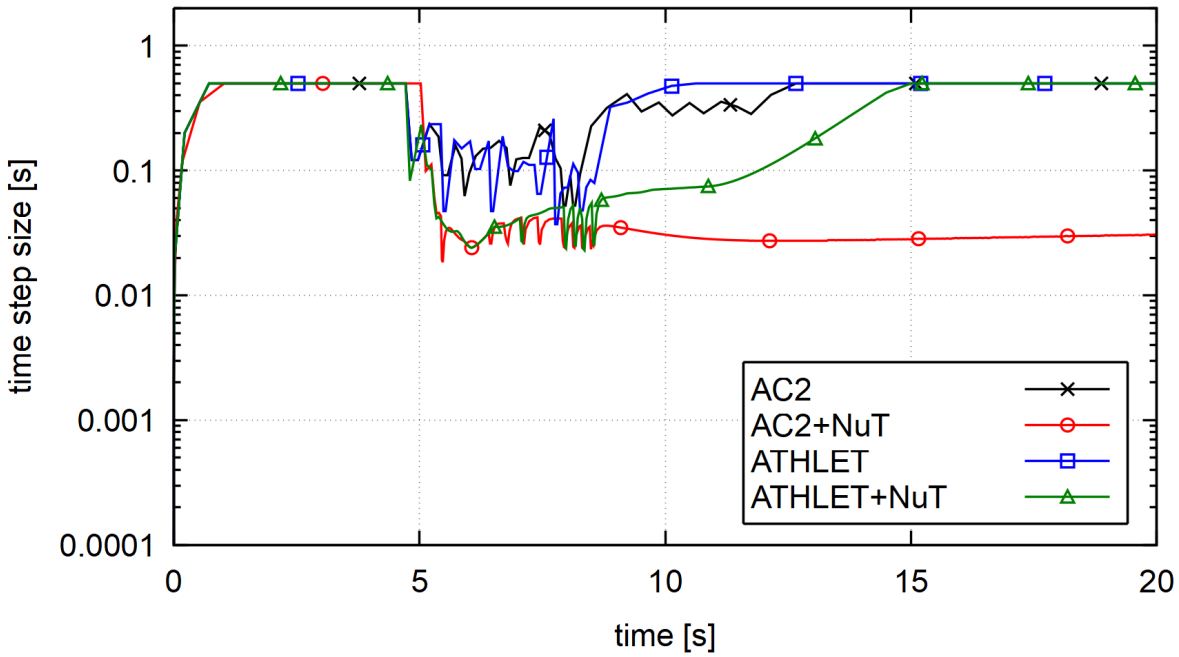

Fig. 2.22 Simple sample: Time step size over time

The numerical efficiency of the different methods can be assessed via the time step sizes of the simulations shown in Fig. 2.22. All simulations run with maximum time step size until the steam injection starts. While the runs that use the traditional FEBE/FTRIX package reduce the time step size to around 0.1 s, those runs using the NuT ODE method go down to about 0.025 s. This means that about 4 times more

time steps have to be calculated in the time interval 5 s to 9 s when NuT ODE is used. The runs that use the FEBE/FTRIX package return to using about the maximum time step size around a time of 10 s. The runs applying NuT ODE use a smaller time step size beyond that time. The ATHLET stand-alone run using NuT does only reach the maximum time step size at around 15 s. The coupled simulation using the monolithic equation system (AC2+NuT) remains at a time step size below 0.03 s for the rest of the simulation.

ATHLET reports the equation that limits the current step size of each time step in its main output file. Here both stand-alone ATHLET simulations report the partial pressure of non-condensable gases in the time interval from 10 s to 15 s. Since the state of the solution vector is also similar, the effective time step size should be similar as well. In case of AC2+NuT, the equation limiting the time step size is reported to be the volume flow rate equation of the junction connecting PCONNECT and COCOZONE for all times large than 8.5 s. And thus the very equation that connects the THY and ATHLET domains. The dependencies considered during the calculation of the Jacobian matrix were manually checked and found to be as expected. It must still be concluded that further investigations are needed to check whether the step size control inside NuT and the determination of the Jacobian matrix values work adequately and are free of bugs.

3 WP2 – Improving NuT and AC^2 on the Level of Software Engineering

This chapter comprises the results of the activities regarding the improvement of software development in NuT and AC^2. Below sections are given according to the project structure of WP2. All required tasks were taken care of successfully.

A crucial part in the success played the DevOps tool GitLab™ [1] /GIT 24b/. Combined with appropriate CMake techniques /KIT 23/ a powerful infrastructure was created which significantly improves the software development process not only for NuT but the whole of AC^2 culminating in a fully automated release pipeline. Details are given in Section 3.2. The combination of tools came in handy for the refactoring of NuT's code as well: Results were easy to verify and the handling of external numerics was elegantly solved by providing an automatism to build corresponding binary packages. The refactoring process and its resulting benefits and features are described in Section 3.1. Further improvements like automated handling of licenses or support for external developers were developed by means of the established tool set. Details are given as part of Section 3.2.6.

As planned, a corporation with the POP project /POP 23/ was initiated to assess NuT's parallel performance. The results of the assessment are discussed in Section 3.3.

3.1 Reviewing the NuT code regarding the potential for refactoring

Several different activities were considered for the task of refactoring the NuT code. Refactoring of NuT's code was done in terms of improvement of architecture, robustness and performance.

3.1.1 Software architecture

While a software project is usually continuously growing and easily getting more complex, it is important that is doesn't get unnecessarily complicated to work with. A possible way to achieve this is to divide it into independent modules, where each module has only a single purpose that developers can focus on. Furthermore, modules should be loosely coupled through high level abstract interfaces instead of direct dependencies. A change in one module should rarely lead to a change in another

[1] GITLAB is a trademark of GitLab Inc. in the United States and other countries and regions

module /MAR 03/ /HUN 00/. This supports local reasoning, which in turn accelerates further development, code reviews and helps to set up unit testing.

The above described technique was used as guidance for the refactoring that was done in NuT. Especially the architecture of the communication (COM) module, see Fig. 3.1, which implements remote method invocation (RMI) was significantly improved. Its responsibility is to provide the functionality of several high-level base classes via MPI to other remote processes. The goal is an extensible implementation where neither the communication module nor the concrete classes "know" about each other, which means that developers can add new classes for remote access without having to modify the communication module nor their own classes.

To make this work the communication module waits for a request of a host, which basically contains an identifier of an object and a respective method it wants to invoke. With that information it picks the corresponding object from its entity list, which contains all objects that are available for remote method invocation.

In the next steps NuT should transfer the method specific input parameter values, invoke the actual method and transfer back possible output parameter values. As this is all highly specific to the method's interface, the communication module needs to delegate those steps through an abstract 'execute' method, which needs to be provided by the actual object.

A possible implementation would be that the communication module provides that abstract interface and uses it as type for its entity list. However, this has the downside that the underlying classes need to inherit from that, which is invasive and would tightly couple it to that interface. Polymorphism also requires pointer semantics, which worsens local reasoning and memory management.

A better approach is that a class can provide the 'execute' method through a pure extension. This has been realized by providing an additional concept class that hides the abstract interface and manages the polymorphic call to the concrete 'execute' method. With that it is sufficient to provide an overload of the 'execute' method outside of the actual class to make it compatible with the communication module. This concept is also referred to as open-closed principle /MEY 97/.

3.1.2 Logging

Logging is essential to document what a software actually does. Typically, it should provide functions to make it easy to write various kinds of information to the console

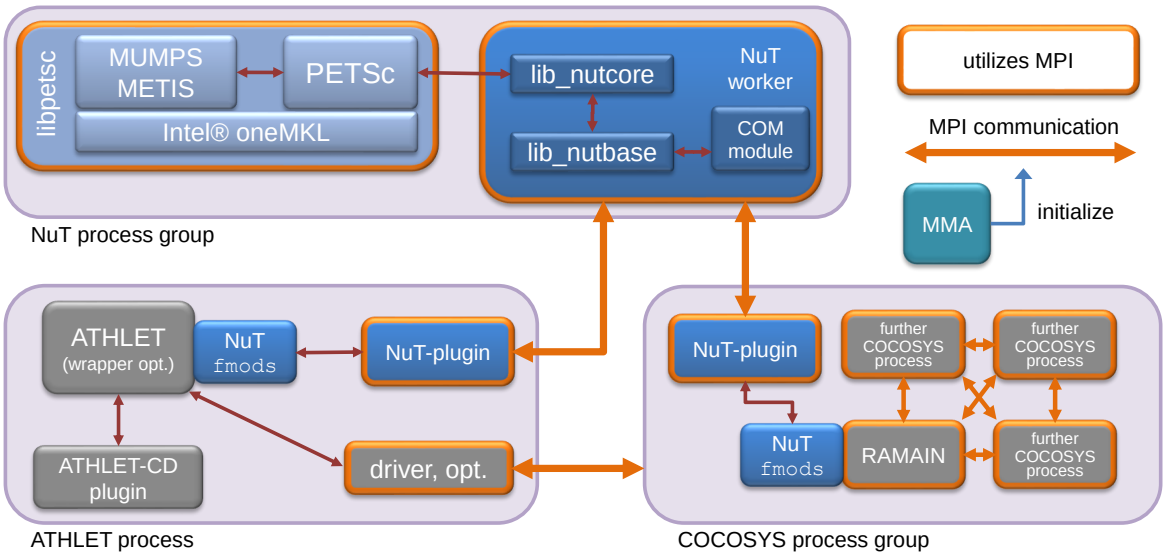

Fig. 3.1 NuT-related software architecture within AC^2

or a log file, while applying some filters like log levels or automatically add further information like timestamps.

Logging in NuT version 1.0 was done through PETSc's own logging functions as they are already MPI aware and can output extensive information about the state of native PETSc objects. Over time this approach showed some weaknesses that made it hard to scale with future developments. Its usage required that PETSc as a library can be accessed from the respective module and the state of PETSc needed to be considered, which particularly limited the scope of logging. It was not possible to log anything in the early and late stages of NuT, when PETSc is either uninitialized or already finalized. Furthermore, each single entity needed to manage everything about logging on its own. Though this brought the advantage that entities could write, for instance, into the same file without file locking or communication, it required the logging functions to be stateless. While that is quite easy to implement, log files had to be opened for each logging transaction and closed afterwards, which degrades performance due to the additional overhead of the filesystem. In addition to that, basic features like making use of various log levels or automatically adding a timestamp to the output were missing.

To fix these issues the logging mechanism has been completely redesigned in NuT. A dedicated logging class contains all basic logging functionality, without dependencies to PETSc or MPI. It provides a global hash map of logging instances that can be accessed via a unique string identifier from all parts of the code. Buffered logging via stream operator is supported. With that, multiple logging instructions can be chained,

```
// Print to console via \Cpp\ standard library
std::cout << "Object state is " << object << std::endl

// Print to console/logfile via \nut{} log library
nut::log() <<<< "Object state is " << object << nut::endl
```

Fig. 3.2 Printing via C++ standard library and NuT log library in comparison

see Fig. 3.2, which looks very tidy in the source code and is still efficient due to the underlying buffering. Furthermore, the streaming operator overload is written in a way that it also accepts output stream objects of the C++ standard library. Consequently, it accepts all objects that are printable via standard `cout` function.
Multiple logging entities with different properties can be set up. Thus, entities can have their private logging instances with individual settings or all share the same. The configurable properties include settings about the destination of the logging stream, which can be a file or the console or both. Logging commands can be supplemented with a verbosity level to differentiate their importance. The lowest verbosity level in NuT is basically for error events, the highest is used for additional debugging information for developers. File and console output can be configured for different maximum verbosity levels.

All logging entries can be provided with the event's log level or an automatically generated timestamp as additional meta information. MPI support was provided by a dedicated class as an optional extension.

To keep support of PETSc logging functions, its default output function `PetscVFPrintfDefault` was replaced by a custom function that intercepts the output and provides it as a string. This is combined with a generic wrapper function called `view` that makes sure that PETSc's default output function is only replaced during the logging event and keeps PETSc's output functions working, if it is used without NuT's wrapper class. With that, PETSc viewer functions can still be used easily with the new logger class by just calling them within the provided `view` function, see Fig. 3.3.

Fig. 3.4 shows the redirected output of PETSc's `KSPView` complemented by the auto-

```
log() << LogLevel::INFO
      << view(communicator,
         [this](PetscViewer viewer) { KSPView(solver, viewer); });
```

Fig. 3.3 Example of using PETSc viewer functions with NuT's logging library

```
[2023-10-30T16:03:20][INFO] KSP Object: 1 MPI process
[2023-10-30T16:03:20][INFO]   type: gmres
[2023-10-30T16:03:20][INFO]     restart=30, using Classical (unmodified) Gram-Schmidt Orthogonalization with no iterative refinement
[2023-10-30T16:03:20][INFO]     happy breakdown tolerance 1e-30
[2023-10-30T16:03:20][INFO]   maximum iterations=10000, nonzero initial guess
[2023-10-30T16:03:20][INFO]   using preconditioner applied to right hand side for initial guess
[2023-10-30T16:03:20][INFO]   tolerances:  relative=1e-14, absolute=1e-50, divergence=10000.
[2023-10-30T16:03:20][INFO]   left preconditioning
[2023-10-30T16:03:20][INFO]   using PRECONDITIONED norm type for convergence test
[2023-10-30T16:03:20][INFO] PC Object: 1 MPI process
[2023-10-30T16:03:20][INFO]   type: lu
[2023-10-30T16:03:20][INFO]     out-of-place factorization
[2023-10-30T16:03:20][INFO]     tolerance for zero pivot 2.22045e-14
[2023-10-30T16:03:20][INFO]     matrix ordering: qmd
[2023-10-30T16:03:20][INFO]     factor fill ratio given 5., needed 1.15789
[2023-10-30T16:03:20][INFO]       Factored matrix follows:
[2023-10-30T16:03:20][INFO]         Mat Object: 1 MPI process
[2023-10-30T16:03:20][INFO]           type: seqaij
[2023-10-30T16:03:20][INFO]           rows=6, cols=6
[2023-10-30T16:03:20][INFO]           package used to perform factorization: petsc
[2023-10-30T16:03:20][INFO]           total: nonzeros=22, allocated nonzeros=22
[2023-10-30T16:03:20][INFO]             not using I-node routines
[2023-10-30T16:03:20][INFO]   linear system matrix = precond matrix:
[2023-10-30T16:03:20][INFO]   Mat Object: 1 MPI process
[2023-10-30T16:03:20][INFO]     type: seqaij
[2023-10-30T16:03:20][INFO]     rows=6, cols=6
[2023-10-30T16:03:20][INFO]     total: nonzeros=19, allocated nonzeros=19
[2023-10-30T16:03:20][INFO]     total number of mallocs used during MatSetValues calls=0
[2023-10-30T16:03:20][INFO]       not using I-node routines
```

Fig. 3.4 Example output of PETSc's `KSPView` via NuT's logging library. Timestamp and log level on left are automatically added.

generated timestamp and log level information on the left of each line.

In addition to refactoring the logging class, the concept of log files was extended to take the newly established interaction with COCOSYS into account, see WP1. Instead of a single log file per simulation run, one log file per entity is now written. Hence, the information about the individual host interactions can be clearly presented and separated from each other. In order to be able to put the stored time data in each log file in relation to the total runtime, an entity should only be deleted in the course of the termination of the associated host application. Such a practice is not accompanied by any significant performance loss: An entity only consumes tiny CPU resources when it executes some NuT functionality. No memory issues should arise either, since the systems considered so far and anticipated can easily be handled with today's memory capabilities.

3.1.3 Maintenance

Several NuT-related maintenance tasks were carried out during the project. These include minor refinements and fixes as well as considerable improvements. The latter ones are described below.

3.1.3.1 Improved seeding and memory management

By means of NuT it is possible to calculate a seed matrix /COL 83/ for the efficient determination of a Jacobian matrix via finite differences. So far, solely PETSc's implementation of the CPR algorithm with incidence degree (ID) ordering was used. However, PETSc offers two additional suitable heuristics. These are also based on the CPR algorithm, but the ordering differs – making use of smallest last (SL) and largest first (LF), respectively.

NuT's code was extended to run all three variants and to choose the seed that has the smallest number of seed vectors. This correlates directly with the number of `AFK` evaluations needed to determine the Jacobian matrix. Tests have shown that the calculation of two additional seeds is negligible compared to the total run time. Note that while different seeds may result in different runtimes, they must provide the exact same bit values for the corresponding Jacobian matrix. Due to these properties the performance is unlikely to suffer. Quite the opposite, the performance is likely to improve. This actually happens if one of the two newly considered heuristics supplies the seed. ATHLET itself (without NuT) uses an adaptation of the CPR algorithm with SL ordering. Thus, NuT with the described extensions can never compute a worse seed for a given Jacobian matrix. From the observations so far it appears that the ATHLET-NuT tandem requires fewer `AFK` evaluations to determine the Jacobian matrices than ATHLET alone. This is directly reflected in better performance.

Furthermore, NuT's internal memory management was improved by introducing caches instead of resorting to repeated creation and destruction of temporary objects. Also, in case of sequential execution some redundant operations were discarded.

3.1.3.2 PETSc

NuT uses the PETSc /BAL 97/ library as backend which provides efficient parallel implementations of numerical algorithms. PETSc in turn requires a BLAS and LAPACK /UNI 23a/ compatible library. In the AC^2 context the additional solver package MUMPS /MUM 23/ is considered to be part of the numerical framework. This comes with a dependency on the graph partitioner METIS /KAR 23/ and on ScaLAPACK /UNI 23b/.

The compilation process of PETSc and its packages can easily take over 30 minutes, and it was challenging to provide the required UNIX-like compilation environment on Microsoft® Windows that works for all packages at the same time. Integrating that into the default AC^2 build process would be difficult to maintain, as not all development

environments are identical and even a slight difference can make the compilation of PETSc fail.

As a remedy, GitLab CI was used to prebuild the whole PETSc package for Microsoft® Windows and Linux in a dedicated PETSc Builder project. The resulting binaries are uploaded to AC²'s package registry and are automatically downloaded by NuT and integrated into its build process. Therefore, PETSc can be used for development, without having to bother about the compilation of PETSc itself.

For the current major release of AC² in 2023, PETSc was updated to version 3.19.4. Unfortunately, this led to some problems with the building process of Netlib's ScaLAPACK on Microsoft® Windows. After some testing and comparisons, it was decided to opt for Intel® oneAPI Math Kernel Library (MKL) /INT 23/ instead. The scripts in the PETSc Builder project were modified accordingly. Also, all externals are either hosted as mirrors/forks or provided as binaries on GRS's internal GitLab instance. This way the project is less dependent on the availability of external sources. Additionally, it's easy to apply small fixes or add additional means to handle things like licenses.

3.1.4 CPU affinity

The general idea of CPU affinity is to (pre)define the CPU cores a given process is allowed to be executed on. The NuT code was modified to support this concept. This proves to be beneficial under Microsoft® Windows 10 combined with certain Intel® CPUs. The motivation is to ensure that simulations including NuT are executed on so-called Performance-cores of such Intel® CPUs, even if other tasks are pursued while running simulations. Accordingly, supporting CPU affinity was done for ATHLET and COCOSYS as well (covered by other projects).

CPU affinity can easily be set for all involved AC² components at once via the environment variable `AC2_CPU_AFFINITY`. Instructions on the usage of the affinity concept in the AC² context were compiled and added to the AC² manual, see /WEY 23, Ch. 3/.

3.1.5 Refactoring NuT's documentation

In addition to the changes to the NuT code, the structural design of the documentation was also examined more closely and subjected to a refactoring process. The manual and the updates document are now based on a self-written class, which encapsulates external dependencies to other packages as well as definitions and language settings. Both German and English are supported. This makes the main document considerably

shorter and much clearer. Only a minimal number of commands related to the structure of the document are present. Most things are set by the user to fit his or her needs. Regarding the automated creation of documents, the make file has been adapted accordingly. Also, a new job was added to the CI. It can be used to create GitLab-based releases. The job is executed in addition to the build job if the commit is tagged. The artifacts are stored in the project's *Releases* tab and labeled with the tag.

3.2 Development and automation of CI processes in GitLab for NuT and AC²

3.2.1 Build techniques

Building a software from source consists of multiple stages and can easily get very complex on its own. In addition to that, there are several factors like different requirements depending on platforms, compilers, or supplementary libraries that often require some specific extra treatments that are not portable to other systems. Often this leads to the practice that native build support for only a small set of build tools like Microsoft® Visual Studio on Microsoft® Windows and `make` on Linux is established. In that case all developers are required to become acquainted with the custom requirements and workflow of the respective build process, while the project's owner has additional maintenance effort as each change in the build process needs to be transferred manually to each build tool in order to retain consistency. Another disadvantage is that more complex scenarios are practically not feasible to implement. For instance, this includes downloading additional projects or resources on demand that are not contained in the original project for technical reasons.

3.2.1.1 CMake

As solution to the above described challenges of building software in a coherent way the open-source build tool CMake was introduced /KIT 23/. CMake is a cross-platform tool, including Microsoft® Windows, Linux and macOS[2] and supports several C, C++ and Fortran compilers. The build process works in three stages as shown in Fig. 3.5. In the first stage, the configuration stage, CMake examines the development environment and makes sure that all project defined requirements are satisfied. This includes for instance the installation of a compatible compiler, python or other libraries

[2]macOS is a trademark of Apple Inc., registered in the U.S. and other countries and regions.

like MPI. If true, the second stage generates native build files for the earlier specified build tool. Following the third stage, the produced build files can be utilized directly by the respective native build tools or via CMake to build the actual project. Fig. 3.6 shows the commands to configure, generate and build a project via CMake. Consequently, the workflow of building a CMake project is highly abstract and almost identical, no matter which platform or compiler is used. /SCO 20/.

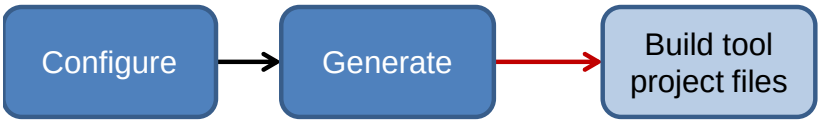

Fig. 3.5 Stages of CMake workflow

```
# Stage 1 configuration/generation of build files
cmake -S <source-folder> -B <build-folder>

# Stage 2 build project
cmake --build <build-folder>
```

Fig. 3.6 Sequence of commands to invoke the build process w.r.t. NuT as well as other AC^2 projects from source

Having the same build commands does not only help developers that are not familiar with the project but supports the automated build process discussed below.

CMake comes with several other useful features. Build files can be placed outside of the source tree. Consequently, source files and derived build files are well separated, which makes it easy to clean up a project or to have multiple independent build configurations at the same time.

Furthermore, CMake can handle a very flexible project structure. Different projects can be combined in a modular fashion, which is key to setup a dynamic large scale project hierarchy which is required by AC^2. AC^2 has a modular structure as it consists of projects like ATHLET and COCOSYS that used to be developed independently. As those codes are getting coupled tighter, new issues needed to be addressed. On one hand, a developer should be able to build all modules and their dependencies at once. Otherwise, there would be no way that the compiler can verify that the modules

have a compatible API and that shared modules are consistent. On the other hand, it is desired that developers can omit modules that are outside of the scope of their current work. For instance, it is possible to develop and use ATHLET stand-alone without loading ATHLET-CD or COCOSYS.

As with most AC^2 projects, the repository of NuT only contains the respective source files and build scripts of NuT alone. It should neither contain any binaries of itself nor of any other required library. The reason is that binaries are usually derived data, hard to track and usually increase the size of the repository by a large degree, which may make it difficult to work with it after some time. Hence, if possible, it is better to build from sources than to use prebuilt binaries.

NuT depends on multiple other projects: MMA /JAC 23a/ for communication, PETSc /BAL 23/ as numeric backend and a runtime project that provides runtime libraries to aid portability of the project.
These external projects are required to build NuT. A naive way to make them available to the build process would be to merge them directly into NuT's repository. Hence it would contain everything needed for compilation. However, that approach would be more difficult to maintain if those external projects change frequently and it can be problematic if other projects in the AC^2 scope need these dependencies as well.

As solution to these issues, CMake techniques were used that make it possible to load mandatory and additional modules on demand and dynamically combine them with the current project. These modules can be either precompiled binary packages, external libraries or other projects that in turn transitively add their own dependencies. In order to resolve those dependencies, only the first reference of an external module is considered, and each module can be loaded only once, but is available to all other modules. If the same module is referenced by multiple projects, but of a different version, then the reference stored in the main project is used.

Fig. 3.7 shows targets and their inter-dependencies that are required to build NuT. While being complex, developers only have to use the respective build commands in Fig. 3.6 and CMake resolves all dependencies itself and makes sure that all required targets are built in the right sequence with minimal effort. This approach scales very well. The AC^2 project, which collects, consolidates and builds all projects that are included in the AC^2 distribution has about 200 targets per platform with even more complex dependencies. However, due to the usage of CMake and the concepts developed within this project, it doesn't get more difficult to work with a large project than with smaller ones.

3.2.2 CI/CD

In general, continuous integration (CI) and continuous delivery (CD) are good practices to improve software quality and reduce maintenance effort. With CI, developers regularly integrate their work into the main branch opposed to long living development branches.

3.2.2.1 Concept

Fig. 3.8 shows the task automation that is realized through the GitLab CI/CD concept. For using it, projects need to provide a gitlab-ci.yml file in the root folder of their repository. That file defines the jobs needed to be run at configurable events in the respective project. Events can be manual pipeline triggers or a push of a commit that makes changes to the project's code. A collection of jobs triggered by a single event is called pipeline. A job basically consists of a script that executes a certain task and contains additional information defining dependencies between jobs and the environment where they should be executed. After triggering a new pipeline, jobs are queued until being processed. For that purpose, GitLab provides the GitLab Runner concept, which is a cross-platform service that pulls pending jobs, runs them and pushes logs and resulting artifacts back to GitLab. Hence, the runner itself doesn't store any permanent data and is practically stateless with respect to the job's data, which is an important property for scaling and availability. This allows multiple runners with identical configuration to form a pool, where pending jobs are distributed among all available runners. Removing a runner from the pool for maintenance or adding another runner to increase the computational capacity can be done with less effort.

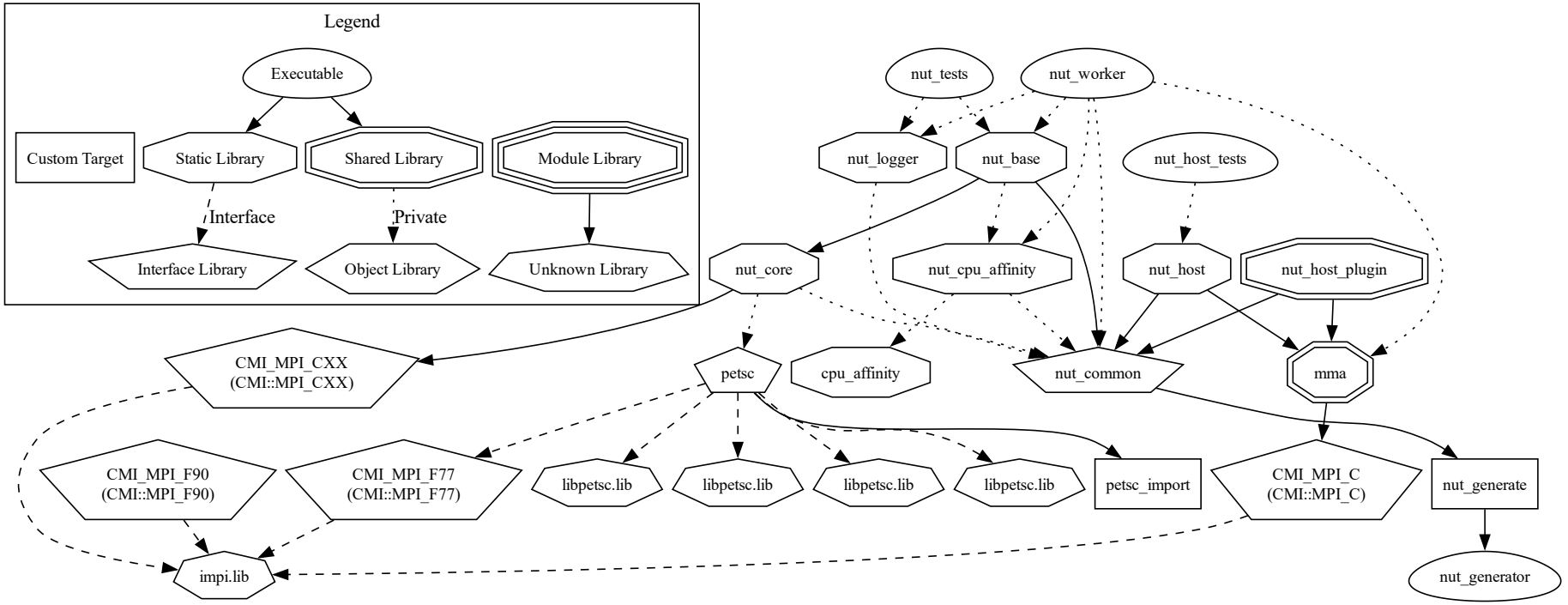

Fig. 3.7 NuT build dependencies

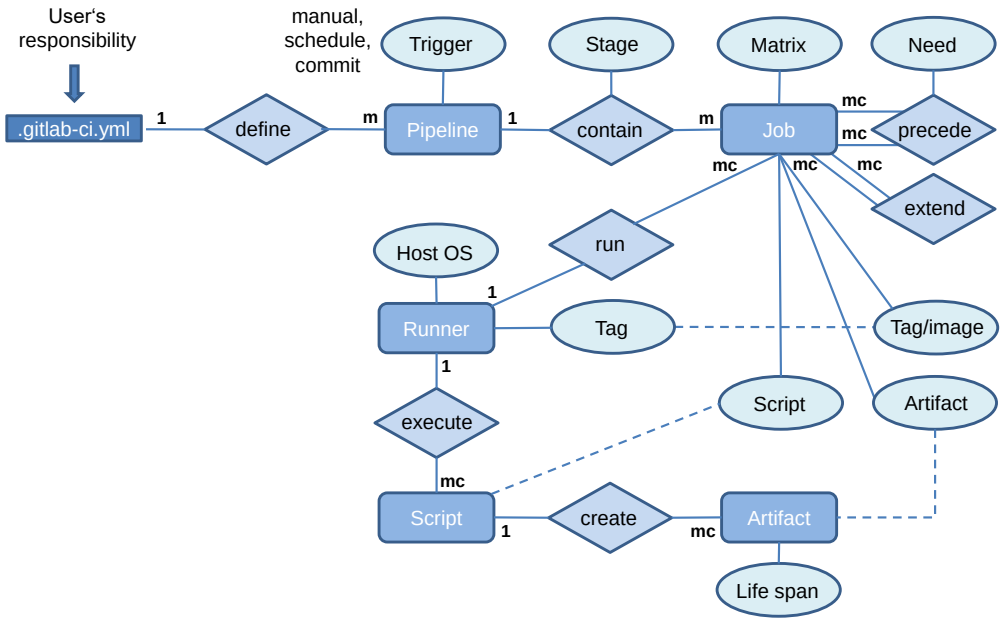

Fig. 3.8 Entity-relationship model of task automation for CI/CD

3.2.2.2 Execution environment

As part of this project, two GitLab Runner servers were deployed using a 32 cores CPU and 128 GB memory each. Such configuration allows for the parallel execution of up to 32 jobs concurrently per node. One node runs Microsoft® Windows Server 2019 and the other one Debian 12, hence Linux and Microsoft® Windows jobs are supported by this configuration.

The GitLab Runner service uses a so-called executor that processes the job and sets up the environment. The Docker executor was found to provide the developers with the greatest possible flexibility and was therefore chosen. With this approach, the developer can set the environment themselves by specifying a Docker image, which can contain a minimal installation of an operating system, but also a full custom installation with necessary compilers and tools already available. Docker then creates a container based on the given image, which executes the job. Compared to alternatives, this approach has the advantage that developers get a controlled and well-defined environment like it is given by virtual machines, but without a performance penalty due to the lightweight isolation opposed to rather heavy virtualization.

Docker images are built by instructions from a Dockerfile. Serving all projects included in AC^2, Dockerfiles for essential tasks were developed and the building of those Docker images was automatized. These images provide encapsulated environments for the

CI to build and test AC^2-related code. This includes Docker images for Microsoft® Windows and Linux, containing essential compilers and tools, and a Docker image with all tools to build documentations. The implemented mechanism uses GitLab's CI capabilities as well but runs on dedicated hardware in order not to interfere with production runs. The corresponding pipeline consists of two jobs where the first one builds an actual image and the second one delivers it to a container registry that can be accessed from within GRS's GitLab instance. The second job requires manual activation to make sure that only images are pushed which are tested beforehand.

In summary the overall workflow can be seen in Fig. 3.9. A group of developers maintain the Docker images by providing and updating Dockerfiles. When a Dockerfile is changed, the GitLab runner automatically builds a new Docker image from that and uploads it to the registry. Those images can be used by other projects through the gitlab-ci.yml file to build applications and run tests by the runners. Built applications and test reports are uploaded back to GitLab and provided to developers and GRS-internal users.

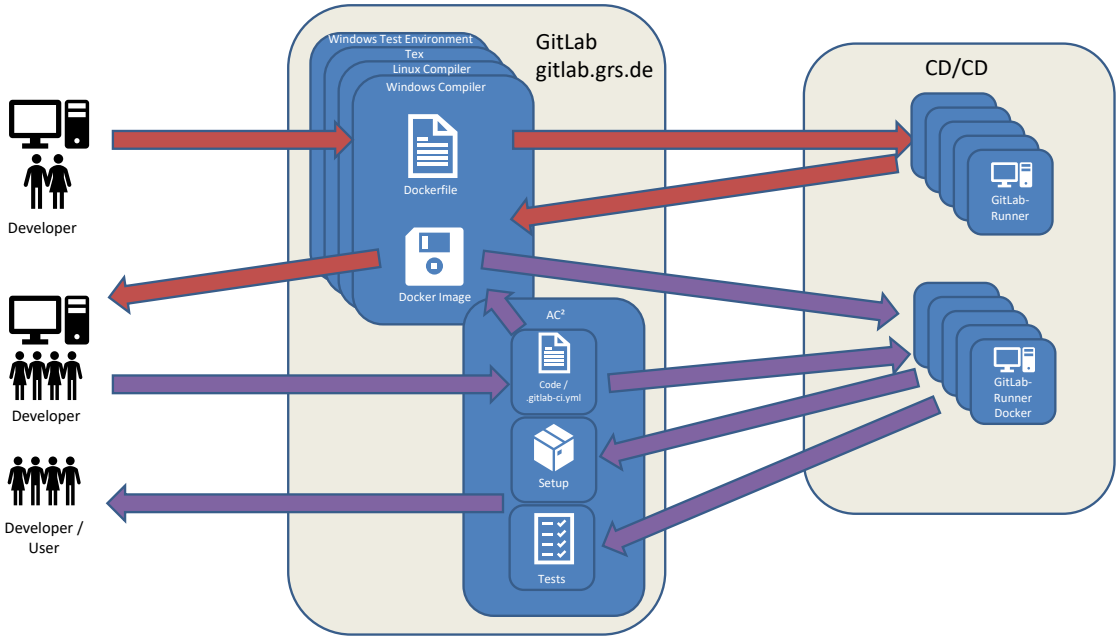

Fig. 3.9 CI/CD workflow using GitLab

3.2.3 Code analysis in NuT

Unit tests can be used to verify that critical functions work without side effects and produce results as specified. A function takes a set of input parameters and produces a corresponding result as output parameters. For unit testing, multiple sets of input

parameters are generated, where the corresponding results are known in advance. Then, the function to be tested is called with each input set and the result is compared with the predefined data. If the result is not within a specified error margin, the unit test failed. Unit tests should be implemented at the same time as the actual function. This helps the developer to verify that the implementation is correct and makes sure that possible regressions due to later changes are detected timely. In NuT, unit tests mainly cover high level functions, as most low level ones are provided by PETSc which are already tested extensively there.

To ensure that the combinations of unit test cases are sufficiently complete, the respective code coverage should be tested as well. This was done with `GCC` and the `gcov` tool, which measures which line was executed and how often. After running all tests, the overall amount of code coverage is determined and a comprehensive report is created, which shows exactly which lines were covered and which were not. In this way, code lines that were not covered by the unit tests can be determined and resolved by adding appropriate test cases.

Several `assert`-instructions were added to the NuT code to check for certain conditions and states of the code. These instructions are only active if the code is compiled in debug mode. Hence, the CI-job for building NuT under Linux and Microsoft® Windows was modified accordingly, giving subsequent test jobs the opportunity to evaluate the `assert`-statements. The described procedure complements the unit testing. Further `assert`-statements can easily be added if considered suitable.

3.2.3.1 Clang tools

The support for several tools that support static code analysis and maintenance of code were added. A configuration file for the tool Clang-Format from the LLVM project /LLVM 23/ was introduced. It defines the formatting style of NuT's code. The employed formatting style in NuT is based on well elaborated presets with minimal deviations. Clang-Tidy is another tool from the LLVM project that is supported by NuT. It is a comprehensive diagnostics tool that warns about poor coding styles that are known to be error-prone or to degrade performance. In addition, certain naming conventions for classes and variables can be enforced, which complements Clang-Format. Both tools can be employed via the language server ClangD, which applies Clang-Format automatically and continuously passes the diagnostic information of Clang-Tidy to IDE, where it is visualized instantly, while the code is written. This standardized way of

writing C++ code helps maintenance of the code and supports a better understanding of the given code lines.

3.2.4 Merge requests workflow

A merge request is a web-based approach to integrate changes done by a developer in a separate branch into the project's main branch.

In the given context of a GitLab instance it provides a workflow to support code reviews and a set of preconfigured rules that need to be satisfied before it can be merged, which is important for quality assurance. A merge request is usually preceded by an issue that provides information about what needs to be solved. After assigning a developer for the issue, a merge request associated with the respective development branch is created and accompanies the further process. It should contain a description about why the changes were necessary, how that was implemented and how verification / validation was done. All staged commits for the merge and their respective changes and CI status are displayed automatically. There is a summary of all changes, hence a code reviewer can easily see what the actual changes to the main branches would be. This is complemented by metrics from the automated code analysis.

The presentation of code coverage by CI tests turned out to be very useful. So far, this feature was implemented for NuT and ATHLET. Other components may follow. In the *Overview* section of a merge request the change in code coverage is shown. Further detailed information about coverage, line by line, can be found in the *Changes* section of the merge request. This makes it easy to see to what extent the new development is covered by tests. For instance, Fig. 3.10 shows a merge request that adds a new function to scale a vector. If merged, the proposed change would decrease the current code coverage to 79%, which is highlighted red. The *Changes* section of the merge request, see Fig. 3.11, shows more detailed information about why the code coverage decreases. New lines of code that are not covered by tests are also marked with a red line. Fig. 3.12 shows the merge request after adding the respective unit test for the new function. Consequently, if merged, the code coverage would improve to 79,09% and therefore marked green. Reviewing the *Changes* section again shows the new function, the complete coverage of the new function and the unit test itself (see Fig. 3.13). This is a great tool to help reviewing the code and motivates developers to deliver tests for their actual implementations. Depending on the project, it might be useful to allow only merge requests that do not decrease code coverage. All proposed changes can be commented and supplemented with code

suggestions, which creates new discussion threads that need to be taken care of by the assigned developer.

After all threads are closed, the reviewer can approve the merge request. Each project specifies a group of members that must approve the merge request to allow merging it. With a feature called *Code Owners*, a set of project members can be defined that are experts for a specific project path. If a merge request intends to alter related files, the approval of the code owners is additionally required. Finally, when all necessary approvals have been given and all rules are satisfied the proposed changes can be either merged directly or via merge trains.

Merging directly has the advantage that no additional CI pipeline needs to be executed. However, even if the last commit of the development branch and of the main branch passed the CI pipeline tests, it doesn't ensure that the merged result also passes. Merge trains provide a remedy to that issue. They create an intermediate merged result commit of main and development branch and trigger a CI pipeline on that. Merging only happens if that pipeline succeeds. This ensures that not only the last commit from the development branch complies with all CI tests, but also the final merge commit.

If multiple merge trains are started concurrently, GitLab accumulates their results. Consequently, the corresponding pipelines can run in parallel and at the same time nothing is merged that hasn't been tested before. Fig. 3.14 shows a merge train with a queue of three merge requests MR1, MR2 and MR3. Each merge request is merged if its pipeline completes successfully and all merge requests before it are merged. With that all three pipelines are executed concurrently. In summary, merge trains guarantee that only commits can be added to production branches that passed CI pipeline tests, and the whole procedure is done without compromising performance.

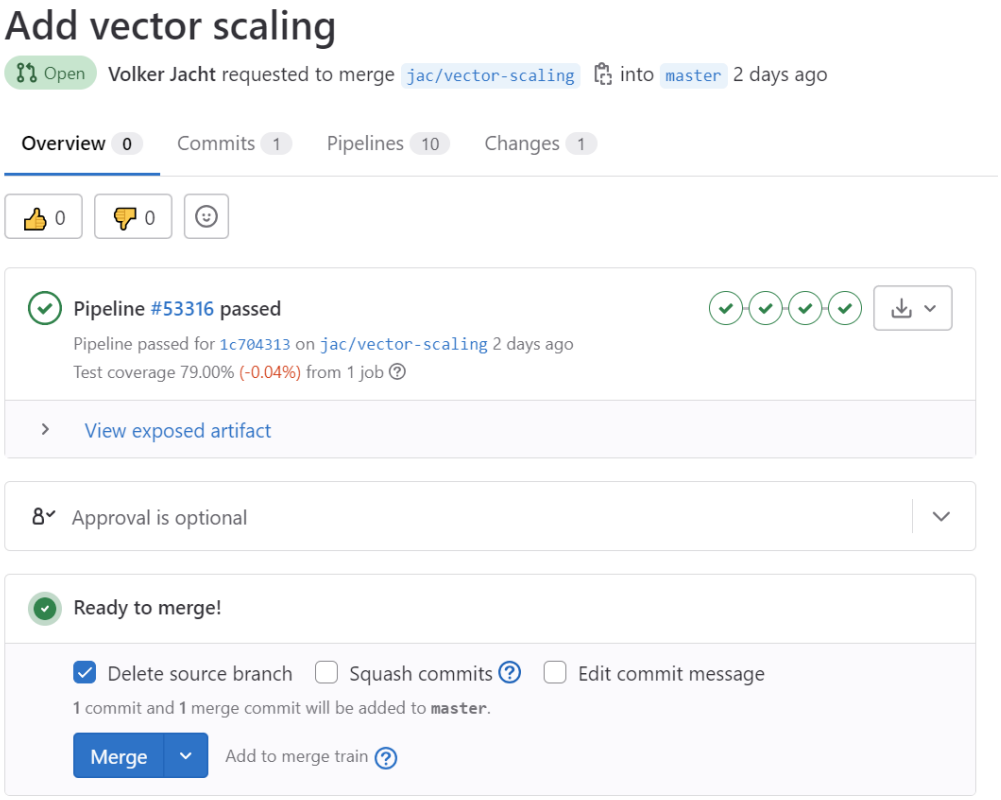

Fig. 3.10 Merge request proposing new untested code, decreasing code coverage

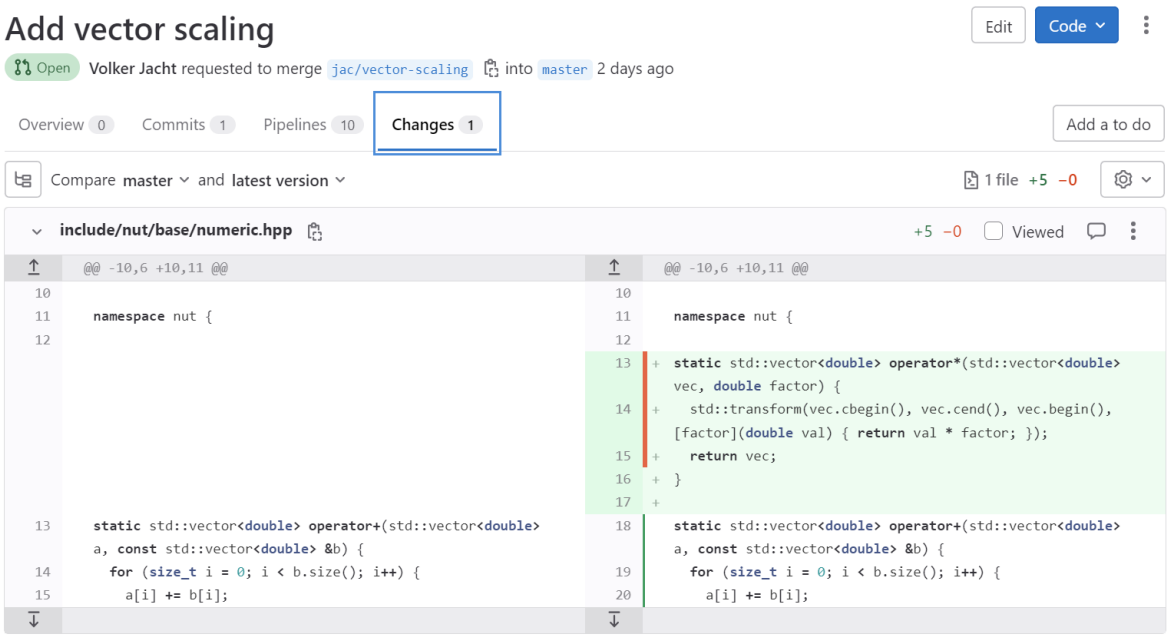

Fig. 3.11 Proposed new code lines are not covered by tests and therefore highlighted red

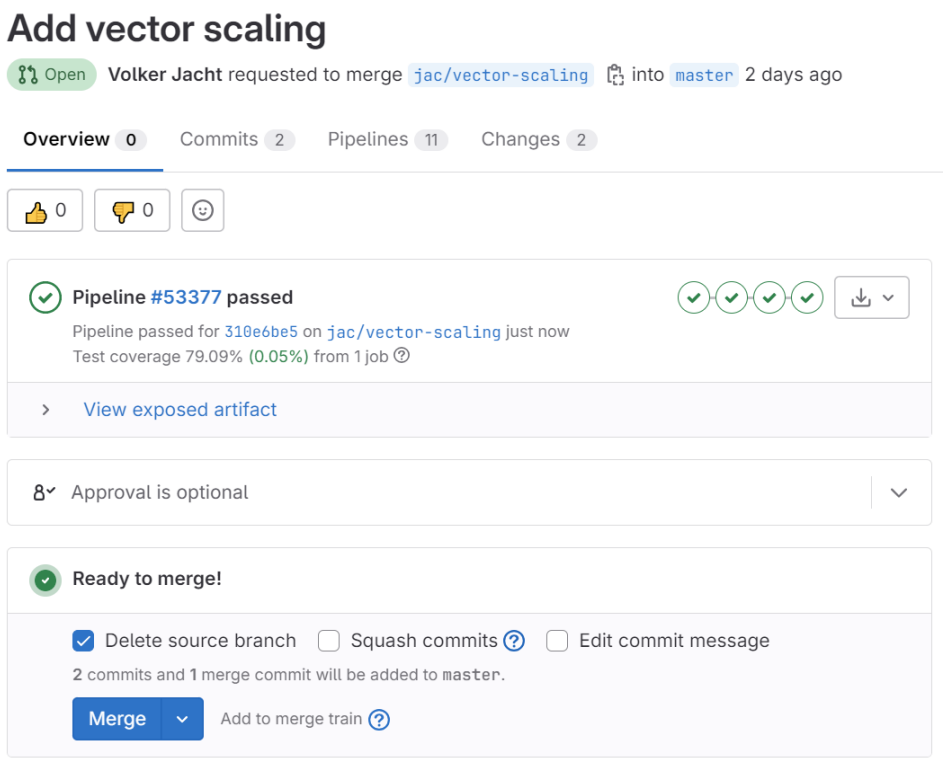

Fig. 3.12 Merge request proposing tested code and increasing code coverage

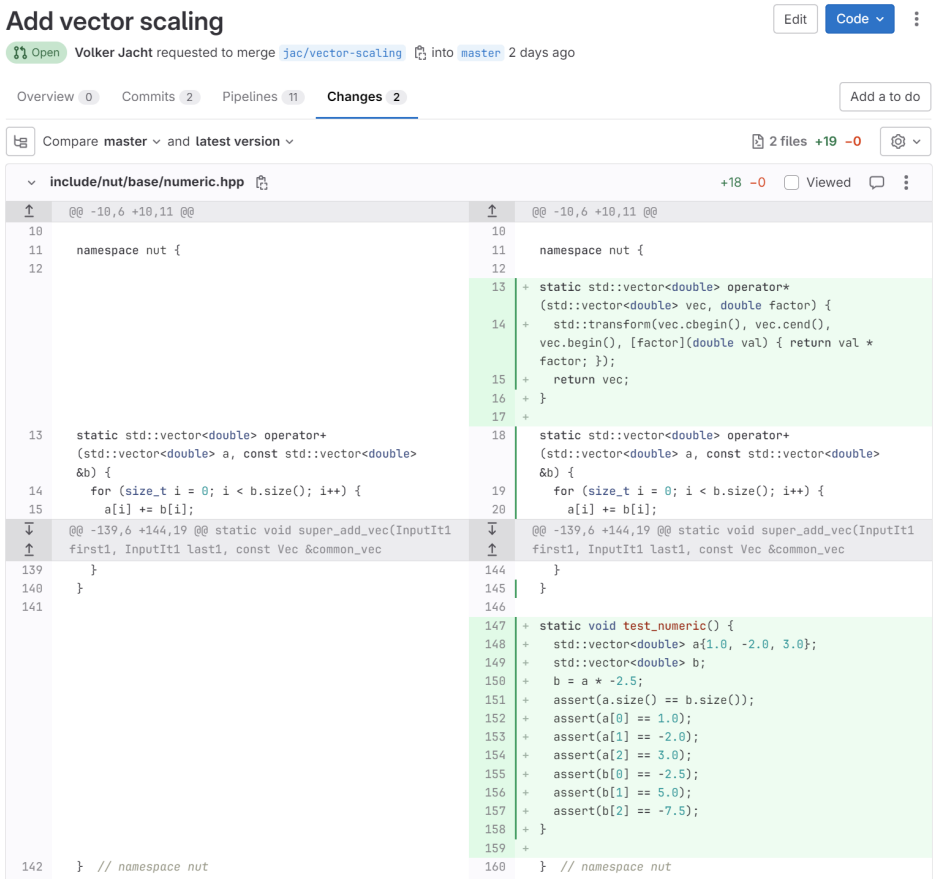

Fig. 3.13 Proposed lines are covered by tests and marked green

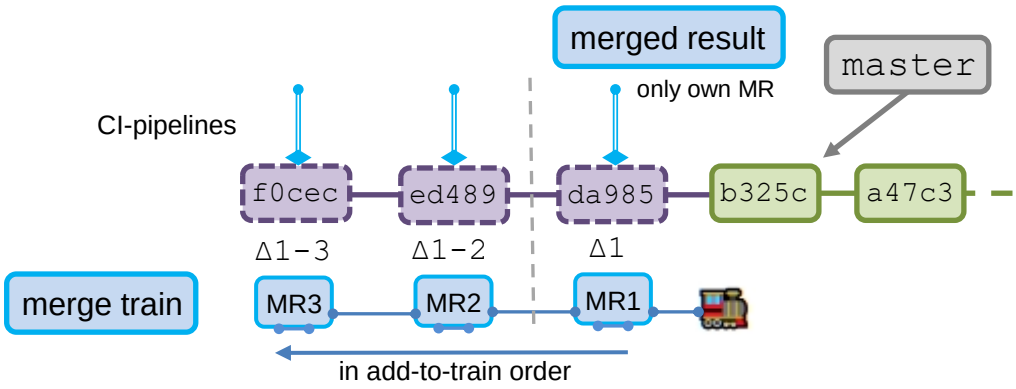

Fig. 3.14 Scheme of a GitLab merge train with three queued merge requests

3.2.5 Project organisation

Guidelines for the organization of groups and projects on GitLab were developed. In general, a group on GitLab can contain multiple projects and (sub)groups. Each of them has a default visibility which is either private, internal or public.

- Private projects can only be accessed by members that have an explicit role
- Internal projects can be accessed by all logged in members of the GitLab instance
- Public projects can be accessed by everyone that can reach the GitLab instance

Members with roles of reporter, developer, maintainer and administrator can be directly assigned to a group or project. Direct roles in groups are recursively inherited to subgroups and included projects. It is not possible to exclude an inherited role from a subgroup or project.

Fig. 3.15 shows the deployed structure. There is one root group GRS that basically contains all project groups. Consolidating everything in a common group simplifies the housekeeping for administrators and allows for greater flexibility when projects need shared items to work with. Only administrators should be members of the main GRS group. A project group usually has two members that are in charge for that group and have owner permissions for it. With that, they can create subgroups and projects within their project group and grant permissions to other members. As groups are mostly for organizational purposes and do not include sensitive data, they should be public by default. This helps new members to get an overview over available project groups and their members. Furthermore, access requests can be made, which are handled by the group owners. Project default visibility is set by the project owners

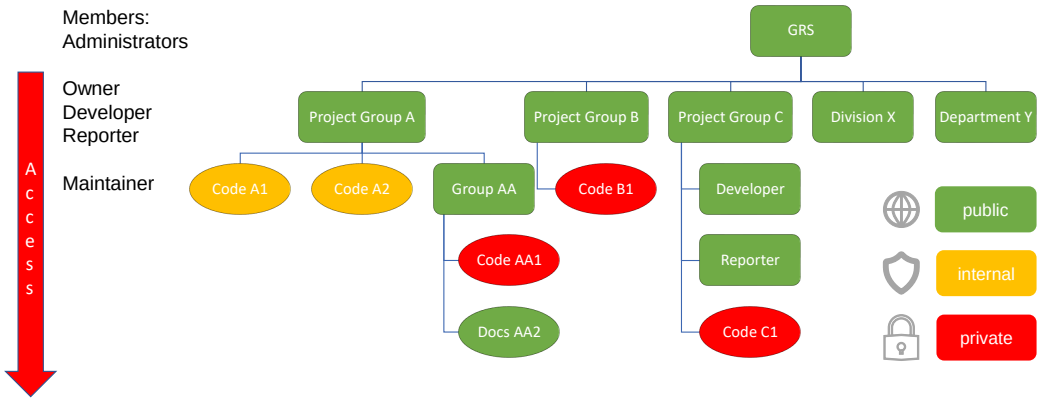

Fig. 3.15 Organisation of groups, projects and members on GitLab

to their needs. Though individual access to single projects can be given, it is not recommended as this can easily get confusing with larger project groups containing many projects. Therefore, as far as possible, members should be assigned to the high-level project groups only, which grants them access to all contained projects respectively.

Various templates for issues and merge requests were written as part of the NuT repository. For issues, standard use cases such as bug report, development, discussion or feature request were considered. Two templates are available for merge requests. A distinction is made as to whether the merge request acts autonomously or can be assigned to an issue and thus essential descriptions are already available. Templates with similar content are also available on the AC^2 level. However, these are rather unsuitable due to certain design decisions. However, the user is free to choose which template to use. In any case, the user is strongly advised to employ the GitLab-internal reference system between issues and merge requests to facilitate navigation and to represent more complex relationships. Such is also explicitly pointed out in the templates provided by the NuT repository.

Remark 3.1. The established workflows are documented in NuT's QA documents /STE 23a/ and in the ATHLET's progammers manual /JAC 23b, Ch. 4/.

3.2.6 Improving software development on the level of AC²

With the tools provided by GitLab the infrastructure for a development cycle for AC² was established, see Fig. 3.16 and Fig. 3.17. A development starts with epics and issues to communicate, document and organize the ongoing work. Next, within the concept of continuous integration, code is developed and tested. After that, continuous delivery makes sure that the current and tested state of the software can be distributed and deployed by users and developers. When problems arise, users can give feedback by creating issues, which closes the development cycle. This process and its automation support all phases of the development and validation process. Developers can focus more on their actual work and be more efficient. Also, the software can be released more frequently as the technical overhead for each release is reduced by the automation. Everything from planning, through development, code reviews, testing and building is completely transparent and reproducible which is key to have fast delivery cycles and high-quality assurance at the same time.

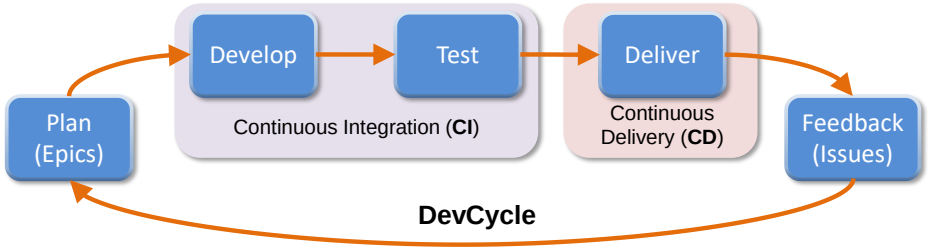

Fig. 3.16 AC² development cycle

3.2.6.1 CI/CD

Currently, the concept mentioned in 3.2.2.1 has been adapted by over 30 projects that use CI/CD regularly in the scope of AC² and since the introduction of GitLab over 440.000 jobs have been processed. It is a key part in the AC² software life-cycle and helps to maintain and improve software quality in a transparent way.

Fig. 3.18 shows the pipeline of the AC² project that creates the AC² installation package for later distribution. Jobs in the prepare and build stages collect all required repositories and compile them on Microsoft® Windows and Linux. The resulting binaries are used in the run stage to create reference plot data included in the release. Next, the built binaries and reference data are used to create an installation assistant and archives. In the last stage, the created archives are used to execute some test

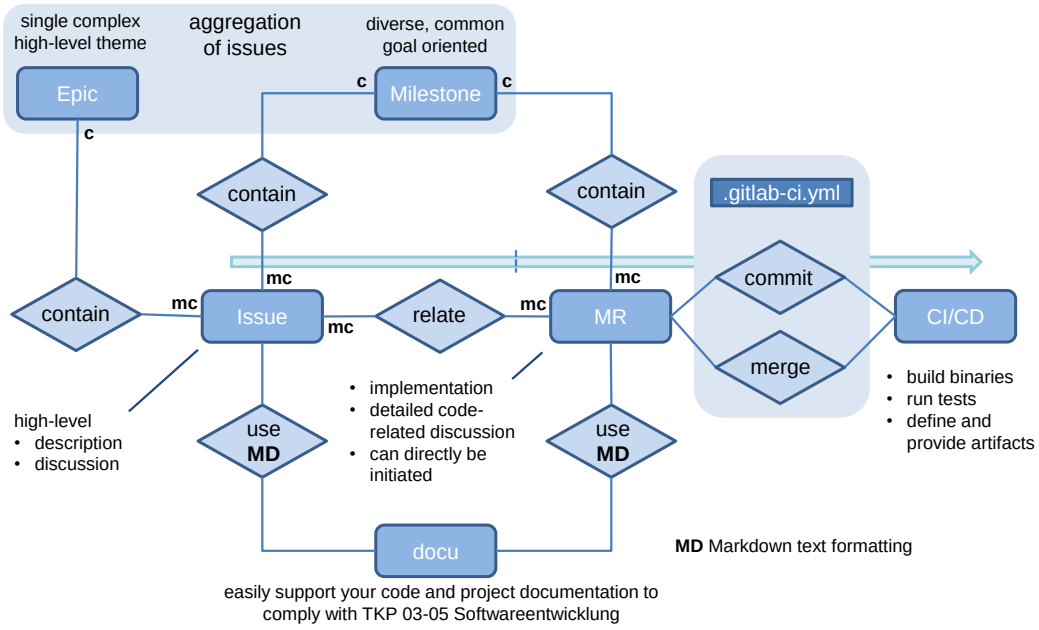

Fig. 3.17 Entity-relationship model of GitLab tools employed by the AC^2 development cycle

simulations in different clean stock environments to verify that the core components work on various platforms. The whole pipeline builds over 400 targets and it takes approximately 38 minutes to create the installation packages for Microsoft® Windows and Linux, respectively.

3.2.6.2 Licenses

As a further addendum to the canon of activities it was taken care of establishing tools for an automatized way of collecting license data within AC^2. Tools and scripts are hosted within a separate repository which makes them easy to include and to maintain.

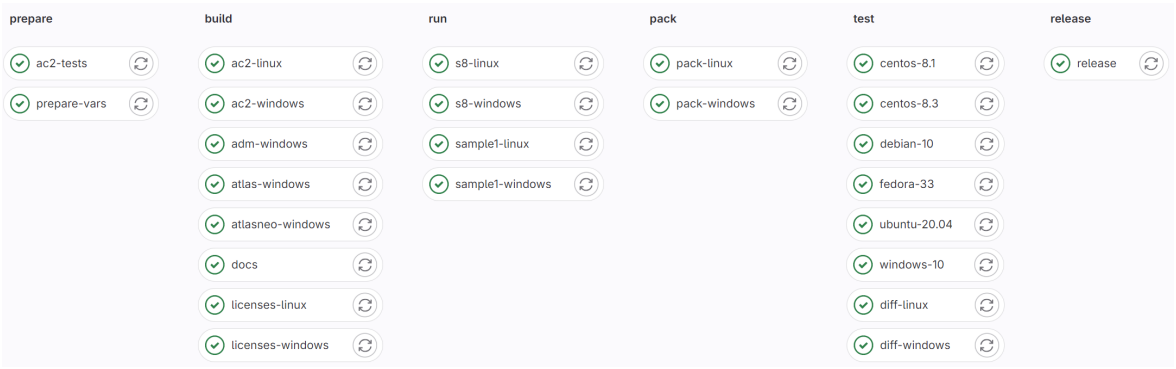

Fig. 3.18 Pipeline that creates the complete AC^2 package for distribution

Collecting given license files and their descriptions is covered by CMake techniques. Processing and converting is done by means of Linux bash tools. Creating a collective presentation makes use of LaTeX. The whole approach is hierarchical. Each project takes care of its own third-party dependencies. Duplicates may arise which are discarded by the collection mechanism. Also, a certain order of licenses may be enforced. Due to the hierarchical nature of the ansatz not only on AC^2-level but also for major single projects automatically created license files are available. The main repositories of AC^2 already support the idea. Accordingly, the new mechanism of processing licenses was applied for the release AC^2 2023.

3.2.6.3 Discarding legacy CI

COCOSYS's CI no longer involves build and test runs on the legacy Jenkins system and is now fully integrated into the runner concept based on Docker images. This comes with the benefit of less maintenance and of more flexible build and testing environments. With these changes, the whole of AC^2 is covered by a Docker-based GitLab CI.

3.2.6.4 Diff jobs

In order to quickly identify changes made to AC^2-related repositories, an auxiliary job was defined on the AC^2 level. Based on the Linux tools `rsync`, `find`, and `diff` the current build can be compared to a reference build. A list is produced that shows all changed files. This makes it easy to narrow down the culprit(s) if a new build behaves erroneously.

3.2.6.5 Supporting external development

GRS not only supplies binary files to end users, but also allows cooperation partners to work directly with the sources, based on a specific license agreement. To provide the respective sources per cooperation, a second GitLab instance was made available by GRS's IT, which can be accessed from outside GRS. Thus, a concept had to be developed to synchronize internal repositories with the external GitLab instance.

It is important to note that a repository should not be mirrored 1:1. It was specified that each time synchronization is triggered, only the current state of certain branches without the internal commit history should be transferred to the second GitLab instance. Furthermore, the synchronization should not affect the internal repositories and the

build process should be handled in the same way, no matter the context. To meet these requirements, a sophisticated interaction of GitLab-CI, CMake, and Python techniques was developed. A new GitLab project was created to store the necessary scripts for synchronization.

In order to flexibly serve different cooperation projects on the external GitLab instance, the GitLab-CI concept of so-called *environments* was used. Environments help to define where code is delivered. A configuration file to be created beforehand lists the desired internal repositories and can be addressed in the manual execution of the synchronization CI via a dedicated variable. The actual synchronization process is handled in a corresponding CI job via a Python script. In addition to the synchronization, a list of deployments to the individual environments (i.e. cooperation projects) is also provided for the sake of a quick overview of the synchronization history.

To handle the build process on the external server the same way as on the internal one, certain features of the CMake script library CMakeIt were used. This is another example that shows the strength of the CMake approach in AC^2 to coordinate the build processes. An additional configuration file is generated in the Python script, which helps CMake to identify where dependencies are to be found on the external server during the initialization of a build process. Specifically, this is done by temporarily redefining CMake variables that are otherwise set in the default CMake configuration files. This change is done transparently to those default scripts, giving the desired result of a unified appraoch to the building process. None of the standard scripts need to be modified. Also analogous to the internal structure, a central package registry is provided for binary files such as runtime libraries. This registry is regularly synchronized with the internal one via a schedule job.

3.2.6.6 Knowledge transfer

In order to implement the task of knowledge transfer, two GRS-internal talks on the use of GitLab were given. The first talk introduced the general systematics of GitLab and explained essential vocabulary and processes. All members of GRS's safety research division were invited to that talk. The second talk focused on the CI possibilities in GitLab. Since not every developer deals with details in this area, the audience was kept smaller, but opened up to a GRS-wide audience. This allowed for more emphasis on details. Both talks are available to any user of the internal GRS GitLab server. Additionally, peer-to-peer support was provided in various ways to foster the unified CI/CD approach for AC^2.

3.3 Assessment of the parallel performance of NuT

In order to obtain a profound evaluation of the HPC performance of NuT, a cooperation with the EU-funded project POP (*Performance Optimisation and Productivity*) /POP 23/ was established. Utilizing POP's expertise is free of charge. However, the cooperation required some formal preparation in the form of permissions to be given by GRS's legal department and export control in order to provide POP with meaningful data. By design NuT is an autonomous piece of software which does not contain any information specific to nuclear technology. Though permission was granted to let POP access NuT's source code, some further internal discussions led to the decision that the performance tests will be run by GRS. No code owned by GRS was made available to members of POP. Furthermore, all of the results of the analyses were reviewed by GRS's export control before they were made available to any member of the POP project.

Tools and setup

To analyse NuT's performance POP provided the tool *Extrae* /BSC 23/. This tool is a dynamic instrumentation package to trace programs compiled and run with the shared memory model (like OpenMP and pthreads), the message passing (MPI) programming model or both programming models. Extrae and its prerequisite *PAPI* (Performance Application Programming Interface /ICL 23/) were installed on GRS's computing cluster *manitu*.

The analysis of NuT was done in the framework of an interaction with an actual application, namely ATHLET. The selected ATHLET model was the artificial *Cube* case, for the reason that it can easily be scaled. The Cube model, see Fig. 3.19, consists of a pipe network forming a three-dimensional cube-shaped grid, with a large number of links between neighboring nodes.

Tests and results

The tests were performed on the GRS computing cluster for different numbers of NuT processes, and the data collected by Extrae were analyzed by POP, see Tab. 3.1 for results. The first round of tests led to the following observations:

- NuT shows a very good overall parallel efficiency,
- on the other hand, poor instruction scalability is given.

The latter result made further evaluation necessary. The low value of the instruction scalability indicates that a significantly higher number of instructions is executed when

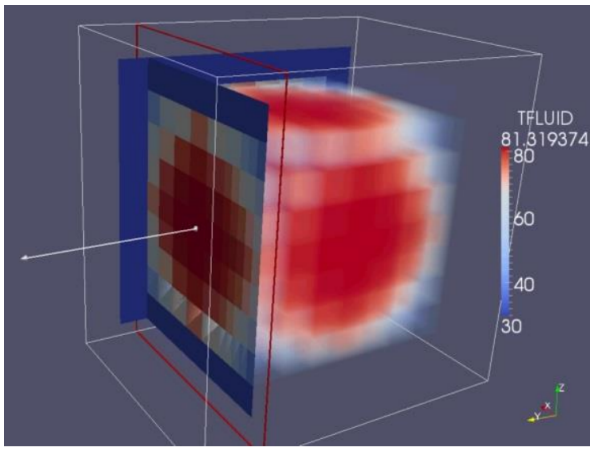

Fig. 3.19 Visualization of the mass flow in the Cube model, originating from a source in the center

the number of NuT processes increases. Theoretically, for an ideal code, the number of useful instructions should be constant. Given only these analysis results, it was not a priori clear in which part of the code these instructions are executed: in the NuT code itself, or in the external numerical libraries PETSc and MUMPS. To get further insight, it was decided by the project partners that additional custom Extrae events shall be defined, which allow to identify the state of the program and to distinguish between NuT code and the external libraries. The necessary modifications of the NuT code were done by GRS, and the analysis was repeated.

Tab. 3.1 POP metrics for the interaction of NuT with ATHLET /ROS 22/. The range of the metrics is from 0 % to 100 %, with a statistical error of ±1 %. The metrics are given in a hierarchical order where each value is obtained by multiplication of the sub-metrics.

Number of NuT processes	1	2	4	8	16
Global efficiency	81.5	63.6	41.8	26.7	12.9
• Parallel efficiency	81.5	88.5	92.1	93.0	93.1
– Load balance	81.5	89.0	93.3	94.6	95.4
– Communication efficiency	99.9	99.5	98.7	98.3	97.6
• Computation scalability	100.0	71.4	45.3	28.7	13.9
– IPC scalability	100.0	98.6	95.8	100.3	100.5
– Instruction scalability	100.0	73.7	48.8	29.8	14.6
– Frequency scalability	100.0	98.8	97.1	96.1	94.8

The corresponding second run of tests revealed that NuT spends a considerable amount of time waiting for new data from ATHLET, see Fig. 3.20. NuT uses the

routine `MPI_Iprobe` to check each connected communicator for new inquiries. These executions of `MPI_Iprobe` are counted by the metrics as useful instructions. Thus, the number of instructions increases with the number of NuT processes, leading to the reported bad value of the instruction scalability. Consequently, the values for the computation scalability and global efficiency are spoiled as well, see Tab. 3.1.

Fig. 3.20 Visualization of tracing with custom events. Color coding only applies to Thread 1.2-5.1. **Top**: value 0: transfer from ATHLET to NuT; value 1: call functions; value 3: waiting for ATHLET; value 2: sending data to ATHLET (not shown since it is negligible). **Bottom**: value 0: PETSc start of computation; value 1: PETSc done.

Conclusions

In the given setup of ATHLET and NuT working in tandem, the value of the instruction scalability can only improve if ATHLET provides its data quicker; it is not an issue in the design of NuT. The most important finding of the analysis is therefore the very good value of the parallel efficiency. It confirms that NuT makes good use of the MPI communication mechanisms. As a consequence of the above results, it is recommended that available computing resources are first used to speed up ATHLET by means of its OpenMP version, before the use of additional NuT processes is considered. In this way, the data delivery of ATHLET can be accelerated, and NuT's instruction scalability, and thus its global efficiency, can be improved. Corresponding recommendations are already included in NuT's manual, see /STE 23b/.

4 WP3 – Reviewing ATHLET's Steady State Calculation on a Conceptual Level

In the scope of this project, conceptual work regarding the ATHLET Steady State Calculation (SSC) was carried out. The investigated ATHLET version is 3.3.0. The SSC was developed in the 1970s and 1980s; its primary documentation is scarce and dates back to this time. A high-level description of the SSC is provided in the ATHLET User's Manual /SCH 23b/. To facilitate code maintenance and further developments, one goal of this project was to provide a rather thorough and up-to-date documentation of the SSC. For this purpose, a re-evaluation of both the thermal-hydraulic models and the numerical methods used was carried out.

In the following sections, the identified numerical algorithms used in the SSC are described and discussed in the context of state-of-the-art numerical methods and software. Furthermore, concepts for improving the SSC are presented.

4.1 General objective of the SSC

The overall goal of the SSC is to initialize the simulation model, which generally consists of thermo-fluid dynamic objects (TFO), heat conduction objects (HCO), neutron kinetics, and GCSM signals, with a limited amount of input parameters in such a way that it is in a steady state, which means that the time derivative of all solution variables should be (approximately) zero. The advantages of performing this SSC, compared to making the user solely responsible to define a reasonably balanced state with input data, are:

- The user input is limited to a manageable amount.
- Inconsistent input is corrected.
- If the SSC is successful, it is ensured that the system is (almost) in a steady state. This minimizes the occurrence of undesired initial transients, and simulation run failures at the start of the calculation are mostly averted.

4.2 Procedure of the SSC

A flow chart of the SSC on high abstraction level is shown in Fig. 4.1. All ATHLET modules, the Thermo-Fluid Dynamic module (TFD), the Heat Conduction and Heat

Transfer module (HECU), the Neutron Kinetics module (NEUKIN), and the General Control Simulation Module (GCSM), are affected by the SSC. The outer iteration loop is done because changes of a variable in one module may affect variables in other modules. This mainly concerns changes in the heat flow calculated within the HECU module which affect variables in the TFD module. Within the outer loop, inner iteration loops are performed for the modules TFD and HECU. A more detailed flow chart of the SSC is shown in Fig. 4.2.

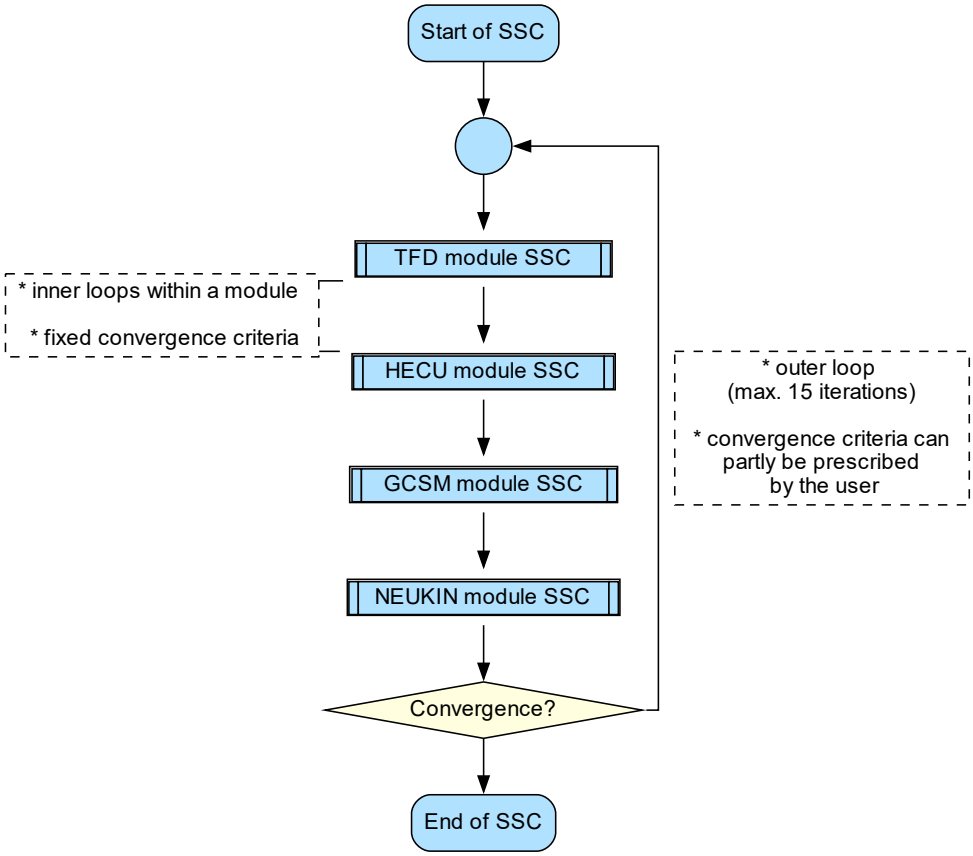

Fig. 4.1 Flow chart of the SSC on high abstraction level

In Fig. 4.3 and Fig. 4.4 a simplified call graph of the most important routines of the SSC is visualized. The colors indicate which module a routine belongs to. Gray means that the routine does not belong to any of the four modules. The routine `HCENBA` has two colors as it is relevant for both, TFD and HECU.

Remark 4.1. A short description of the routines mentioned in the flow charts and call graphs, as well as a list of the global variables used in ATHLET, can be found in /JAC 23b/.

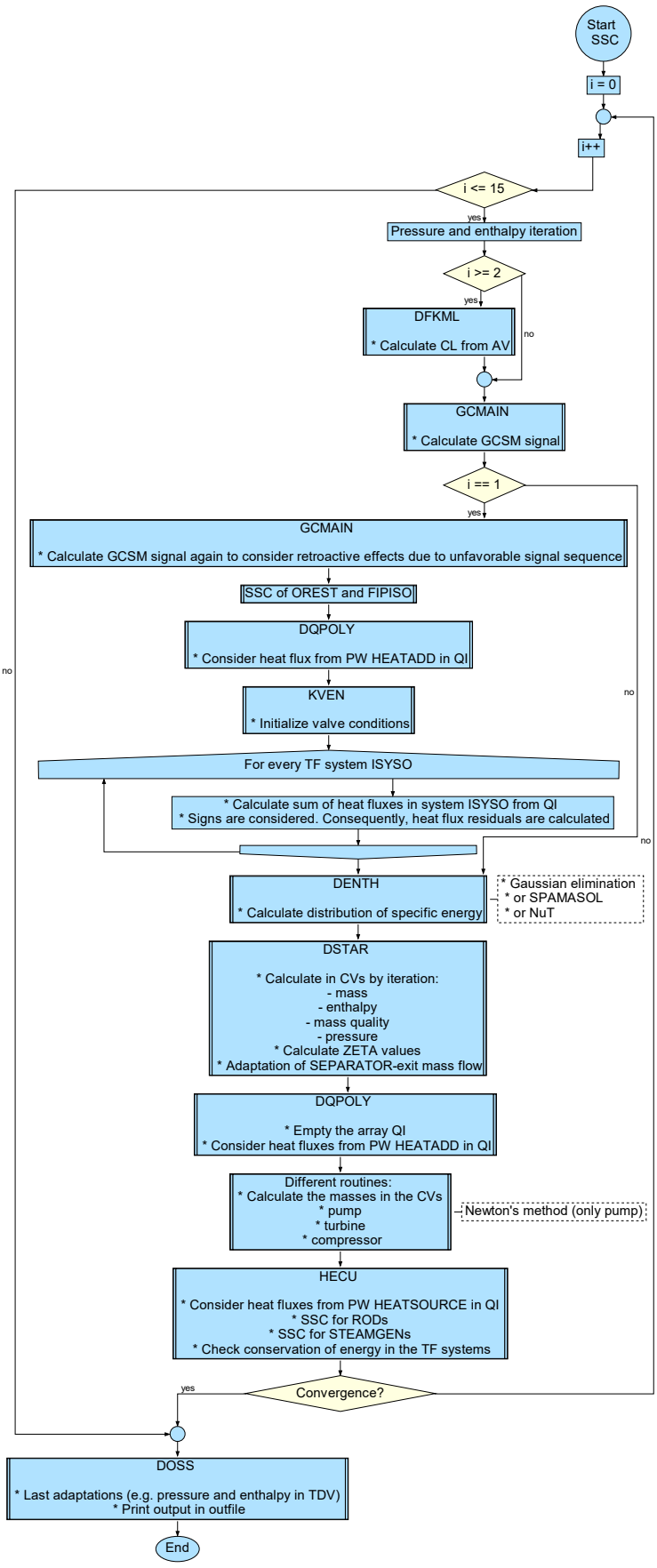

Fig. 4.2 Flow chart of the SSC on a medium abstraction level. The headlines in the boxes indicate ATHLET subroutines.

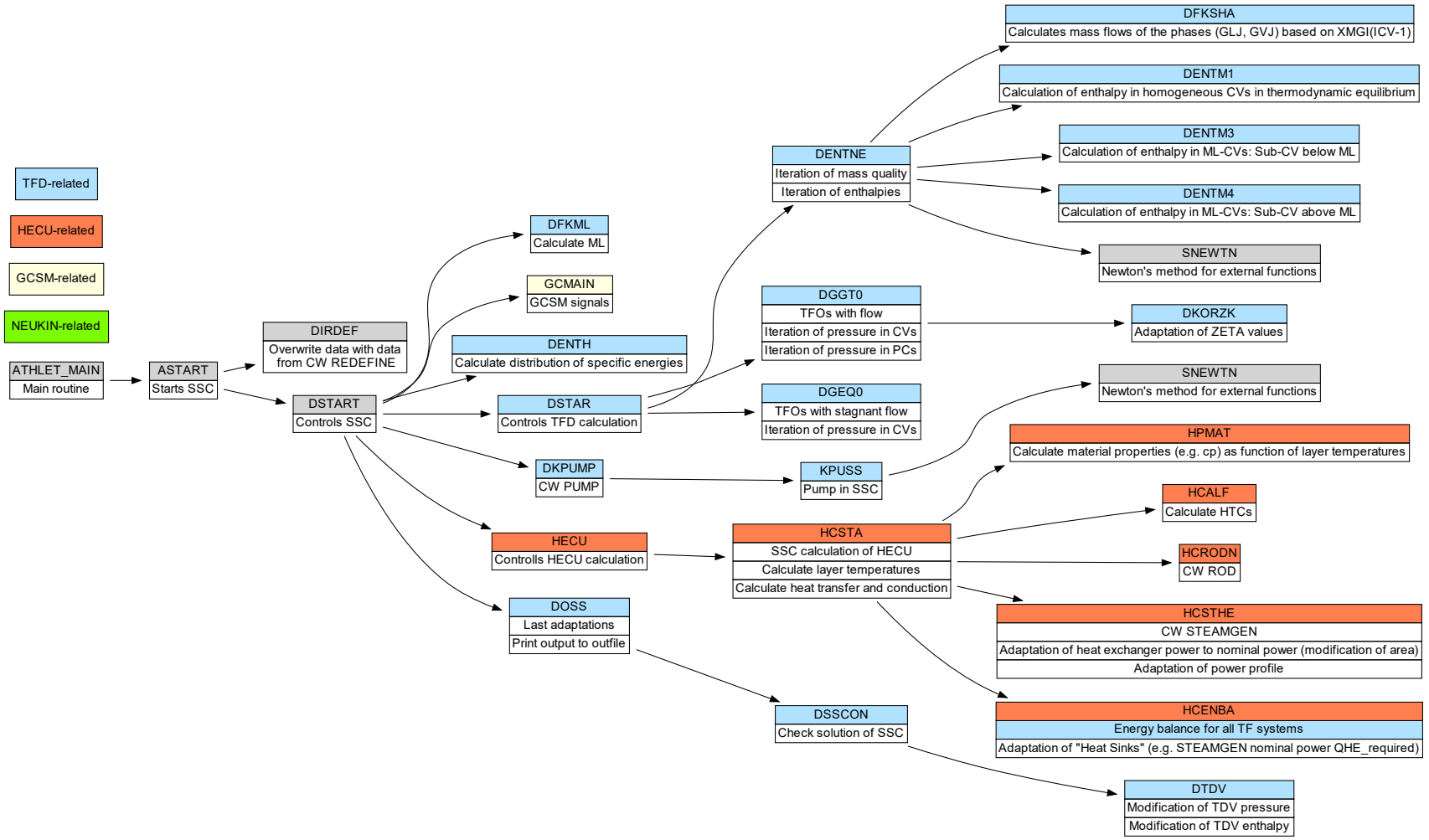

Fig. 4.3 Simplified call graph of the SSC – left part

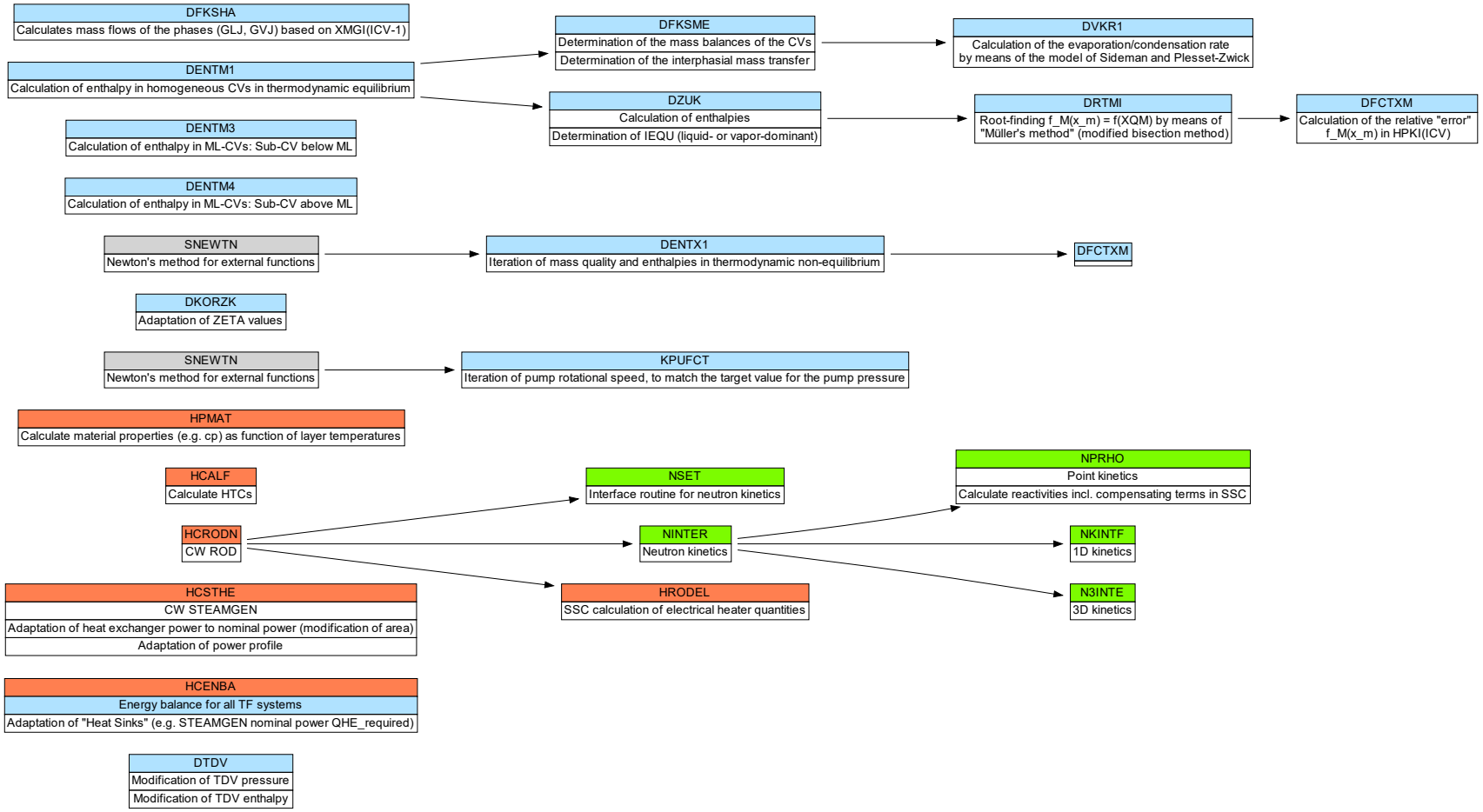

Fig. 4.4 Simplified call graph of the SSC – right part

4.3 Overview of currently used algorithms and thermal-hydraulic models

To achieve the goals of the SSC listed in Section 4.1, a combination of direct algebraic solutions and nested iterations is performed in the current implementation. The main role has the TFD module, in which the 4-equation model is applied during the SSC. This model solves the equations for conservation of mass in the control volumes (CV) separately for liquid and vapor phase and conservation of momentum and energy for the mixture. The solution variables in the SSC are:

- `XQM`: mass quality of the mixture in a CV
- `HDOM`: specific enthalpy of the dominant phase in a CV. Here, the model assumption is that only the dominant phase can deviate from saturation conditions. The other phase is always at saturation state.
- `PRESS`: static pressure in a CV
- `GJ`: mass flow rate of the mixture between CVs

Although the objective of the SSC is to provide stationary conditions, for two-phase flow small disturbances may occur during the start of the transient simulation. This is partly due to the fact that in the transient simulation the 5- or 6-equation model is used and there both phases might deviate from saturation conditions, and partly due to the fact that in the transient phase the full functionality of the mixture level and non-condensable gas models as well as GCSM and NEUKIN comes into play.
During the SSC, a large number of thermal-hydraulic models (e. g. for calculation of heat transfer coefficients) are called. These models are not SSC specific, they are also used during transient simulation. A detailed description of these models can be found in /SCH 23a/.

4.3.1 Iteration loops

4.3.1.1 Outer iteration loop

The outer iteration loop includes all ATHLET modules, see Fig. 4.1). The loop is executed until the convergence criteria are met, but not for more than 15 iterations. If the convergence criteria are not met within 15 iterations, the outer iteration loop is exited. Nevertheless, the transient ATHLET simulation is started even if the convergence criteria are not met. Higher disturbances can be expected in that case compared to a converged SSC.

Convergence criteria apply to the modules TFD, HECU and NEUKIN, while there is no convergence criterion for the GCSM module. All of them must be fulfilled to achieve convergence. The following explicit criteria are used:

$$\left| \frac{\mathtt{QH}_j^{\langle i+1 \rangle} - \mathtt{QH}_j^{\langle i \rangle}}{\mathtt{QH}_j^{\langle i+1 \rangle}} \right| < \varepsilon_{\mathtt{QH}} \tag{4.1a}$$

$$\left| \frac{\mathtt{QHE}_{k,\mathrm{req}}^{\langle i+1 \rangle} - \mathtt{QHE}_k^{\langle i \rangle}}{\mathtt{QHE}_k^{\langle i+1 \rangle}} \right| < \varepsilon_{\mathtt{QHE}} \tag{4.1b}$$

$$\frac{\sum \dot{E}_\ell}{\sum \left| \dot{E}_\ell \right|} < \varepsilon_E \tag{4.1c}$$

$$\left| \frac{W_m^{\langle i+1 \rangle} - W_m^{\langle i \rangle}}{W_m^{\langle i+1 \rangle}} \right| < \varepsilon_{P_{\mathrm{el}}} \tag{4.1d}$$

In (4.1a) $\mathtt{QH}_j^{\langle i+1 \rangle}$ denotes the heat flow (both QHL and QHR – see Fig. 4.5 for definition) in the heat conduction volume (HCV) with index j for the current iteration step $i+1$. This criterion is evaluated for every HCV in the simulation domain (in routine HCSTA) and is related to the HECU module.

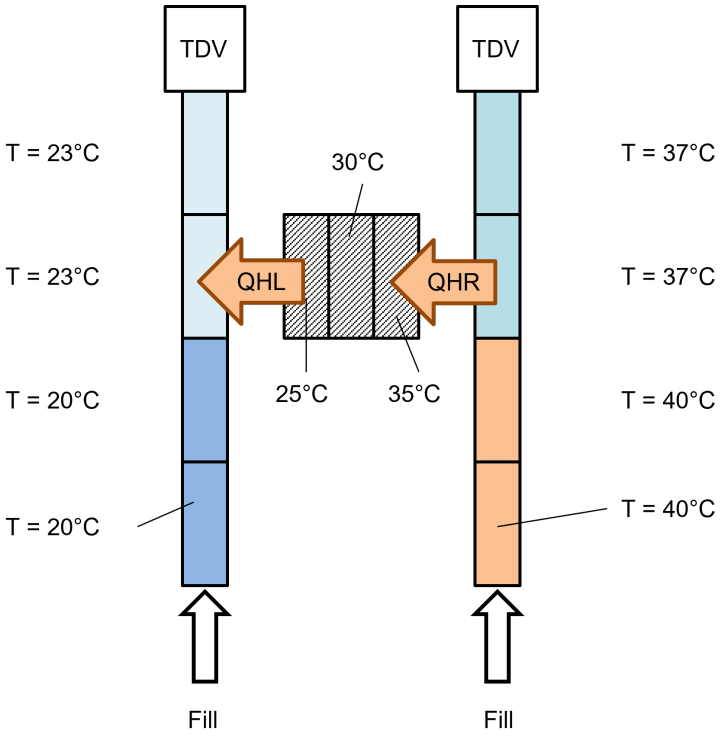

Fig. 4.5 Heat flow through a HCV. QHL denotes the heat flow on the left side. QHR is the heat flow on the right side. In steady state, QHL=QHR.

QHE in (4.1b) denotes the heat flow for the STEAMGEN component (note: the terms STEAMGEN, HTX, HTEX or heat exchanger are used in ATHLET for the same component;

the different designations are historical). Here $\mathtt{QHE}_k^{\langle i+1\rangle}$ is the calculated heat flow for iteration $i+1$ and $\mathtt{QHE}_{k,\mathrm{req}}^{\langle i+1\rangle}$ is the heat flow required to be transferred in iteration step $i+1$ for the heat exchanger with index k. Note that this is usually not the same as the input value $\mathtt{QHE}_{k,\mathrm{req}}^{0}$, as this value is modified during the SSC (in connection with the heat balance of the overall system and the adjustments of the heat sinks; this modification takes place in routine HCENBA, see description for (4.1c)). This criterion is evaluated for every STEAMGEN (routine HCSTHE).

$\dot{E}_\ell$ in (4.1c) is the energy flux in the thermo-fluiddynamic system ℓ. Consequently, the term on the left hand side of (4.1c) denotes an energy residual. This criterion is evaluated for every modeled thermo-fluiddynamic system in routine HCENBA. Due to this criterion, TFD properties have an impact on the energy balance, which in turn affects $\mathtt{QHE}_{k,\mathrm{req}}^{\langle i+1\rangle}$ and thus the previous convergence criterion.

The convergence criterion (4.1d) applies to electrical heaters. $W_m^{\langle i+1\rangle}$ is the specific heat generation in layer m. Contrary to the first three convergence criteria, $\varepsilon_{P_{\mathrm{el}}}$ is fixed, see Section 4.4.5, and cannot be set by the user.

The convergence criterion for neutron kinetics depends on the model. For applications that use point kinetics, no convergence criterion must be fulfilled, instead the reactivity is shifted to achieve the specified nuclear power. Using 3D kinetics, the convergence criterion is defined by an external software that is called via plugin.

The following adjustments are made as part of the outer iteration loop:

- Reactivity coefficients (neutron kinetics)
- Heat exchanger surface and power profile
- Heat sinks (STEAMGEN, condenser and insulation losses)
- Turbine and compressor data
- Exit mass flow of the separator
- Pressure and enthalpy in the TDVs (time dependent volumes); strictly speaking, these are not modified during, but immediately after the outer iteration loop. These modifications can be undesired, see Section 4.6.2.

4.3.1.2 Inner iteration loop

Unlike the outer iteration loop, each inner iteration loop has fixed convergence criteria that cannot be changed by the user. If these criteria are not met within a fixed number

of iteration steps, the SSC and consequently the overall ATHLET run is terminated with an error message. Iterations are not performed for all ATHLET modules:

- The NEUKIN module is called by the SSC and modifications are made (adding reactivity terms, to keep the reactor critical during SSC), but no inner iteration takes place.
- The GCSM module is called by the SSC to allow signal updates, but no inner iteration takes place.
- For the modules TFD and HECU inner iterations are performed.

TFD

Within the TFD module, the following calculations are done as part of the inner iteration loop:

- Specific energy in the CVs: For this purpose, a system of linear equations is solved using the Gaussian elimination method or alternatively NuT. The applied method with NuT depends on the chosen solver preset, see /STE 23b/.
- Enthalpies and mass qualities in the CV: Müller's method and thereafter the two-dimensional Newton's method are applied.
- Pressure in the CV: Pressure is calculated in every CV iteratively using fixed-point iteration. For a closed loop, the system of equations would be over-determined. For that reason, the friction coefficients are modified as part of the iteration in the case of a closed loop. The modification of the friction coefficients is performed iteratively using the one-dimensional Newton's method.

As the fluid properties in a CV depend on pressure and enthalpy, these two quantities are iterated until convergence. Furthermore, the rotational speed of pumps is adapted in the TFD module in its own inner loop until the target value for the pump pressure is met, by means of Newton's method.

HECU

An inner loop takes place within the HECU module to calculate the layer temperatures of the HCV. These depend, among other things, on material properties like the thermal conductivity λ, which in turn are temperature-dependent: $T\Big(\ldots, \lambda\left(T\right)\Big)$. For that reason, the solution is calculated iteratively using a fixed-point iteration. In case of poor convergence damping is applied.

4.3.2 Algorithms for the solution of equation systems

The following algorithms for the solution of equations and equation systems are used in the current implementation of the SSC:

- Direct solution of systems of linear equations:
  - Gaussian elimination method (specific energy in the CV)
  - Alternatively NuT (specific energy in the CV)
  - Tridiagonal matrix algorithm (calculation of layer temperatures)
- Iterative solution:
  - Müller's method, a modified bisection algorithm using inverse parabolic interpolation (mass quality; routine `DRTMI`)
  - Standard fixed-point iteration (pressure in the CV; routines `DGGTO` and `DGEQO`)
  - Fixed-point iteration with damping in case of poor convergence (layer temperatures; routine `HCSTA`)
  - Newton's method (pressures and modification of friction coefficients, enthalpies and mass qualities in the CV, pump rotational speed; routine `SNEWTN` or directly implemented in the routines `DGGTO` and `DSTAR`)
  - Modified regula falsi is performed for mixture level CV (routine `DENTNE`)

4.4 Detailed description of relevant algorithms

In the following, a description of the algorithms listed above is given.

4.4.1 Iteration of enthalpy and mass quality

First, the distribution of the specific total energies

$$e = h + \frac{w^2}{2} \tag{4.2}$$

is calculated within the outer iteration loop in routine `DENTH` in all CVs by solving a system of linear equations. After that, `DSTAR` is called within the outer iteration loop, see Fig. 4.2. In `DSTAR` the calculation of mass qualities and enthalpies takes place similarly to the pressure iteration at the basal network level: `DENTNE`, the routine used for the enthalpy and mass quality iteration, is called individually for each junction, see Fig. 4.6.

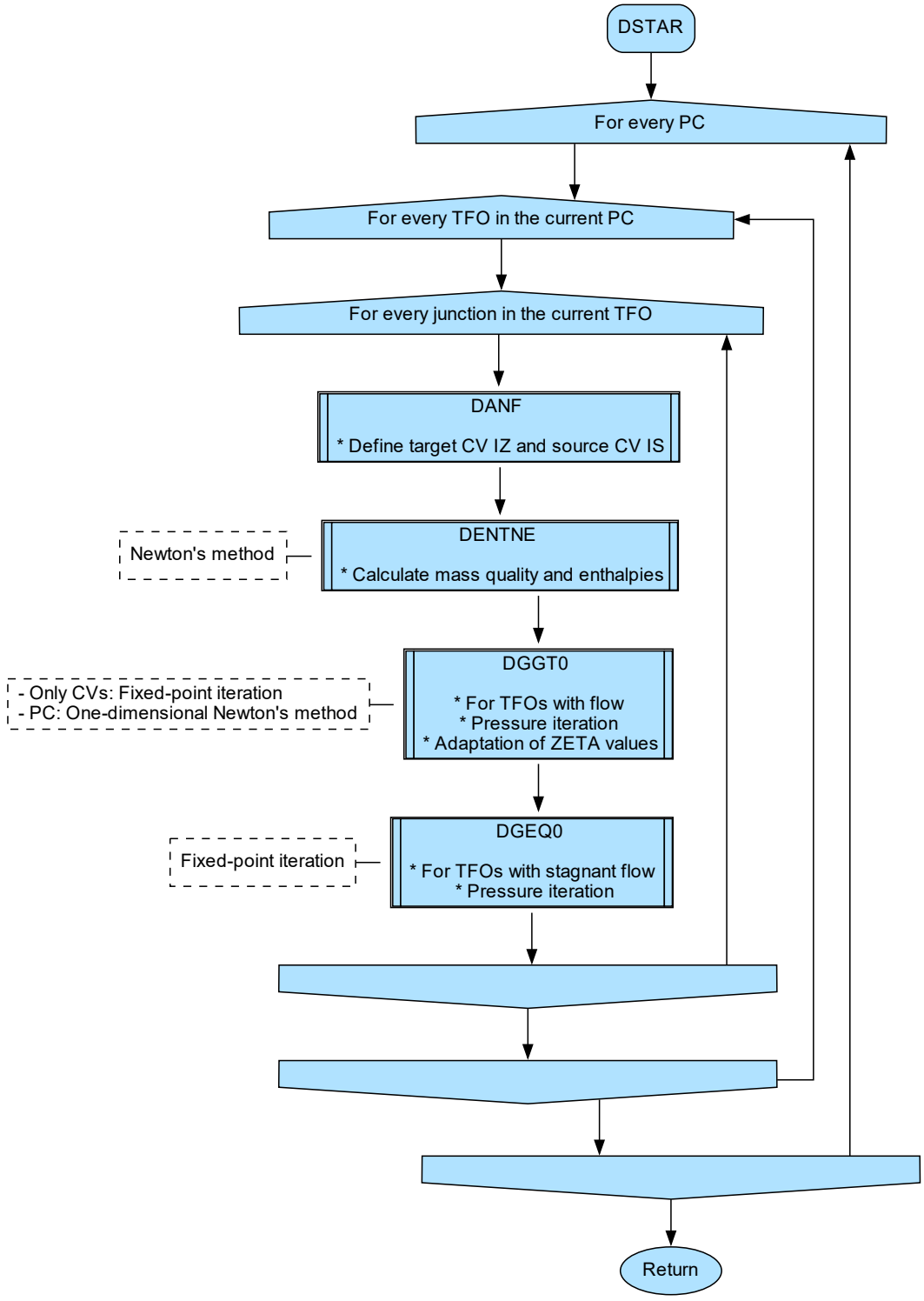

Fig. 4.6 Flow chart of the SSC in `DSTAR`

The following is an outline of the process of enthalpy and mass quality iteration. Since it is the most complex one, only the case *two-phase flow in thermodynamic non-equilibrium* (i. e. both vapor and liquid are present and both phases have different velocities and temperatures) is addressed. The process is described below mainly in text form. A description in equation form can be found in Appendix B.

4.4.1.1 Goal of the iterations

By solving the system of linear equations in DENTH, the specific total energy is known in every CV, but neither the enthalpy h nor the mass quality x_m. However, enthalpy and mass quality are solution variables in the 4-equation model that is used in the SSC. For that reason, the goal of the iterations is the determination of the following variables for every CV:

- mass quality x_m,
- enthalpy h,
- the dominant phase (liquid or vapor).

4.4.1.2 Procedure

Basically, Newton's method (routine SNEWTN) is used to iterate the mass qualities and enthalpies in the homogeneous CV of the TFO with flow. However, Newton's method is preceded by Müller's method that calculates the initial values for the iteration with Newton's method. Müller's method (= bisection with inverse parabolic interpolation) is implemented in the routine DRTMI. A more detailed description of Müller's method can be found in /POI 78/. Determining the unknowns, the code progressively reduces constraining conditions:

1. In order to provide initial data for Müller's method, thermal and mechanical equilibrium of liquid and vapor is assumed.
2. During the application of Müller's method thermal equilibrium is still assumed, however, mechanical non-equilibrium is possible. The results of Müller's method serve as initial values for Newton's method.
3. For Newton's method both thermal and mechanical non-equilibrium are allowed.

The following course of actions take place in DENTNE and its auxiliary subroutines. As DENTNE is called CV by CV, the computations are performed for a single CV. All mentioned quantities (e. g. e or x_h) always refer to the CV under consideration.

Preparation of the initial values for Müller's method

In the Müller iteration, the static vapor quality $x_m = \frac{m_{\mathrm{vap}}}{m_{\mathrm{tot}}}$ in a CV is the unknown quantity. Since the iteration is basically a bisection algorithm, a lower bound $x_m^{\langle 0,\mathrm{low}\rangle}$ and an upper bound $x_m^{\langle 0,\mathrm{up}\rangle}$ have to be provided as initial values.

Regarding the lower bound, $x_m^{\langle 0,\mathrm{low}\rangle} = 0$ would be the physical limit, however

$$x_m^{\langle 0,\mathrm{low}\rangle} = 10^{-12} \tag{4.3}$$

is chosen in ATHLET (subroutine `DZUK`). This value seems to be arbitrary and is possibly used as a very small value close to, but still larger than zero in order to avoid numerical problems. So far, this choice of the lower bound has proven itself in practice.

Concerning the upper bound, $x_m^{\langle 0,\mathrm{up}\rangle} = 1$ would be the physical limit. However, presumably in order to make the bisection range narrower, a smaller value closer to $x_m^{\langle 0,\mathrm{low}\rangle}$ is prepared, apparently based on the following reasoning:

Having the equation which is to be iterated in Müller's method in mind (4.13), it is clear that

$$e_{\mathrm{cur}} = e_{\mathrm{tar}} \tag{4.4}$$

must be fulfilled, with e_{tar} being the desired specific total energy (obtained as `HPKI` via the solution of a linear equation system as mentioned above), and e_{cur} as the specific total energy of the current iteration step. The quantity e_{cur} can be expressed as a function of the mass flow quality $\dot{x}_{m,SR} = \frac{\dot{m}_{\mathrm{vap}}}{\dot{m}_{\mathrm{tot}}}$:

$$e_{\mathrm{cur}} = \dot{x}_{m,SR} \cdot e_{\mathrm{vap}} + (1 - \dot{x}_{m,SR}) \cdot e_{\mathrm{liq}} = e_{\mathrm{tar}}. \tag{4.5}$$

The indices vap and liq describe the vapor and liquid phase. Since kinetic energy is usually small compared to enthalpy, the equation can be approximated by:

$$e_{\mathrm{cur}} = \dot{x}_{m,SR} \cdot h_{\mathrm{vap}} + (1 - \dot{x}_{m,SR}) \cdot h_{\mathrm{liq}} = e_{\mathrm{tar}}. \tag{4.6}$$

Due to the assumption of thermal equilibrium (both phases are at saturation conditions), this becomes

$$e_{\mathrm{cur}} = \dot{x}_{m,SR} \cdot h'' + (1 - \dot{x}_{m,SR}) \cdot h' = e_{\mathrm{tar}}. \tag{4.7}$$

Re-arranging the equation yields:

$$\dot{x}_{m,SR} = \frac{e_{\mathrm{tar}} - h'(p)}{h''(p) - h'(p)} = \frac{e_{\mathrm{tar}} - h'(p)}{\Delta h_v(p)}. \tag{4.8}$$

Since mechanical equilibrium is assumed at this stage (i. e. vapor and liquid have the same velocity and thus $\dot{x}_{m,SR} = x_m$), one obtains

$$x_m = \frac{e_{\mathrm{tar}} - h'(p)}{\Delta h_v(p)} = \frac{\mathtt{HPKI} - h'(p)}{\Delta h_v(p)}. \tag{4.9}$$

As explained in Appendix B.1.1.1, for co-current two-phase flow in mechanical non-equilibrium, usually $\dot{x}_{m,SR}(x_m) > x_m$ holds true which means that (4.9) is not the root of (4.13), but a reasonable upper bound for Müller's method:

$$x^{\langle 0,\mathrm{up}\rangle} = \frac{e_{\mathrm{tar}} - h'(p)}{\Delta h_v(p)} = \frac{\mathtt{HPKI} - h'(p)}{\Delta h_v(p)}. \tag{4.10}$$

This can be found in the source code in `DZUK`, see Code 4.1.

Code 4.1 Calculation of (4.10) in routine `DZUK`

```
Y(JH1+I) = HPKI(I)                                                    1
XHI      = (HPKI(I)-HSWI)/HWDI                                        2
```

Execution of Müller's method

Performing Müller's method serves as preparation of Newton's method. Thermal equilibrium and mechanical non-equilibrium are assumed for Müller's method. The iteration procedure is implemented in `DRTMI`. The scalar function for which a root is to be found is implemented in `DFCTXM`.

Thermal equilibrium is assumed, which means that both phases are at saturation conditions and $x_m = x_h = \frac{h-h'}{\Delta h_v}$ applies. However, mechanical equilibrium is not assumed anymore, which means that vapor and liquid can have different velocities, and consequently momentum exchange between the phases can occur. The effect of x_m on the phase slip (slip ratio = $v_{\mathrm{vap}}/v_{\mathrm{liq}}$) and consequently on the specific total energy in the CV is non-trivial. For that reason, an iterative procedure is applied.

1. x_m determines the volume fraction α
2. α determines the slip ratio
3. the slip ratio determines the phase velocities and the mass flow quality $\dot{x}_{m,SR}$ (SR stands for slip ratio)
4. $\dot{x}_{m,SR}$ determines the specific total energy in the CV, see (4.11)

Basically, when performing Müller's method, x_m is iteratively changed within the previously determined interval limits until the target value in the CV is reached. Here,

the target value is the specific total energy e_{tar} (= HPKI) that is prescribed in DENTH.

$$e_{\mathrm{cur}} = \dot{x}_{m,SR} \cdot \left(h_{\mathrm{vap}} + \frac{w_{\mathrm{vap}}^2}{2} \right) + (1 - \dot{x}_{m,SR}) \cdot \left(h_{\mathrm{liq}} + \frac{w_{\mathrm{liq}}^2}{2} \right). \tag{4.11}$$

As thermal equilibrium is assumed, the specific enthalpies of both phases can be considered as saturation enthalpies h' (liquid) and h'' (vapor). For that reason, an iteration must only be performed for x_m but not for the enthalpies. Therefore, (4.11) becomes:

$$e_{\mathrm{cur}} = \dot{x}_{m,SR} \cdot \left(h'' + \frac{w_{\mathrm{vap}}^2}{2} \right) + (1 - \dot{x}_{m,SR}) \cdot \left(h' + \frac{w_{\mathrm{liq}}^2}{2} \right). \tag{4.12}$$

As already mentioned, the target value of the iteration is e_{tar} (= HPKI). The implementation in DFCTXM determines the deviation $e_{\mathrm{tar}} - e_{\mathrm{cur}}$ relative to the evaporation enthalpy in the CV:

$$f_M(x_m) = \frac{e_{\mathrm{tar}} - e_{\mathrm{cur}}}{\Delta h_v}. \tag{4.13}$$

Two convergence criteria must be met. The first one evaluates the endpoints of the interval for the current iteration step i:

$$x_m^{\langle i,\mathrm{up}\rangle} - x_m^{\langle i,\mathrm{low}\rangle} \leqslant 10^{-10} \cdot x_m^{\langle i,\mathrm{up}\rangle}. \tag{4.14}$$

The second convergence criterion is as follows:

$$f_M\left(x_m^{\langle i,\mathrm{up}\rangle}\right) - f_M\left(x_m^{\langle i,\mathrm{low}\rangle}\right) \leqslant 10^{-10} \cdot 100. \tag{4.15}$$

The value 10^{-10} is hard-coded in DZUK.
After the iteration, in DENTNE/DENTM1/DZUK the information about which phase is dominant is stored in the global variable IEQU, see Code 4.2.

Code 4.2 Determination of the dominant phase

```
IF (AV(I).LE.0.98D0) IEQU(I)=0                                    1
IF (AV(I).GT.0.98D0) IEQU(I)=1                                    2
```

AV is the void fraction $\alpha = \frac{V_{vap}}{V_{\mathrm{tot}}}$ and determined by the quantities that are calculated within the Müller iteration. As routine DZUK is called during the SSC within the iterations, note that it is possible that the dominant phase in a CV changes during an outer iteration loop.

Execution of Newton's method

For Newton's method, thermal and mechanical non-equilibrium are assumed. The goal of Newton's method is to determine the mass quality x_m and enthalpy of the dominant phase h_{dom} such that the specific total energy in the CV meets the target value e_{tar} (= `HPKI`) even for thermal and mechanical non-equilibrium. Accordingly, a system of two equations (one equation for enthalpy and one for mass quality in the CV) is solved. The initial values of the iteration are the values of thermal equilibrium.

- For enthalpy:
  - $h_{\mathrm{dom}} = h'$, if liquid is phase dominant;
  - $h_{\mathrm{dom}} = h''$, if vapor is phase dominant.
- For mass quality: x_m equals the root obtained by Müller's method.

The valid ranges for h_{dom} and x_m are given as

- h_{dom}: fluid-specific constant values, see subroutine `ALLOCMPR1` for details;
- x_m: $0 \leqslant x_m \leqslant 0.99$.

The system of equations is included in routine `DENTX1`, which is called by `DENTNE` via `SNEWTN`. In the first equation of the system, i. e. (4.16a), the routine `DFCTXM` is called analogously to the procedure of Müller's method to calculate the slip ratio. The slip ratio is then used to determine the mass flow quality, which has an effect on the specific total energy, see (4.11). As no thermal equilibrium is assumed for the application of Newton's method, (4.11) cannot be simplified to (4.12). Again, the target value is the specific total energy e_{tar} (= `HPKI`) that is prescribed in `DENTH`. Variable quantities are the mass quality x_m and the enthalpy of the dominant phase h_{dom}, which is either h_{vap} (then $h_{\mathrm{liq}} = h'$) or h_{liq} (then $h_{\mathrm{vap}} = h''$).
In the second equation of the system, mass conservation is considered. For this purpose, the routine `DFKSME` is called, which takes into account convective mass flows into or out of a control volume. Furthermore, evaporation and condensation rates are calculated (thermal non-equilibrium; application of the models of Sideman or Plesset & Zwick) within this routine. The basic idea is:

- $\dot{x}_{m,SR}$ can be expressed as a function of phase mass flows and evaporation or condensation rate.
- Phase mass flows and evaporation or condensation rates are depending on x_m, h_{liq}, and h_{vap} and consequently on x_m and h_{dom}.

For Newton's method the roots of the following equations are searched (more information on the specific terms can be found in Appendix B):

$$f_N\left(x_m, h_{\mathrm{dom}}\right) = \frac{e_{\mathrm{tar}} - e_{\mathrm{cur}}}{\Delta h_v\left(p\right)} \tag{4.16a}$$

$$g_N\left(x_m, h_{\mathrm{dom}}\right) = \dot{x}_{m,SR} - x_{m,\mathrm{MB}} \tag{4.16b}$$

The convergence criteria are described in the header of `SNEWTN`:

Code 4.3 Description of the convergence criteria

```
!  CONVERGENCE CRITERIA:                                                  1
!  1) THE DIFFERENCE BETWEEN THE APPROXIMATIONS X(I,J) OF THE J-TH        2
!     STEP AND X(I,J-1) OF THE (J-1)-ST STEP IS NOT LARGER (IN            3
!     ABSOLUTE VALUE) THAN MAX(/X(I,J-1)/ * EPSN,CLMN(I)) FOR ALL         4
!     I=1...N.                                                            5
!  2) NONE OF THE RESIDUALS Y(I) IS GREATER (IN ABSOLUTE VALUE)           6
!     THAN FLMN(I)                                                        7
```

Both convergence criteria must be met. The hard-coded criteria (defined in `DENTNE`) are as follows:

- For the variable quantities:
  - `EPSN` = 10^{-7}
  - h_{dom}: `CLMN(1)` = 1
  - x_m: `CLMN(2)` = 10^{-7}
- For the function values:
  - f_N: `FLMN(1)` = 1
  - g_N: `FLMN(2)` = 10^{-5}

Miscellaneous

- If a state in a CV (characterized by x_m, h_{dom}, p and `IEQU`) is determined by Newton's method, this state has an impact on the state of the downstream CV. For that reason, iterations for enthalpy, mass quality and pressure are performed within a priority chain CV by CV.
- Mixture level CVs are split into one sub-CV above the mixture level and one below. For both sub-CVs `SNEWTN` is not called, but a modified regula falsi iteration is performed.

4.4.2 Pressure iteration for TFOs with flowing fluid

The routine DGGT0 initializes the CV of a priority chain (PC) with physically meaningful values for the pressure. The pressure p_0 in the first CV of a PC that is set by the user is not modified. It serves as an initial value, and for closed loops also as a target value, of the pressure iterations. Two pressure iterations must be distinguished:

- *Inner iteration* on junction / CV level: Iterative determination of the pressure in every CV separately.
- *Outer iteration* on PC level: Iterative determination of the pressure in all CVs of a PC. Note that in the current context the term *outer iteration* does not refer to the outer iteration loop described in section Section 4.3.1.

In the inner iteration, the pressure in the subsequent CV is determined successively, starting from the fixed preset pressure in the first CV. In Fig. 4.7 the pressure is known for the orange CV. The pressure in the green CV is calculated from the pressure in the orange CV, the pressure increase caused by the pump, and pressure loss Δp over the junction. The pressure loss over the junction depends on material properties (e. g. density) which in turn are usually temperature and pressure dependent. Regarding the solution variables of the 4-equation model this means that the material properties are depending on x_m, h_{dom} and p. Both, x_m and h_{dom} are already calculated in DENTH and DENTNE before the call of DGGT0. The inner iteration solves the problem that $p = f\Big(\ldots, \Delta p(p)\Big)$.

The outer iteration is only necessary for closed loops. In this iteration it comes to modifications of the form loss coefficients ζ.

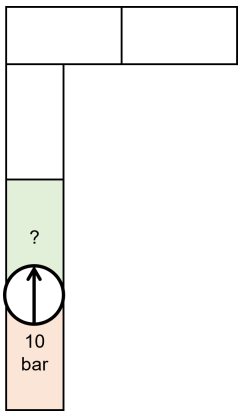

Fig. 4.7 Pressure iteration sketch. The pressure in the green CV is calculated from the pressure of the orange CV, the pressure increase caused by the pump, and the pressure losses over the junction.

DGGTO is called in DSTAR for CVs with flow after DANF and DENTNE, see Fig. 4.8. In the figure the outer iteration on PC level that is controlled by the iteration variable IANZ10 is shown. The test whether the pressure iteration was successful for the last junction of a closed PC also refers to this iteration.

A flow chart of DGGTO is displayed in Fig. 4.9. The flow chart contains the variable IANZ, which is initialized in DANF and used in DGGTO. IANZ is the iteration variable of the inner iteration. Furthermore, compared to the simplified representation in Fig. 4.6, one recognizes that DENTNE is only called for TFOs with input parameter ICKO $\neq 0$ (input parameter ICKO $= 0$ corresponds to single-phase liquid water). This seems to make sense insofar as for ICKO $= 0$ it is always true that $x_m = 0$. Consequently, a change of x_m is not necessary in that case. Interestingly, DENTNE is called for ICKO $= 3$ (= only steam/vapor), where a change of x_m does not occur as well.

4.4.2.1 Inner iteration

Within the IANZ loop the pressure p_j in the target CV is determined. It depends on the pressure in the upstream CV, i. e. p_{j-1}, and various terms that result in pressure decrease or increase ($\Delta p_{...}$):

$$p_j = p_{j-1} + (\Delta p_{\mathrm{MomFlux}} + \Delta p_{v_{\mathrm{rel}}} + \Delta p_{\mathrm{Fric}} + \Delta p_{\mathrm{grav}} + \Delta p_{\mathrm{Pump}}) \cdot \mathtt{PITEJ}_j. \qquad (4.17)$$

The parameter $\mathtt{PITEJ}_j$ specifies for every junction whether the TFO is went through from left to right ($\mathtt{PITEJ}_j = 1.0$) or from right to left ($\mathtt{PITEJ}_j = -1.0$). The various terms that result in pressure decrease or increase are depending on material properties. The material properties depend on the following three solution variables of the SSC: XQM, HDOM and PRESS. While h_{dom} and p define the material properties of the single phases, x_m is used to calculate the material properties of the mixture. It generally applies that $p = f(x_m, h_{\mathrm{dom}}, p)$. The variables x_m and h_{dom} are calculated in DENTH and DENTNE. Within DGGTO the pressure PZ is iterated.

The inner iteration is a fixed-point iteration. The previous implementation of the iteration erroneously used an old pressure value for the calculation of the material properties, which led to convergence after the second iteration step. The implementation was therefore revised and corrected as part of this project. However, this had only a very minor effect on the solution (quasi-identical results).

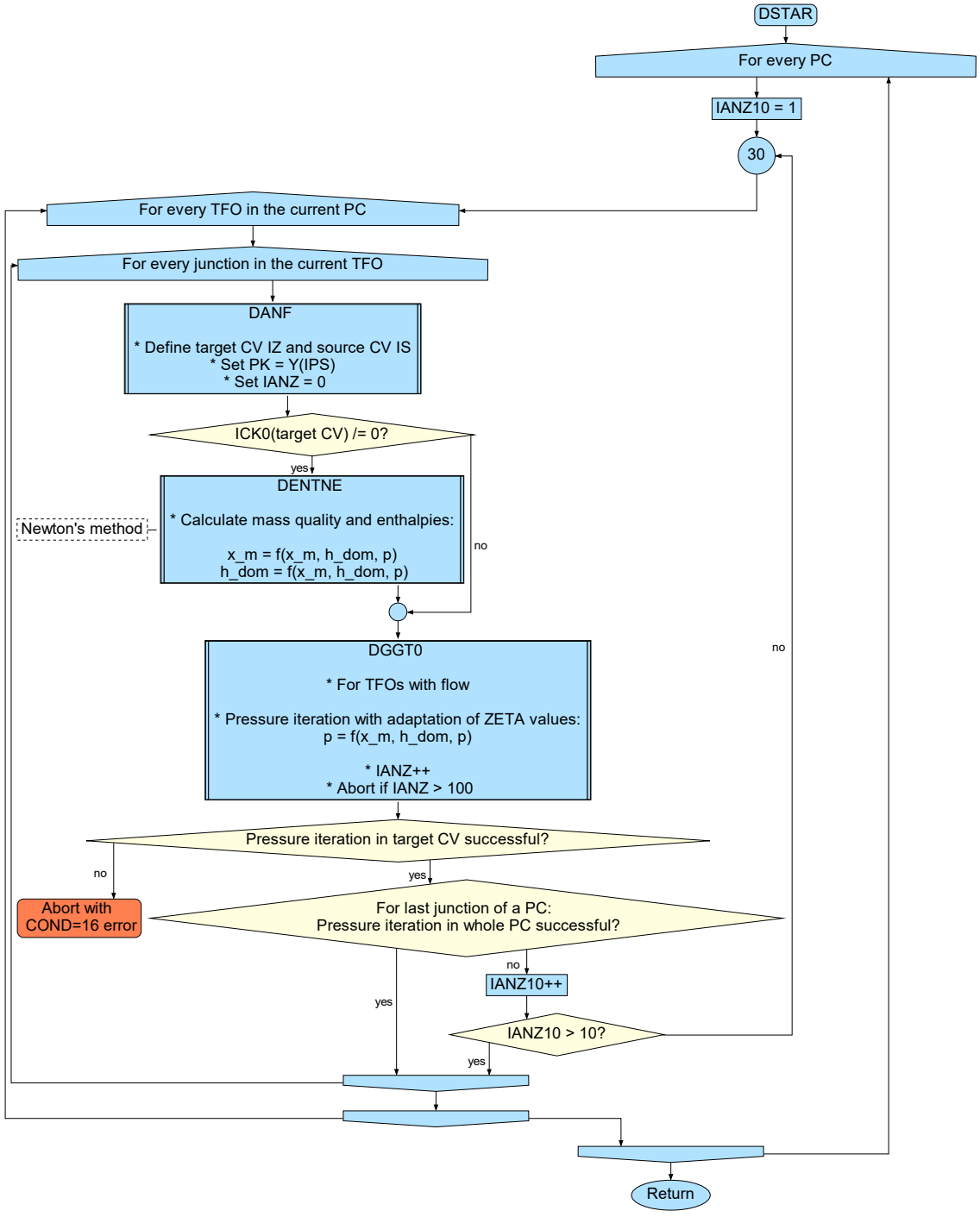

Fig. 4.8 Detailed flow chart of the SSC in DSTAR

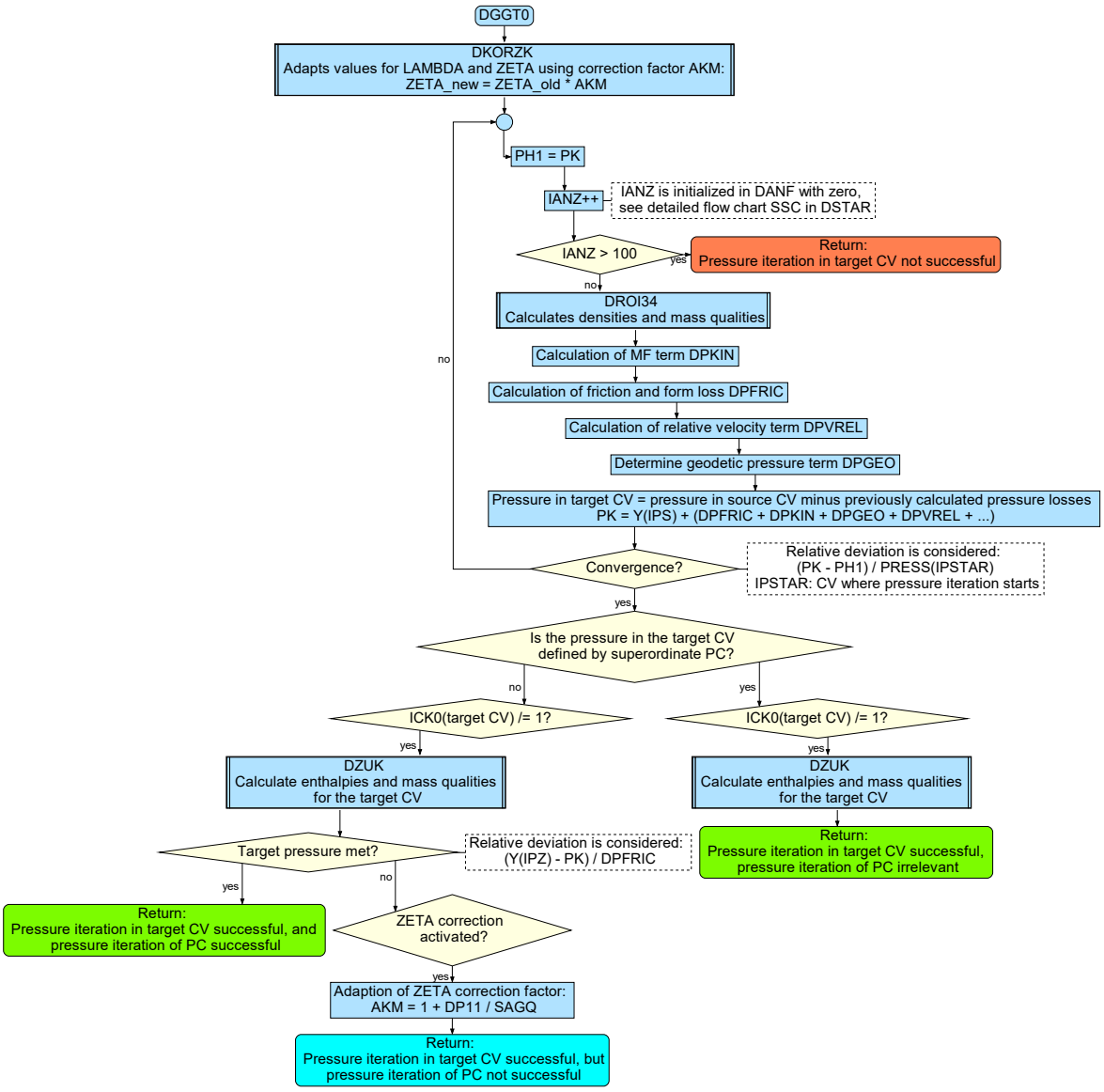

Fig. 4.9 Simplified flow chart of the SSC in `DGGT0`

4.4.2.2 Outer iteration

The outer iteration is done to avoid a jump of pressure from the last to the first CV in case of a closed loop. Therefore, the ζ values are modified. This is done in `DGGTO`, see Fig. 4.9. If the changes are too significant compared to the previous iteration step of the outer iteration, `DSTAR` takes care of continuing the iteration with the next step, see Fig. 4.8.

The variable quantity in the code is the correction factor `AKM`, which is used to modify the ζ values applied in the loop (subroutine `DKORZK`):

```
ZFFJ(JGV) = ZFFJ(JGV)*AKM
```

The SSC starts with the user input of the ζ values ($\texttt{AKM} = 1$).

Given a closed PC, the pressure in the start CV is to be met after the calculation:

$$\underbrace{p_0}_{\substack{\text{pressure}\\ \text{in start CV}}} - \underbrace{\left(p_0 + \sum_j^n \Delta p_{\mathrm{tot},j}\right)}_{\substack{\text{pressure in start CV}\\ \text{after calculation of closed PC}}} = -\sum_j^n \Delta p_{\mathrm{tot},j} \stackrel{!}{=} 0 \tag{4.18}$$

Here $\Delta p_{\mathrm{tot},j}$ denotes the sum of all terms of junction j that result in a pressure increase (e. g. by a pump) or decrease (e. g. by form losses). Consequently, the sum considers the pressure losses and increases over all n junctions of the closed PC. The friction and form losses (combined to $\Delta p_{\mathrm{Fric},j}$) can be separated from the other terms (here called $\Delta p_{\mathrm{Rest},j}$):

$$\begin{aligned}\sum_j^n \Delta p_{\mathrm{tot},j} &= \sum_j^n (\Delta p_{\mathrm{MomFlux},j} + \Delta p_{\mathrm{Fric},j} \\ &\qquad + \Delta p_{v_{\mathrm{rel}},j} + \Delta p_{\mathrm{geo},j} + \Delta p_{\mathrm{Pump},j} + \dots) \\ &= \sum_j^n \Delta p_{\mathrm{Fric},j} + \sum_j^n \Delta p_{\mathrm{Rest},j}.\end{aligned} \tag{4.19}$$

For the sake of clarity, a simplified notation is used in which the summation symbols are omitted:

$$\begin{aligned}\Delta p_{\mathrm{Fric}} &:= \sum_j^n \Delta p_{\mathrm{Fric},j}, \\ \Delta p_{\mathrm{tot}} &:= \sum_j^n \Delta p_{\mathrm{tot},j}.\end{aligned} \tag{4.20}$$

In order to fulfill (4.18), a common correction factor α for the pressure loss due to friction is applied. It can be interpreted as a correction to the ζ values since

$$\alpha \cdot \Delta p_{\mathrm{Fric}} = \sum_j^n \frac{\rho_j}{2} v_j^2 (\zeta_j \cdot \alpha). \tag{4.21}$$

The correction factor α appears linear in (4.18). Hence, with an initial value of $\alpha^{\langle 0\rangle} = 1$ this leads to the iteration

$$\alpha^{\langle i+1\rangle} = \alpha^{\langle i\rangle} - \frac{\Delta p_{\mathrm{tot}}(\alpha^{\langle i\rangle})}{\Delta p_{\mathrm{Fric}}}. \tag{4.22}$$

In (4.22) the indices i and $i+1$ representing the iteration steps are for clarity written as superscripts in angle brackets. Thus, if $\alpha^{\langle i\rangle}$ denotes the old correction factor for the ζ_j, then $\alpha^{\langle i+1\rangle}$ is the new correction factor at iteration step $i+1$. In the current implementation of the SSC, the correction is determined in terms of a factor β:

$$\alpha^{\langle i+1\rangle} = \alpha^{\langle i\rangle} \cdot \beta^{\langle i+1\rangle}. \tag{4.23}$$

Hence, the overall correction factor at the end of the SSC $\alpha^{\langle i_{\mathrm{end}}\rangle}$ reads as

$$\alpha^{\langle i_{\mathrm{end}}\rangle} = \prod_{i=1}^{i_{\mathrm{end}}} \beta^{\langle i\rangle}. \tag{4.24}$$

Substituting (4.23) into (4.22) yields:

$$\begin{aligned} \beta^{\langle i+1\rangle} &= 1 - \frac{\Delta p_{\mathrm{tot}}(\alpha^{\langle i\rangle})}{\Delta p_{\mathrm{Fric}} \cdot \alpha^{\langle i\rangle}} \\ &=: 1 - \frac{\Delta p_{\mathrm{tot}}^{\langle i\rangle}}{\Delta p_{\mathrm{Fric}}^{\langle i\rangle}} \end{aligned} \tag{4.25}$$

The assignment (4.25) is the iteration procedure for the ζ correction factor as applied in the SSC. In the source code this can be found in the routine `DGGT0`:

Code 4.4 Iteration procedure of the ζ correction factors

```
DP11    = Y(IPZ)-PK                                                    1
...                                                                    2
AKM     = 1.D0 + DP11/SAGQ                                             3
```

`DP11` is the difference between the specified and calculated pressure in the first CV and corresponds to the sum of all pressure increases and losses over all junctions along the PC. `SAGQ` is the sum of friction and form losses over all junctions along the PC. `AKM` corresponds to β in (4.25). The product used in (4.23) and (4.24) can be found in the code in `DKORZK`:

Code 4.5 Correction of the ζ values

```
SDFJ (JGV) = SDFJ (JGV)*AKM                                            1
ZFFJ (JGV) = ZFFJ(JGV)*AKM                                             2
ZFBJ (JGV) = ZFBJ(JGV)*AKM                                             3
```

The convergence criterion is $\Delta p_{\mathrm{tot}}^{\langle i\rangle} / \Delta p_{\mathrm{Fric}}^{\langle i\rangle} < 10^{-5}$, which means that β is sufficiently close to one.

4.4.3 Pressure iteration for TFOs with stagnant fluid

The routine DGEQ0 initializes the CVs of a priority chain with stagnant fluid with physically meaningful values for the pressure. The pressure in the first CV of a PC that is set by the user is not modified. It serves as an initial value, and for closed loops also as a target value, of the pressure iterations. Contrary to routine DGGT0, only an inner iteration must be performed for routine DGEQ0. This iteration is used to iteratively determine the pressure in each CV separately.

In the inner iteration, the pressure in the subsequent CV is determined successively, starting from the fixed preset pressure in the first CV. The procedure of the inner iteration is analog to DGGT0, see Section 4.4.2. However, for a TFO with stagnant fluid only pressure changes due to hydrostatics and pressure increases caused by pumps must be considered. The pressure changes caused by hydrostatics are depending on the fluid density and thus on material properties. The fluid density is temperature and pressure dependent and consequently depends on the following solution variables of the 4-equation model: x_m, h_{dom} and p. Both, x_m and h_{dom} are already calculated in DENTH and DENTNE before the call of DGEQ0. The inner iteration solves the problem that $p = f\Big(\ldots, \Delta p(p)\Big)$.

For TFOs with stagnant fluid no outer iteration is considered. Thus, for closed loops neither geodetic heights nor pump pressure heads are adapted for stagnant fluid. For that reason, the existence of an activated pump (without mass flow) results in an inconsistency of pressure between first and last CV of a closed loop. This will be discussed in more detail at the end of this section.

A flow chart of DGEQ0 is shown in Fig. 4.10. The routine is called in DSTAR for CVs with stagnant fluid after DANF and DENTNE, see Fig. 4.6.

4.4.3.1 Inner iteration

Within the inner iteration the pressure p_j in the target CV is determined. It depends on the pressure in the upstream CV, i. e. p_{j-1}, the hydrostatic pressure Δp_{grav} and (if existent) the pump head Δp_{Pump}:

$$p_j = p_{j-1} + (\Delta p_{\mathrm{grav}} + \Delta p_{\mathrm{Pump}}) \cdot \mathtt{PITEJ}_j. \tag{4.26}$$

The various terms resulting in pressure decrease or increase depend on material properties. These depend on the following three solution variables of the SSC: x_m, h_{dom} and p (in ATHLET: XQM, HDOM, PRESS). While h_{dom} and p define the material

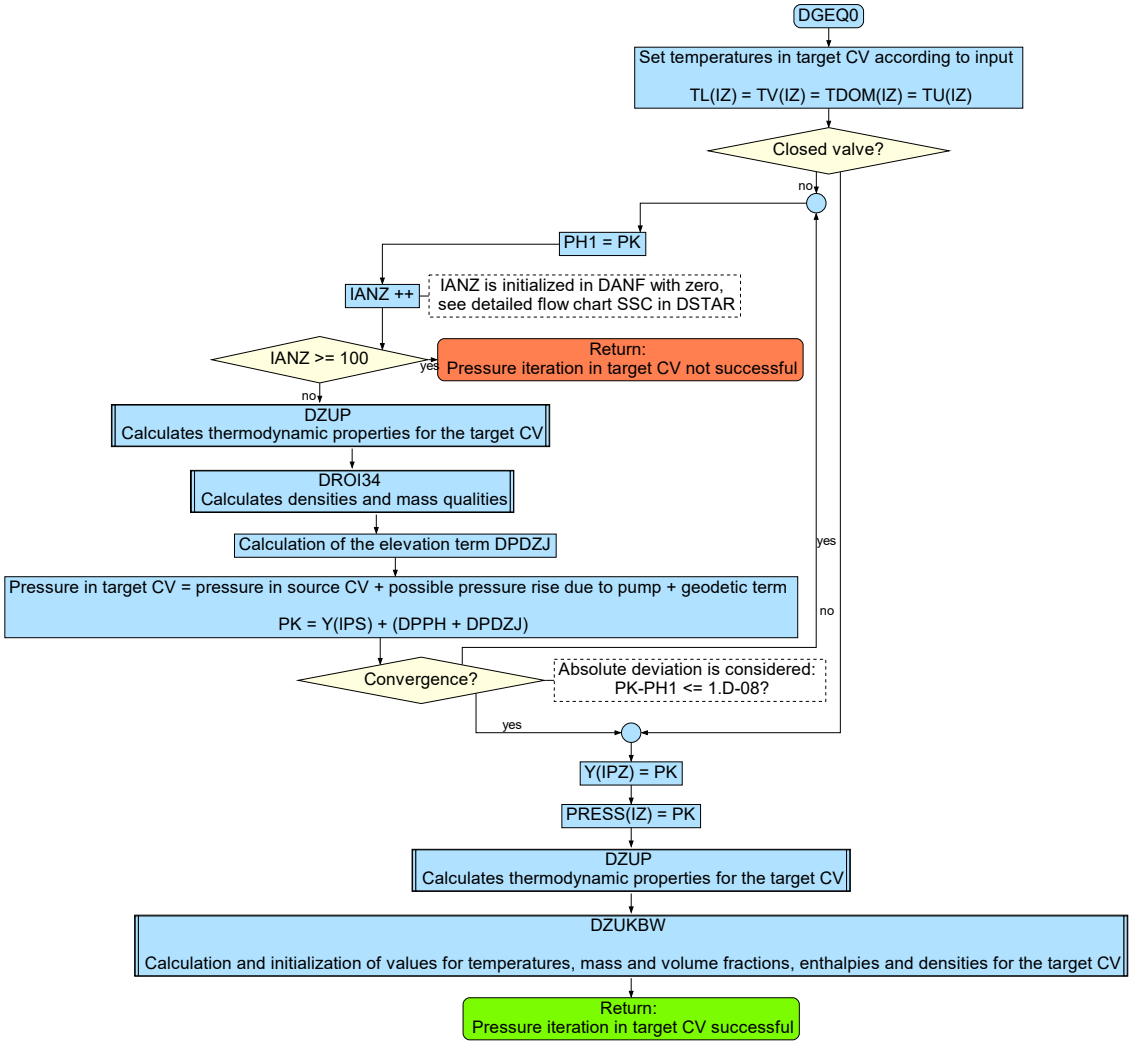

Fig. 4.10 Flow chart of the SSC in `DGEQ0`

properties of the single phases, x_m is used to calculate the material properties of the mixture. It generally applies that $p = f\left(x_m, h_{\mathrm{dom}}, p\right)$. The variables x_m and h_{dom} are calculated in `DENTH` and `DENTNE`. The pressure is iterated in `DGEQ0`.

The inner iteration is a fixed-point iteration. As convergence criterion the absolute deviation of the pressure of the current iteration to the previous one is used. The fixed-point iteration is converged if the absolute deviation is smaller than 10^{-8}.

4.4.3.2 Inconsistency for closed loops

As can be seen on the left side of Fig. 4.11 the calculated pressures of the SSC are meaningful for a closed loop without a pump. For that case only the geodetic height affects the pressure, which results in consistent pressures at the start and end of the closed loop.

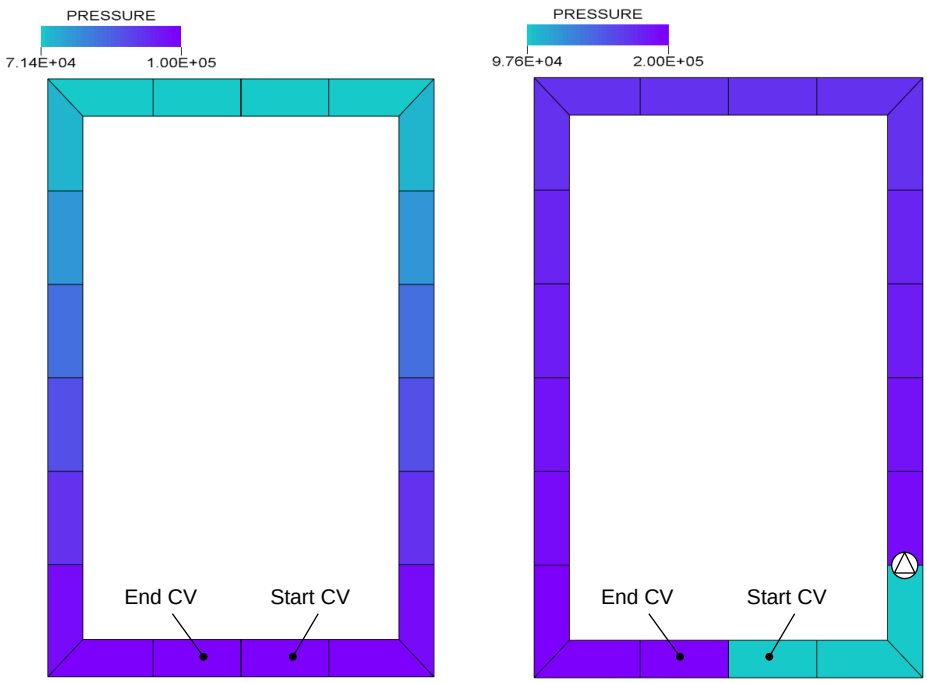

Fig. 4.11 Closed loop of a system with stagnant fluid without (left) and with (right) a pump

If an activated pump is added to the closed loop, it causes a pressure increase. As for TFOs with stagnant fluid, no further pressure losses occur and geodetic heights do not have a "compensatory" effect for closed loops. A jump of pressure can be observed between start and end of the closed loop for this case as can be seen from see right side of Fig. 4.11.

The described inconsistency does not seem to be problematic for realistic applications. Nevertheless, it would be desirable to add a check that catches this inconsistency.

4.4.4 Iteration of the pump rotational speed

For pumps that are in operation, the pump input data together with the hydraulic data coming from the SSC itself lead to an over-determined system. The pump operating point is completely defined by the homologous head curve, the nominal values of the pump, and the steady-state pump rotational speed and pressure. The SSC calculates the local density of the liquid, which together with the steady-state mass flow yields the steady-state flow rate. All these quantities together will most likely not match to a sufficient degree to ensure a true steady state. For this reason, the pump speed is automatically adjusted by the SSC so that the pump pressure corresponds to the value from the input `DPPS`.

If a pump model is selected in which variations of rotational speed play a role, KPUFCT is called from Newton's method SNEWTN in KPUSS, see Fig. 4.12. KPUSS is called by the SSC (ITRANS = 0) in DKPUMP, see Fig. 4.13. DKPUMP is called in DSTART in a loop over all objects.

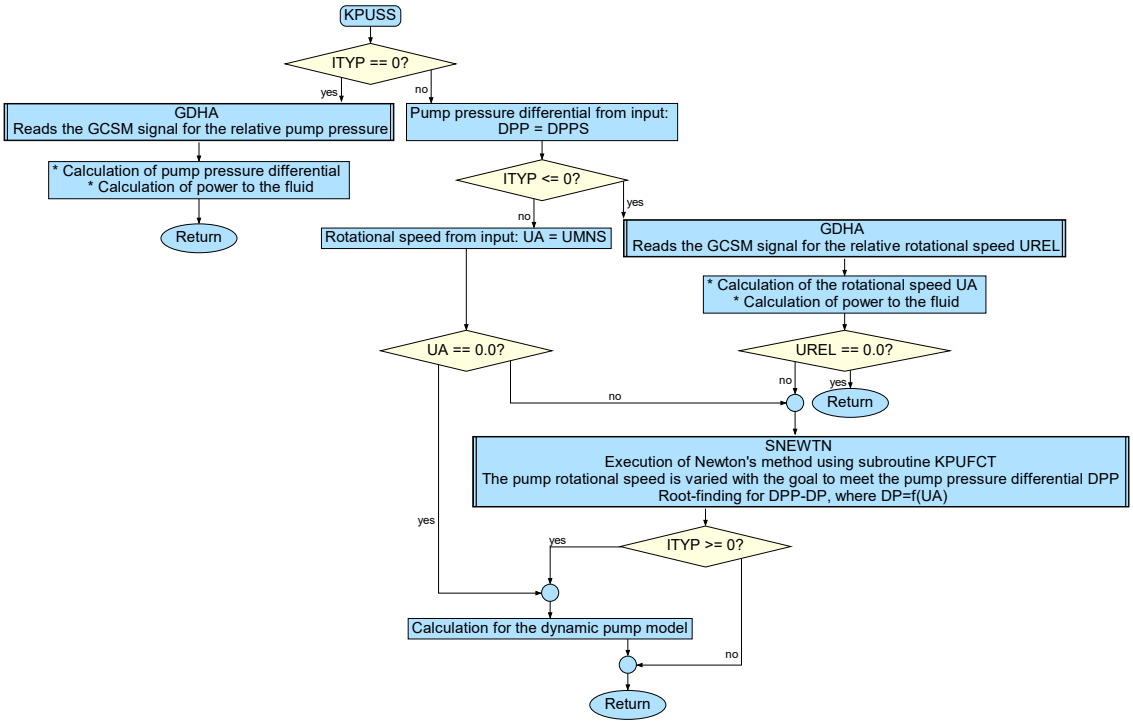

Fig. 4.12 Flow chart of the SSC in KPUSS

Variation of rotational speed with Newton's method

For the simplest pump model (differential pressure control), no speed variation needs to be performed. For the other pump models, the following must be considered: To ensure that the pump pressure in the SSC corresponds to the value from the input DPPS, the speed is varied using Newton's method. Newton's method is executed in subroutine SNEWTN for the function defined in KPUFCT. According to /SCH 23a/ the correlations for the homologous head curve, using the dimensionless variables pump speed ratio $\alpha = n/n_R$, pump volumetric flow ratio $v = Q/Q_R$ and pump head ratio $h = H/H_R$ (the subscript R indicates rated values), are for the case $\alpha \neq 0$ and $|v/\alpha| < 1$:

$$\begin{aligned} \frac{h}{\alpha^2} &= \text{const.} \\ \frac{v}{\alpha} &= \text{const.} \end{aligned} \tag{4.27}$$

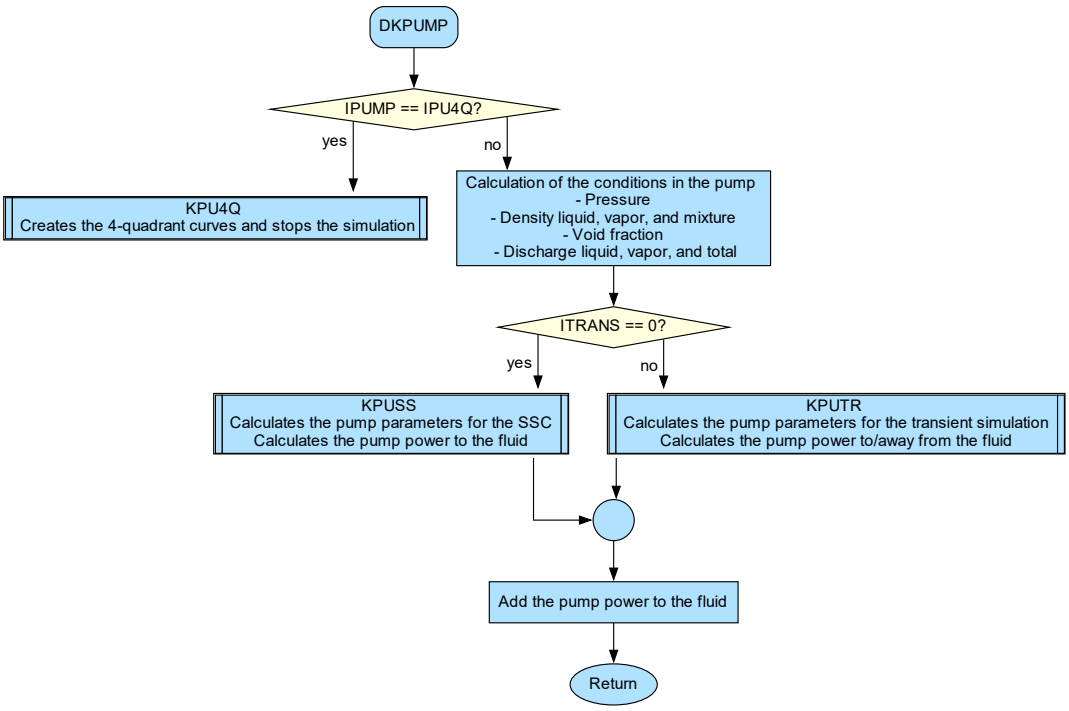

Fig. 4.13 Flow chart of the SSC in DKPUMP

and else

$$\begin{aligned} \frac{h}{v^2} &= \text{const.} \\ \frac{\alpha}{v} &= \text{const.} \end{aligned} \tag{4.28}$$

This is used to calculate the pump head HDNP and afterwards the pump pressure DP. By adjusting the speed, the deviation between the target value for DPP, DPPS, and the current value DP is minimized using Newton's method.

Miscellaneous

The simulation is aborted at the end of the SSC if the iterated value deviates more than 10 % from the value that is defined by the rotational speed from input UMNS and the GCSM signal SGPUMP.

4.4.5 Iteration of layer temperatures

The iteration of the temperature of HCO layers is performed within routine HCSTA which is called in the SSC by routine HECU, see Fig. 4.2 and Fig. 4.14.[1] Basically, this routine is used in the SSC to calculate heat transfer and heat conduction. At the end

[1] In ATHLET 3.4, HCSTA is called by its own wrapper routine HECUSSC with identical functionality.

of the SSC, this gives the initialization of the HCV with physically reasonable layer temperatures as well as the heat flows between CV and HCV. Furthermore, heat sources, rods and heat exchangers are considered or calculated within this routine. Two different iterations are performed within HECU in the scope of the SSC:

- Inner iteration over all layers of an HCV for iterative calculation of layer temperatures in a HCV
- Outer iteration over entire system to meet the user-defined convergence criteria, see Section 4.3.1

For the inner iteration, the fluid temperatures are used as boundary conditions to determine the heat transfer. An iterative procedure for the calculation of the layer temperatures is necessary, as these depend on material properties, such as the thermal conductivity λ, which in turn is temperature dependent: $T\Big(\ldots,\lambda(T)\Big)$. Since there is an over-determined system of equations due to the user inputs, the heat exchanger surface is adjusted as part of the inner iteration. However, since the modification is only performed in the first inner iteration, it actually takes place in the outer iteration loop.

The outer iteration is performed until the user-defined convergence criteria are met, with a maximum of 15 iterations being performed. During the outer iteration, all ATHLET modules (TFD, HECU, NEUKIN, GCSM) are called again because the TFD and HECU calculations influence each other.

4.4.5.1 Inner iteration

The calculation of the layer temperatures in the inner iteration loop is performed successively for each HCV. As the calculation of the layer temperatures requires the thermal conductivity or the heat transfer coefficient at the layer boundaries, in a first step an iteration loop is performed over all layers to calculate the heat transfer coefficient and the thermal conductivity. The calculation of the layer temperatures is then performed in the actual inner iteration loop.

Calculation of layer temperatures

For the calculation of the layer temperatures T a fixed-point iteration is used. An iterative procedure is necessary because the layer temperatures depend on the thermal conductivity, but the thermal conductivity in turn depends on the layer temperatures:

$$T = f(\lambda(T)). \tag{4.29}$$

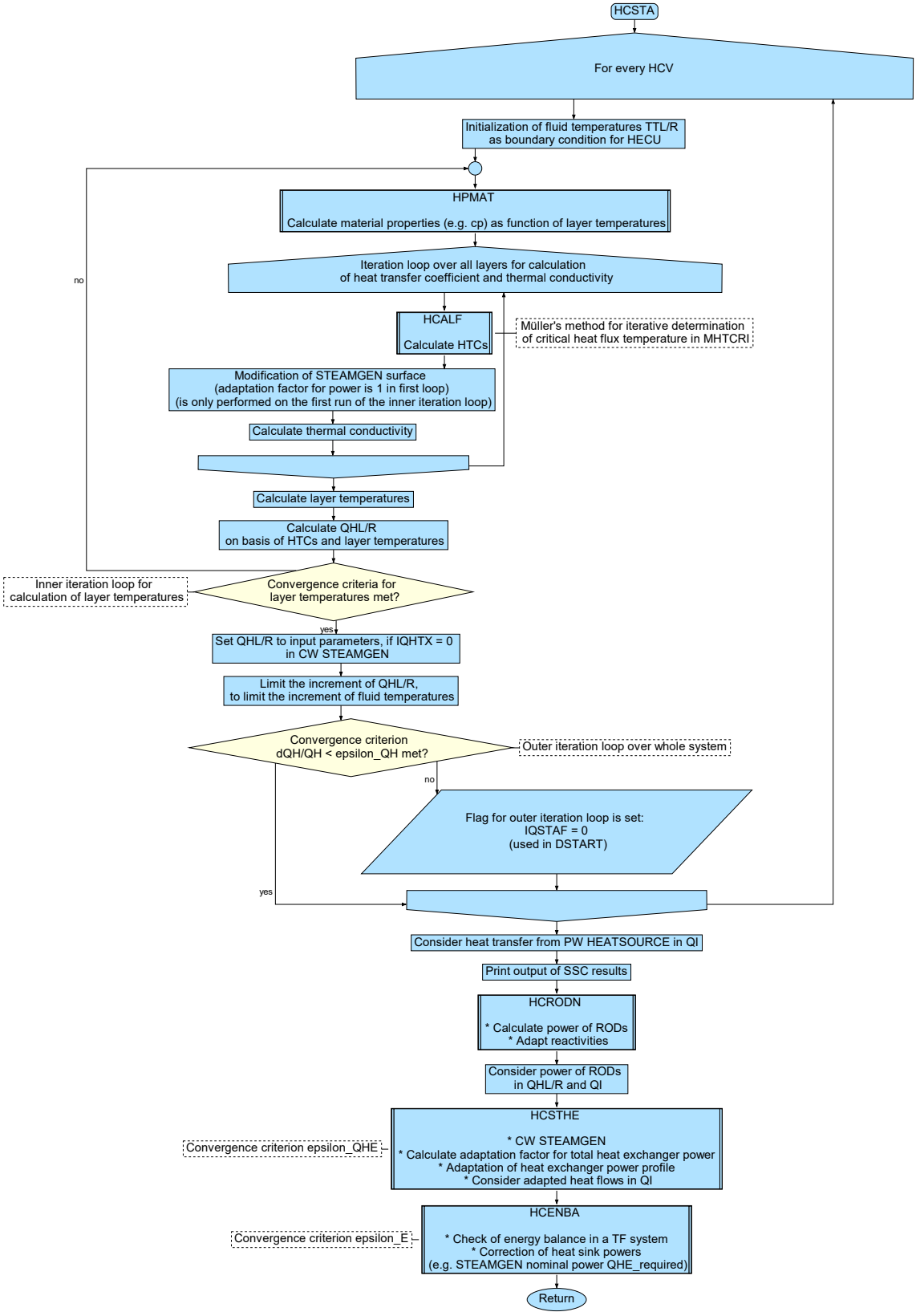

Fig. 4.14 Details of the SSC in `HCSTA`

In the numerical treatment, the thermal conductivity is calculated using the layer temperatures from the previous iteration i. The layer temperature for the new iteration $i+1$ is calculated as follows:

$$T^{\langle i+1\rangle} = f(T^{\langle i\rangle}) \tag{4.30}$$

It is important to notice that $T^{\langle i\rangle}$ represents the considered layer as well as the neighboring layers. The initialization (value $T^{\langle 0\rangle}$) of the layer temperatures is performed within the routine `HCTINI`. If the left or right side is adiabatic, the layer temperatures are initialized with `TTR` or `TTL`, respectively. In case that none of the sides is adiabatic, a linear distribution from `TTL` to `TTR` is used.

For the (general) transient case the layer temperatures are calculated as

$$\frac{dT_j}{dt} = \frac{1}{\rho_j c_{p,j} V_j} \cdot \left[\frac{1}{R_j} \cdot T_{j-1} - \left(\frac{1}{R_j} + \frac{1}{R_{j+1}} \right) \cdot T_j + \frac{1}{R_{j+1}} \cdot T_{j+1} + W_j \cdot V_j \right]. \tag{4.31}$$

R is the heat transfer resistance, W is the specific volumetric heat generation and j represents the current layer. The indices $j-1$ and $j+1$ stand for the neighboring layers. For the SSC, the time derivative vanishes. Therefore, (4.31) can be simplified to give the expression

$$A_j^{\langle i\rangle} T_{j-1}^{\langle i+1\rangle} + B_j^{\langle i\rangle} T_j^{\langle i+1\rangle} + C_j^{\langle i\rangle} T_{j+1}^{\langle i+1\rangle} = D_j^{\langle i\rangle}. \tag{4.32}$$

The superscript i is used to show that for the calculation of the coefficients, layer temperatures from the previous iteration are used (which for simplicity is omitted in the following). For cases with multiple layers the coefficients in (4.32) are calculated as follows:

$$\begin{aligned} A_j &= 1/R_j \\ B_j &= -(1/R_j + 1/R_{j+1}) \\ C_j &= 1/R_{j+1} \\ D_j &= -W_j V_j \end{aligned} \tag{4.33}$$

For the first layer ($j = 1$) and the last layer ($j = N$), the following must be used:

$$\begin{aligned} A_1 &= 0 \\ C_N &= 0 \end{aligned} \tag{4.34}$$

For the case that only one layer is present the coefficients are calculated as follows:

$$\begin{aligned} A_j &= 0 \\ B_j &= -(1/R_1 + 1/R_2) \\ C_j &= 0 \\ D_j &= -W_1 V_1 \end{aligned} \tag{4.35}$$

In this case, no system of equations must be solved and the layer temperature is calculated as follows:

$$T_1^{\langle i+1\rangle} = D_j / B_j. \tag{4.36}$$

As the layer temperatures of neighboring layers influence each other, a system of equations has to be solved if several layers are present. It has the form of a tridiagonal matrix:

$$\begin{pmatrix} B_1 & C_1 & & & 0 \\ A_2 & B_2 & C_2 & & \\ & \ddots & \ddots & \ddots & \\ & & A_{N-1} & B_{N-1} & C_{N-1} \\ 0 & & & A_N & B_N \end{pmatrix} \begin{pmatrix} T_1^{\langle i+1\rangle} \\ T_2^{\langle i+1\rangle} \\ \vdots \\ T_{N-1}^{\langle i+1\rangle} \\ T_N^{\langle i+1\rangle} \end{pmatrix} = \begin{pmatrix} D_1 \\ D_2 \\ \vdots \\ D_{N-1} \\ D_N \end{pmatrix}. \tag{4.37}$$

Rearranging (4.37), the relation in (4.30) can be written as:

$$\begin{pmatrix} T_1^{\langle i+1\rangle} \\ T_2^{\langle i+1\rangle} \\ \vdots \\ T_{N-1}^{\langle i+1\rangle} \\ T_N^{\langle i+1\rangle} \end{pmatrix} = A(T^{\langle i\rangle})^{-1} D = \begin{pmatrix} B_1 & C_1 & & & 0 \\ A_2 & B_2 & C_2 & & \\ & \ddots & \ddots & \ddots & \\ & & A_{N-1} & B_{N-1} & C_{N-1} \\ 0 & & & A_N & B_N \end{pmatrix}^{-1} \begin{pmatrix} D_1 \\ D_2 \\ \vdots \\ D_{N-1} \\ D_N \end{pmatrix}. \tag{4.38}$$

Method for solving the system of equations

The linear system (4.37) is solved using a tridiagonal matrix algorithm, also known as Thomas algorithm. It consists of a forward sweep and a backward substitution. In the forward sweep, the coefficients are modified recursively:

$$\begin{aligned} F_j &= \begin{cases} \frac{C_1}{B_1} & \text{if} \quad j = 1 \\ \frac{C_j}{B_j - A_j F_{j-1}} & \text{if} \quad 2 \leqslant j < N \end{cases} \\ G_j &= \begin{cases} \frac{D_1}{B_1} & \text{if} \quad j = 1 \\ \frac{D_j - A_j G_{j-1}}{B_j - A_j F_{j-1}} & \text{if} \quad 2 \leqslant j \leqslant N \end{cases} \end{aligned} \tag{4.39}$$

The solution of the system of equations is obtained by a backward substitution:

$$\begin{aligned} T_N^{\langle i+1\rangle} &= G_N \\ T_j^{\langle i+1\rangle} &= G_j - F_j T_{j+1}^{\langle i+1\rangle} \end{aligned} \tag{4.40}$$

Code implementation

In the forward sweep, at the beginning, the first layer is calculated, for which the following applies regarding the layer temperature and its time derivatives: $\texttt{TT}(\texttt{LAY1}) = G_j$ and $\texttt{DT}(\texttt{LAY1}) = F_j$.

Code 4.6 Forward sweep for calculation of the layer temperature in first layer

```
X        = 1.D0 / (YL + YR + YYLO + R2WRL)                                  1
DT(1) = -YR*X                                                               2
TT(LAY1) = ( YL*TL + ATT(LAY1)*SV(LAY1) + QOXILR(NHV) + QRADL +             3
    YYLO*TTLO ) * X
```

The other layers are calculated using a forward sweep loop:

Code 4.7 Forward sweep loop for calculation of the layer temperatures (except first layer)

```
DO ILAY = 2,NLAY                                                            1
   IF (ILAY==NLAY) THEN                                                     2
      ! RIGHT END                                                           3
      X        = 1.D0 / ( YL+YR + YYRO+R2WRR + YL*DT(ILAY-1) )             4
      TI       = YR*TR + YYRO*TTRO + QRADR + QOXILR(NHV+IHV)               5
   ELSE                                                                     6
      X        = 1.D0 / ( YL + YR + YL*DT(ILAY-1) )                        7
      TI       = 0.D0                                                      8
      DT(ILAY) = -YR*X                                                     9
   ENDIF                                                                    10
                                                                            11
   TT(LAY1) = ( YL*TT(LAY1-1) + ATT(LAY1)*SV(LAY1) + TI ) * X               12
                                                                            13
   LAY1     = LAY1 + 1                                                      14
   LAY2     = LAY2 + 1                                                      15
   YL       = YR                                                            16
                                                                            17
ENDDO                                                                       18
```

The backward substitution is performed using a reverse loop:

Code 4.8 Backward substitution

```
DO JLAY = 1,NLAYM1                                                          1
   TT(LAY1) = TT(LAY1)-DT(ILAY)*TT(LAY1+1)                                  2
   LAY1     = LAY1 - 1                                                      3
   ILAY     = ILAY - 1                                                      4
ENDDO                                                                       5
```

Note: Additional terms can be found in the code snippets that are not present in the above equations. These terms result from oxidation (QOXILR), thermal radiation (QRADL and QRADR) and the presence of a mixture level (YYLO*TTLO or YYRO*TTRO).

Furthermore, the temperature boundary condition of the TFO is considered (YL*TL or YR*TR). The occurrence of these phenomena makes it necessary to extend (4.31) or (4.32) for the outer layers. This is not described in /SCH 23a/.

Special characteristic for the execution of the fixed-point iteration

The default fixed-point iteration, as performed within the first 150 iterations, follows the lines of calculations as shown above (formally, (4.38) is considered). If convergence has not been achieved by iteration step 150, an additional relaxation (damping) comes into play, namely,

$$T^{\langle i+1\rangle} = T^{\langle i\rangle} + \lambda\left(T^{\langle i+1\rangle} - T^{\langle i\rangle}\right) \qquad \text{with} \quad \lambda = 0.5. \tag{4.41}$$

Code 4.9 Relaxation for calculation of the layer temperatures in case of poor convergence

```
IF(IGL > IGLMA1) THEN                                          1
   LAY1   = LAY1SV                                             2
   DO ILAY = 1,NLAY                                            3
      TT(LAY1)=5.D-1*(TT(LAY1)+TOLD(ILAY))                     4
      LAY1 = LAY1 + 1                                          5
   ENDDO                                                       6
ENDIF                                                          7
```

Convergence criterion

The convergence criterion of the inner iteration is that for every layer in an HCV the difference between the layer temperature of the current iteration $i+1$ and the previous iteration i is smaller than a threshold ε_{TT}:

$$\left|T_j^{\langle i+1\rangle} - T_j^{\langle i\rangle}\right| < \varepsilon_{TT}. \tag{4.42}$$

The initial threshold is $\varepsilon_{TT} = 10^{-5}$. If this value is not met within the first 250 iterations, the threshold is increased in certain steps. For the case that after 500 iterations the then given threshold value of $\varepsilon_{TT} = 10$ is still exceeded, the simulation is aborted with an error message.

4.4.5.2 Outer iteration

The checks whether the user-defined convergence criteria (see Section 4.3.1 for a more detailed description of convergence criteria) are met, are done in the routines HCSTA (4.1a), HCSTHE (4.1b) and HCENBA (4.1c).

The flag IQSTAF is used to control the outer iteration loop. If IQSTAF is set to 0, another outer iteration is performed. As HCSTA is also responsible for the output of the HECU results upon successful convergence or reaching the maximum number of 15 outer iterations, there is another flag JQSTAF. At the beginning of HCSTA, JQSTAF is initialized to 1. In case that at least in one HCV one convergence criterion from (4.1a) is not met, JQSTAF is set to 0. This causes IQSTAF to be set to 0 when convergence has not yet been achieved. This controls the outer iteration loop (in routine DSTART) with respect to the convergence criterion (4.1a). For the other two convergence criteria, IQSTAF is set to 0 directly in the routines HCSTHE or HCENBA. If the convergence criteria are met – that is, if at the end of HCSTA the variables IQSTAF and JQSTAF have the value 1 – or if 15 outer iterations have already been performed, then in DSTART the variable IQSTAF is set to 2 and HCSTA is invoked one last time to generate the output of the HECU calculations.

4.4.5.3 Modification of the heat exchanger surface

Since the user input for heat exchangers result in an over-determined system, the heat exchanger surface is adjusted by default. This ensures that the correct total heat flux of the heat exchanger is transferred. The modification of the heat exchanger surface is controlled by the variable IQHTX. For its default value 1 it is adjusted, while for values 0 or 2 no adjustment is made. If no heat exchanger surface modification is allowed, there is a risk that the energy balance will not be met.

The purpose of adjusting the heat exchanger surface is to ensure that the heat flux $\mathtt{QHE}_{k,\mathrm{req}}$ to be transferred by the heat exchanger with index k matches the calculated heat flux $\mathtt{QHE}_k$ within a user-defined convergence criterion, see (4.1b). At the first outer iteration, the heat exchanger surface adaptation factor FQ1 has the value 1. The calculation of the adaptation factor for the new outer iteration takes place at the end of HCSTA by calling the routine HCSTHE. For the heat exchanger with index k the adaption factor is calculated via

$$\mathtt{FQ1}_k^{\langle i+1\rangle} = \frac{\mathtt{QHE}_{k,\mathrm{req}}^{\langle i+1\rangle}}{\mathtt{QHE}_k^{\langle i+1\rangle}}. \tag{4.43}$$

Note that $\mathtt{QHE}_{k,\mathrm{req}}^{\langle i+1\rangle}$ depends on the outer iteration step $i+1$. Thus, a modification occurs during the SSC in combination with the heat balance of the whole system and the adjustments of heat sinks. This adjustment takes place in the routine HCENBA.

4.4.5.4 Conservation of energy and adaptation of heat sinks

The routine HCENBA has two main purposes:

- Check whether energy is conserved in the TF systems
- Adjustment of heat sinks (heat exchanger, condenser, heat insulation losses)

The check whether the energy is conserved in the TF systems is done by the convergence criterion (4.1c). If the convergence criterion is not satisfied, the heat sinks are adjusted. The correction factor $F_{\mathrm{corr},Q}$ is calculated as follows:

$$F_{\mathrm{corr},Q} = 1 + \frac{\sum \dot{E}_\ell}{Q_{\mathrm{out}}}. \tag{4.44}$$

The energy flux Q_{out} is composed of the energy fluxes of all heat sinks (heat exchangers, condensers and thermal losses). The adjustment of the correction factor for one iteration is limited to the range $0.1 \leqslant F_{\mathrm{corr},Q} \leqslant 10.0$. Finally, all required energy fluxes (all heat sinks) are adjusted via the correction factor. Thus, among other things, the heat flow $\mathtt{QHE}_{k,\mathrm{req}}$, which is to be transferred for the heat exchanger with index k, is adjusted as follows:

$$\mathtt{QHE}_{k,\mathrm{req}}^{\langle i+1\rangle} = F_{\mathrm{corr},Q} \cdot \mathtt{QHE}_{k,\mathrm{req}}^{\langle i\rangle}. \tag{4.45}$$

4.4.5.5 Calculation of heat transfer coefficients

The calculation of the heat transfer coefficients (HTC) is done in the routine HCALF. This routine is called by the SSC as well as by the transient simulation. In the routine, the models for the calculation of the HTC are applied. Regarding the numerical methods of the SSC, it is worth mentioning that in the iterative calculation of the critical heat flux temperature, Müller's method is used within the routine MHTCRI.

4.4.5.6 Calculation of specific heat generation W

The calculation of the specific heat generation W is done by the routine HCRODN. A simplified flow chart of HCRODN is shown in Fig. 4.15. Depending on whether electrical heaters are to be modeled or the heat generation is to take place via neutron kinetics models, different routines are called. For electrical heaters, the routine HRODEL is used. Neutron kinetics is calculated by using the routines NSET and NINTER and their subroutines.

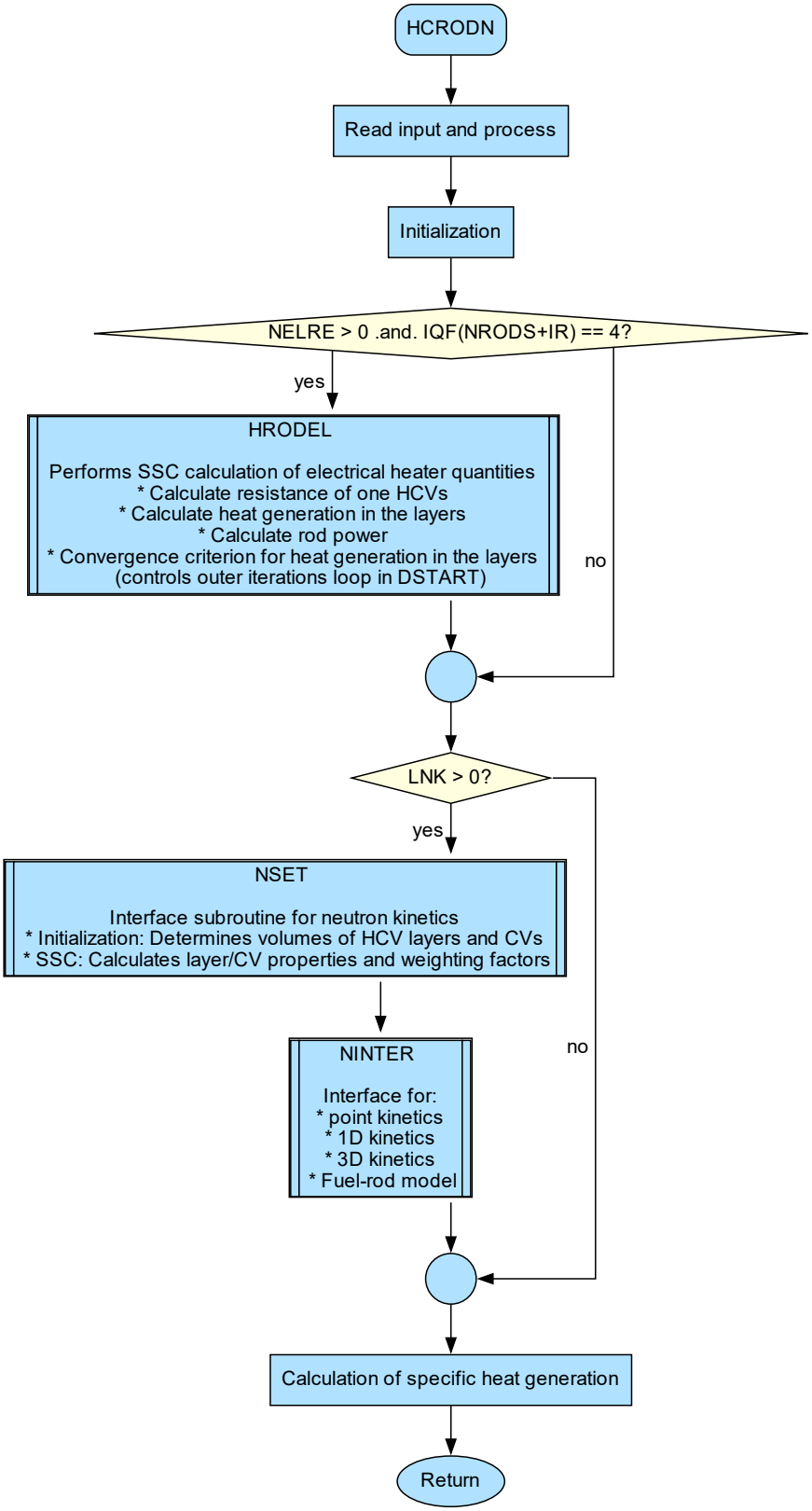

Fig. 4.15 Simplified flow chart of the SSC in HCRODN

Electrical heaters in HRODEL

For electrical heaters, the control of the outer iteration loop is done by the variable IPOWER, which influences the behavior in DSTART in the same way as IQSTAF. IPOWER is initialized to 0 and set to 1 when convergence is reached. The convergence criterion is the following:

$$\max\left(\left|\frac{W^{\langle i+1\rangle}-W^{\langle i\rangle}}{W^{\langle i+1\rangle}}\right|\right)\leqslant\varepsilon_{P_{\mathrm{el}}}. \tag{4.46}$$

By using $\max$, the largest deviation of all layers is considered. For the first 4 outer iterations the threshold is $\varepsilon_{P_{\mathrm{el}}}=10^{-4}$ and $\varepsilon_{P_{\mathrm{el}}}=10^{-3}$ afterwards.

Neutron kinetics

For neutron kinetics different models can be applied: point kinetics, 1D kinetics or 3D kinetics. However, 1D kinetics code is not maintained anymore.[2]

The control of the outer iteration loop is done by the variable ITNK, which affects the behavior in routine DSTART in the same way as the variables IQSTAF and IPOWER. In case of convergence, ITNK is set to 1. For point kinetics, no convergence criterion has to be met and consequently ITNK is always set to 1. The convergence criterion of 3D kinetics is specified and controlled by external software. For that reason, the convergence criterion cannot be described here.

Regarding numerics the only relevant algorithm for neutron kinetics is given in the routine NKBLOF where a linear system of equations is considered. This algorithm uses some kind of block factorization, which is probably based on a LU decomposition. As this algorithm is only needed for 1D kinetics, which currently has no relevance for ATHLET, it is not described in more detail here.

4.5 Comparison of the used methods with state-of-the-art numerical algorithms

The analysis of the SSC has shown that many small scale problems (involving one- or two-dimensional systems) are solved within the SSC. Large systems that would have a high potential to significantly accelerate simulations by the use of suitable numerical algorithms, practically do not exist in the SSC. For the solution of the small-scale systems, depending on the problem, methods of regula falsi type (e. g. Müller's method) or Newton's method are used. Only the applied fixed-point iterations (partly

[2] In ATHLET 3.4, 1D kinetics is no longer available.

with additional damping) have the potential to be replaced by more efficient algorithms. However, a change, for example to Newton's method, does not seem practicable, because this would require information about the derivative in contrast to using simple fixed-point iteration. It is expected that obtaining the information about the derivative and doing the necessary implementation work would involve a disproportional amount of effort. In the given scenario, the use of the secant method may be worth giving a try, since no derivative information is required.

4.5.1 NuT integration

Apart from the already established NuT support for calculating the specific energy in the CVs, see p. 99, further application of NuT does not seem promising during the SSC for the reasons given above. However, NuT may be used for a subsequent improvement of the approximation y_0^{SSC} which was obtained in the SSC as an initial value for the transient phase of the calculations via

$$y' = f(y,t), \qquad y(t_0) = y_0^{\mathrm{SSC}}. \tag{4.47}$$

It ought to hold that $f(y_0^{\mathrm{SSC}}) \approx 0$ (which is the whole purpose of the SSC). However, due to the environment of the transient phase where $f = \texttt{AFK}$ is considered for the first time, further improvements may be possible.

Using y_0^{SSC} as a reasonably good starting value, two approaches may be considered to achieve better approximation:

- Newton-type methods
- Optimization

In the optimization approach, the norm of f is minimized, that is, the problem

$$\min_y \|f(y)\| \tag{4.48}$$

for some suitable norm $\|\cdot\|$.

Both methods have to consider additional constraints $y_{i,min} < y_i < y_{i,max}$ for certain solution components y_i which apply for physical reasons. Examples are $0 \leqslant x_m \leqslant 1$ and $0 < p$.

The PETSc library provides several optimization algorithms (/BAL 23, TAO: Optimization Solvers/) and Newton-type solvers with constraints /BAL 23, SNES: Nonlinear Solvers/) which could be made available via NuT after some proper modifications.

Both Newton-type methods and the optimization approach require information about the derivative in each step. Thus, the Jacobian must be evaluated repeatedly, which could make this improvement of the approximation for the steady-state solution computationally expensive.

4.6 Suggestions for improvement of the SSC

During the re-evaluation and discussion of the methods applied in the SSC, some potential for improvement was detected. The following subsections contain various related suggestions. First, suggestions which have already been implemented within the current project are mentioned. Then, improvements which seem realizable within the current methodology of the SSC – and therefore would require only relatively small code modifications – are described. Thereafter, improvements which would need significant modifications of the methodology of the SSC are presented. Finally, some further modifications of the SSC are suggested which seem to be rather specific or of subordinate importance.

4.6.1 Improvements accomplished within the current project

Initialization with multiple non-condensable gases

For simulations with AC^2 it can be necessary to consider multiple non-condensable (NC) gases. Hence, it is necessary to initialize them. In older AC^2 versions it was only possible to initialize one single NC gas. By means of the work done in this project it is now possible to initialize multiple non-condensable gases. This feature was already made available for AC^2 2023.

The implementation realizes the following ansatz: For the initialization, the handling of the input data of the mixture is performed by the routine `DIMC`. The first NC gas is read in using the pseudo-keyword `INITGAS`. In this step also the relative gas partial pressure (variable `XQVMCI` in the code or X_{pg} in /SCH 23a/) is read in. The new implementation now allows the input of further NC gases by using the new pseudo-keyword `INITXVNC`. Under this pseudo-keyword, the mass fractions of the additional non-condensable gases to the total non-condensable gas mass can be input. The mass fraction for the first gas specified under the pseudo-keyword `INITGAS` is derived as the remainder to the summation value 1.0.

Within most of the SSC the material properties of vapor are used and are sufficient. Only for the density, the NC gases are already taken into account in the SSC, because

simulations can be very sensitive regarding density due to its impact on geodetic pressure while other material properties are typically less problematic. The calculation of the mixture density is performed by the routine DROI34. Therein, the mixture density is calculated from the mixture of vapor and NC gases:

$$\rho_m = X_{pg}\rho_g + (1.0 - X_{pg})\rho_v. \tag{4.49}$$

Note that for user input XQVMCI $= 0.0$ or $0.0 <$ XQVMCI < 1.0 with TVS $>$ T0 the partial pressure must be calculated by the routine MPTS. Otherwise the partial pressure is the value of XQVMCI from the input file. Further variables in (4.49) are the vapor density ρ_v and the density of the NC gases ρ_g. The latter is calculated from the mass quality of the single gas components (variable X_{gi}; vapor is not considered in this variable) according to equation (2.109) from /SCH 23a/:

$$\rho_g = \frac{1}{\sum X_{gi}/\rho_{gi}}. \tag{4.50}$$

Note that this procedure uses the closure equations of the non-condensable gas model for the transient calculation as of ATHLET 3.3 and therefore produces consistent results. Consequently, its consistency rests on the basic assumption of an ideal mixture of ideal gases, which underpins the implementation of this model in ATHLET.

4.6.2 Improvements applicable for the current methodology of the SSC

Prevention of undesired modifications of boundary conditions in the SSC

The ATHLET user input can result in an over-determined system of equations. For that reason, modifications are performed during SSC to solve this problem. However, in the current implementation there are modifications of the following boundary conditions that are undesired:

- It may happen that the pressure and specific enthalpy prescribed as boundary conditions in a TDV are modified. Typically, boundary conditions are well known and for that reason a modification is undesired. It would be possible to prevent a modification of the boundary conditions. The methodology would follow the procedure for closed loops, which means that the friction loss coefficients are modified in an outer iteration, see Section 4.4.2.2.
- For heat exchangers the surface of the heat exchanger may be modified by the SSC. This modification is undesired as the heat exchanger surface is a fixed geometric factor, typically well-known by the user. A better way to deal with the

over-determined system of equations would be to leave the heat exchanger surface as it is and to apply the modifications to the heat transfer coefficient (HTC) which is usually subject to uncertainty. This solution is feasible because mathematically heat exchanger surface and HTC are both factors of the same product and the modifications are just multiplied with this product. Although it does not introduce any new feature, this small change of the SSC can lead to a behavior which is much more comprehensible for the ATHLET user. In this case, though, the changed HTC value should be used for the HECU module output.

Switch off iteration of the pump rotational speed

In some cases it might be beneficial for experienced users to have the possibility to switch off the iteration of the pump rotational speed, see Section 4.4.4. For the specific case of a 3-loop facility it was very challenging to properly set specific known pressure losses. The option to manually set the rotational speed of the pump that is not overwritten by the iteration would be a good measure for experienced users to derive a proper simulation setup. For that reason the option to switch off the iteration of pump rotational speed would be a valuable improvement.

Solve a common system of equations for `XQM`, `HDOM` and `PRESS`

In the current implementation, first x_m and h_{dom} are determined iteratively in `DENTNE`. After that, the pressure p is determined iteratively in `DGGT0`. For a two-phase mixture in the target CV and `ICK0` $\neq 1$, `DZUK/DRTMI` is called in `DGGT0` after pressure calculation to determine new values of x_m and h_{dom} for the current iterated pressure. One possible improvement could be to solve a common system of equations for the solution variables `XQM`, `HDOM` and `PRESS`.

Consider hydraulic connections of currently autonomous systems

Autonomous systems currently have no influence on each other in the SSC (e. g. in the pressure iteration). However, such autonomous systems are often hydraulically connected, for example via cross-connection objects (CCO). It could be desirable to consider such a hydraulic connection and thus prevent disturbances at the beginning of the transient calculation.

More reliable convergence of the layer temperatures

For some cases, the layer temperatures are not converging, if the heat flux between fluid and wall changes direction between the iterations. This is often the case for small mass flows in a TFO (e. g. guide tube). A possible solution might be to increase the damping in the fixed-point iteration.

4.6.3 Improvements that need major modifications of the methodology of the SSC

4.6.3.1 Modification of friction coefficients

In the current implementation, the modification of friction coefficients depends on the order of the priority chains. For that reason, the solution of the ATHLET simulation can depend on user experience. It would be desirable to modify the friction coefficients independently of the priority chains.

However, to some extent it might be desired to keep the concept of priority chains. At least for the main priority chain it is often desired to have a higher priority, because for this priority chain the conditions are often better known than in other parts of the system. Consequently, it might be beneficial to keep the main priority chain but break up the priority chain concept for the modification of the friction coefficients for all subsequent chains.

The following concepts and ideas have been developed in order to improve the modification of the friction coefficients.

Concept "equal priority chains"

The idea behind this concept is to perform the modification of the friction coefficients independently of the order of the priority chains. In this case, the modification of the friction coefficients is not performed sequentially, with the second priority chain "reacting" to the changed pressures of the first priority chain, to guarantee consistent pressures at the branches. Instead, for the case of a closed loop, see Fig. 4.16, the two conditions "pressure at start CV consistent" and "pressure at branch CV consistent" are satisfied by the two variables $f_{\zeta,1}$ (friction coefficient modification factor in priority chain 1) and $f_{\zeta,2}$ (friction coefficient modification factor in priority chain 2). At least for simple systems this concept seems to result in the same behavior as the current concept using priority chains. However, if small modifications of the mass flow are allowed, the concept "equal priority chains" might become relevant.

In the case of a non-closed loop, the condition "pressure at start CV consistent" could be replaced by the condition that the specified pressure in the TDV – i. e. the boundary condition – must be satisfied. For this, a friction coefficient modification would then also have to take place for non-closed loops (not done so far). Thus, it could be possible to make a contribution to the prevention of undesired modifications of boundary conditions in the SSC (see above).

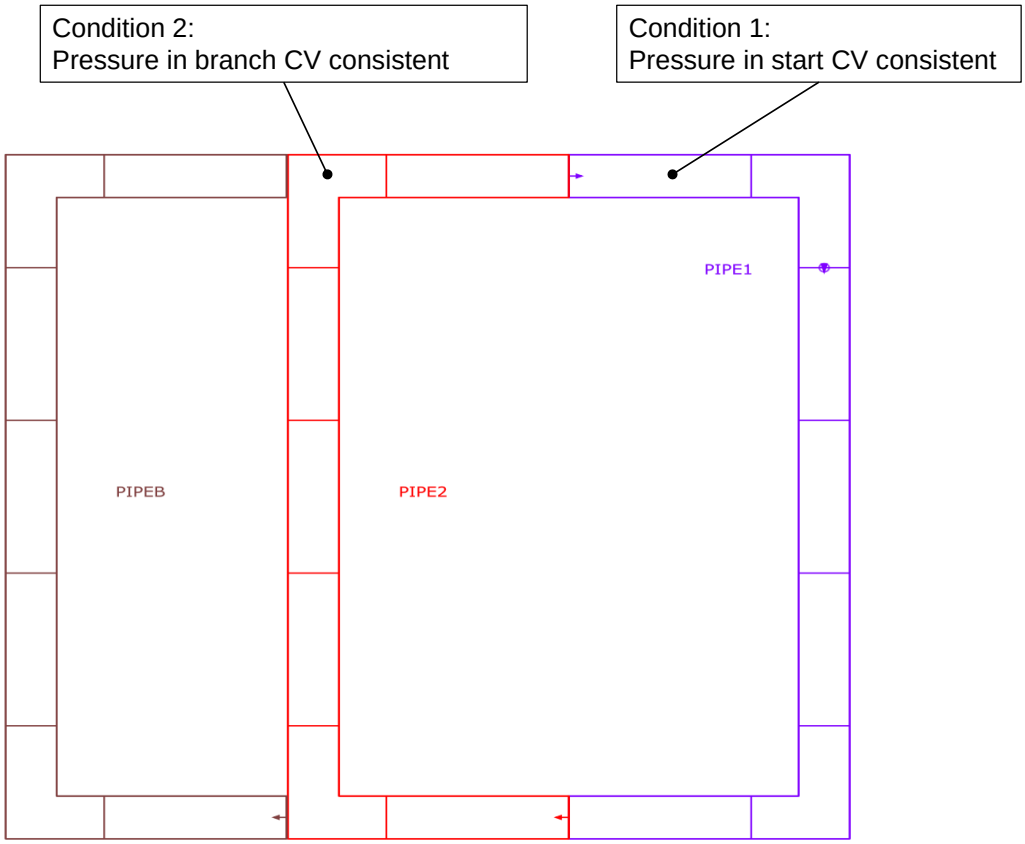

Fig. 4.16 Conditions for the concept "equal priority chains"

For the described concept, the procedure has to walk through the system in flow direction. This is not the case for the "voltage concept" (see below).

"Voltage concept"

The idea behind this approach is that in a parallel connection of two resistors in an electrical circuit the same voltage is applied across the parallel connected power lines, see Fig. 4.17.

For the friction coefficient modification, this approach no longer considers the concept of priority chains in their current form. Instead, beginning at the start CV of a thermo-fluiddynamic system, the next branch, i.e., a branch object or a pipe CV with minor branching connections, in all possible directions[3] is searched, see left part of Fig. 4.18. From these branches, a subdivision of the simulated system is then automatically made into individual chains that meet at the branches, see right part of Fig. 4.18. The friction coefficients of the individual chains are then adjusted so that consistent pressures are established at the branch CV.

[3] "Next branch" means the branch which is the least number of junctions away from the start CV.

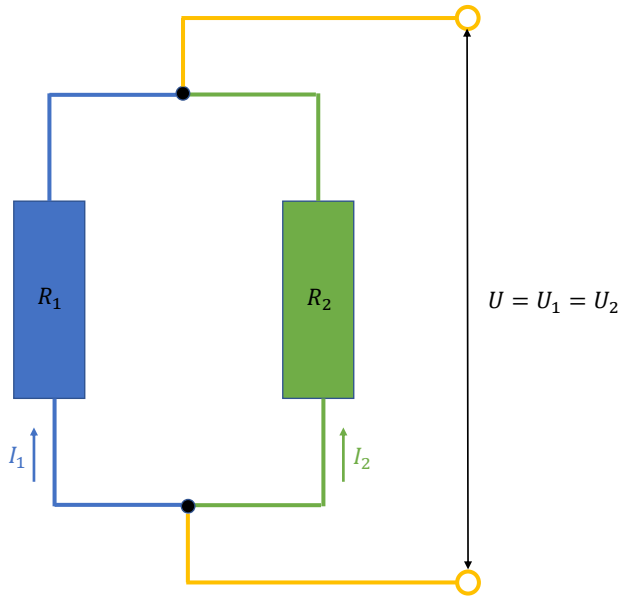

Fig. 4.17 Parallel connection of two resistors

In this concept, only the loss coefficients in chains connected in parallel would have to be adjusted (e. g. chains 2 and 3 in Fig. 4.18). However, it seems reasonable in the example to also modify the friction coefficients in chain 1. Here, the additional condition could be that the modification of the friction coefficients should be as small as possible. For this approach, it still has to be checked whether this reliably guarantees that the pressure in the start CV remains the same as the specified value.

As reliable friction losses are not known for all pipe sections, while for other sections these values are much better known, it would be advisable to additionally include a reliability factor. This reliability factor could be input by the user to limit friction coefficient modifications in well known pipe sections. As an alternative to the reliability factors, the user could read in a *range* of acceptable values for each pipe section the iteration procedure may vary the friction and form loss coefficients in.

The described concept will need some further discussion. Open questions are:

- The current concept using priority chains contains some information of the system, which supports modeling the physics properly. To include this information in the voltage concept, additional input might be required.
- The development and implementation of the algorithm which would be responsible for the automatic subdivision of the simulated system into the individual chains might turn out to be complicated for complex systems.

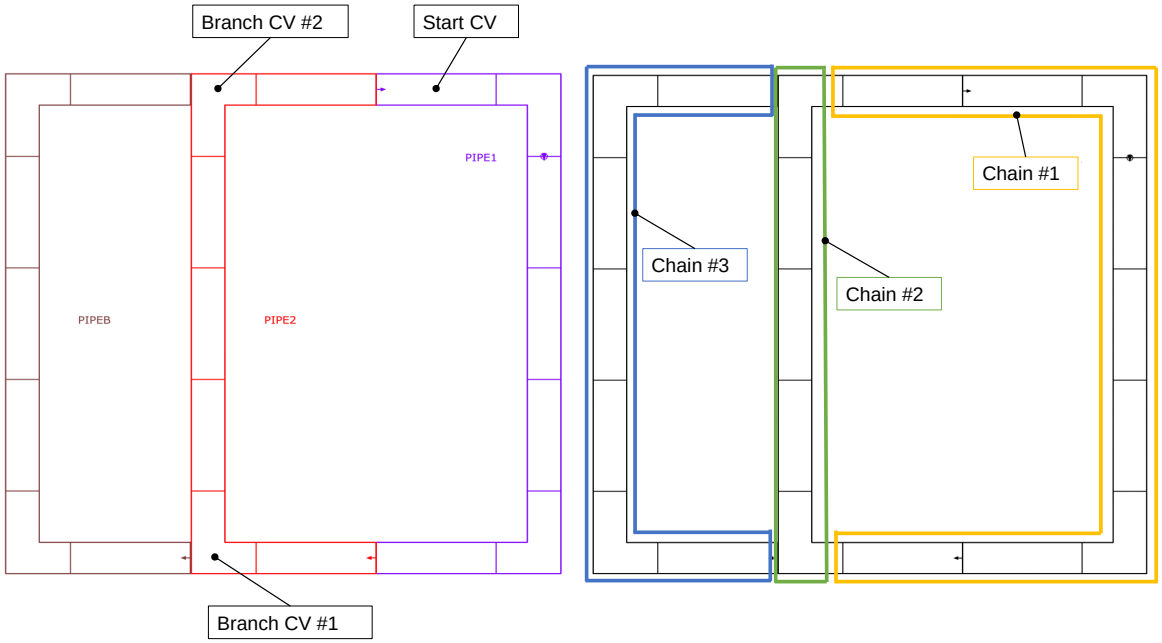

Fig. 4.18 Idea for implementation of the "voltage concept"

The proposed concept might be particularly useful for modeling parallel channels with inhomogeneous power distribution (a problem that currently needs iterative manual modifications by the user).

Further ideas

Further ideas have been brought up regarding the modification of the friction coefficients:

- The previously described concepts focus on the modification of the friction coefficients. However, it would also be possible to allow the modification of the mass flow rate. As the mass flow rate is approximately known by the user (at least in the main parts of the system), significant modifications of it should be avoided. For that reason, one possible concept would be to allow a limited modification of the mass flow rates and, in addition, to modify the friction loss coefficients. The idea of a reliability factor (see above) could also be used for the mass flow rate.
- Another concept is to take into account the absolute value of the mass flow rates in the junctions. The modification of the friction loss coefficients is then weighted according to the size of the local mass flow rates, e. g., by allowing the largest percentage modification of the friction loss coefficient in the junction with the lowest mass flow rate.
- Another option would be to minimize the deviation of all modified friction coefficients

from the input values. In this case, however, a reliability factor to steer adaptations to less well-determined values might be essential.

4.6.4 Further suggested modifications

- Under unfavorable conditions a small input error can lead to an initial condition that seems completely absurd. In one specific case, the core power was specified slightly asymmetrically, but the input for steam generator power was not adjusted accordingly (all steam generators were specified identically). Due to the adjustments made by the SSC and due to the non-mixing of the mass and energy flows during the SSC (mass flows on the CCOs were specified to be zero), an extremely asymmetric initial state of the dynamic calculation occurred, which significantly differed from the specified slightly asymmetric state. Thus, one could criticize an unexpected behavior of the SSC here which should be remedied.
- One user reported that it would be more comfortable if a priority chain could contain both, flowing and stagnant fluid. In the specific case, the user wanted to compare simulation results with experimental data of a T-junction. For most experiments there was flowing fluid in both pipes behind the T-junction, but for one experiment one pipe was a dead end with stagnant fluid. For the latter the data set had to be modified. A more user-friendly implementation would allow to simulate all experiments with the same data set.
- It would be desirable to intercept the inconsistency in `DGEQ0` that for a PC with stagnant fluid and closed loop the target pressure does not meet the pressure in the start CV if a pump is present, see Section 4.4.3.
- In routine `DENTNE` the mass quality x_m and the specific enthalpies h in the CV are determined iteratively. x_m and h of the first CV of a fluid dynamic system are fixed by the user input, where x_m must be 1 or 0 for the current implementation. It could be desirable to allow the user to specify two-phase conditions in the first CV of a TF system, i. e., an input of $0 \leqslant x_m \leqslant 1$. The input temperature `T0` would then correspond to the temperature of the dominant phase. However, a disadvantage is that the user usually does not know the exact two-phase state (x_m or `T0`). The effort of such a modification of the SSC would probably be too high in relation to the benefit.
- For parallel channels with different power (e. g. core or steam generator) suitable mass flows can be determined only iteratively by the user. It would be desirable if

the SSC could adapt the parallel channels in a way to avoid that a manual iterative adaptation is necessary. One possible remedy might be the voltage concept (see the above discussion on the concept).

5 Conclusions and Outlook

The main subject of this project was the further improvement of AC^2's numerical toolkit, NuT. To this end, firstly the functionality of NuT was extended in order to execute ODE methods. This was implemented in terms of a per-step logic where a method performs one time integration step per given step size. The logic required implementing interface routines to auxiliary PETSc procedures within NuT itself and verifying them. In addition, a selection of several ODE methods was made available. An extension of NuT's interfaces was taken care of to allow for an efficient access of the new ODE logic. Finally, the interfaces for the AC^2 codes ATHLET and COCOSYS (module THY) were implemented and tested. This required substantial work within the ATHLET and COCOSYS source code in order to provide an alternative control logic to the legacy `FEBE` approach. As a first step, the usage of NuT's ODE logic was made available for stand-alone calculations with ATHLET and COCOSYS. This was tested in verification calculations to check the correct implementation of these ODE methods and control means.

The next major progress achieved in this project was to establish the framework for thermo-hydraulically coupled calculations between ATHLET and COCOSYS in AC^2 simulations facilitated by advanced coupling numerics provided via NuT. Different coupling approaches were discussed theoretically, and boundary conditions and constraints from ATHLET and COCOSYS-THY were analyzed. Due to the complexity of implementing a coupling for these legacy codes, the monolithic coupling approach was chosen, where the differential equations from each code are treated within one unified overall ODE system. This implies the same time step for each code. To make this approach work, consistent derivative information about the influence of ATHLET on COCOSYS-THY zone variables at the coupling interfaces and vice versa had to be made available. To this end, NuT had to be extended to build up the overall Jacobian matrix from the Jacobians of ATHLET and COCOSYS and from the derivatives that represent the mutual influences. One major task was implementing a consistent treatment of discontinuities and time step reductions due to code model requirements for the coupled code system. Moreover, the AC^2 communication architecture had to optimized for this task. Within this project, a first functional implementation was achieved, but further work will be needed to fully establish this method for coupled AC^2 simulations. Furthermore, some open issues regarding thermal-hydraulics at the coupling interface were identified, which will need to be addressed as well.

Throughout the project, the code base of NuT was further refined by improving logging, enhancing the flexibility of the communications interface, and optimizing the seeding process as well as NuT's memory and process management. In the course of this, some refactoring of the code was done. In addition, the implementation of NuT was checked within the EU-funded project POP. This showed that NuT's implementation of its communication structure via MPI is efficient. Additionally, it was found that the instruction scalability of NuT appears to be weak. This is due to long waiting times for ATHLET processes to send data to NuT. Consequently, future efforts should be focused on enhancing the speed of the system code calculations, e. g., by parallel processing via OpenMP.

The second large topic in this project was improving the AC^2 development cycle, where substantial improvements could be realized. This includes particularly the automation of the AC^2 build process within the GRS GitLab infrastructure, facilitating not only continuous integration testing but also enabling continuous deployment of GRS-internal code versions. In addition, a structured approach to multi-project developments within the AC^2 program landscape was established, and tools and methods to apply this approach for development, testing and validation tasks were provided.

Finally, the steady calculation of ATHLET was analyzed in depth and documented accordingly. Based on this work, some improvements could be implemented, e. g., the initialization of ATHLET zones with multiple non-condensable gases. In addition, proposals and recommendations were derived for further improvements of the steady state calculation.

Overall, the main objectives of the project have been achieved. However, the work regarding coupled computations in AC^2 by means of a NuT-driven monolithic approach was found to be more challenging and thus took longer than initially anticipated, limiting the features of the current implementation. Missing features include partial updates of the Jacobian, ensuring consistency during restarts and contraction checks for Newton-type processes.

Follow-up work on improving NuT and the thermal-hydraulic coupling of ATHLET and COCOSYS within AC^2 should include the optimization of the time step control to allow for larger time steps. This should particularly address the recovery of time step sizes, which is currently limited by coupling quantities, after fast transient processes are over. Additionally, the robustness of the coupling interface in case of multiple coupling locations needs to be thoroughly tested in settings relevant for reactor calculations.

This should include coupling interfaces with two-phase flow, including counter-current flow conditions. Moreover, the presence of mixture-level models at the coupling interface should be investigated. For making use of the different time step sizes preferred by the ATHLET and COCOSYS calculation domain, consistent multi-rate methods should be developed and made available via NuT.

With regard to the AC^2 software development, further work should be done on establishing an improved infrastructure within the GRS GitLab instance for handling code changes affecting multiple code projects. This aims at verifying that code changes in one code project do not lead to problems or regression with regard to another code within AC^2.

References

/ARN 23/ Arndt, S. et al. *COCOSYS 3.2.0 – User Manual*. GRS-P-3 / Vol. 1. Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH. Nov. 2023 (cit. on p. 4).

/BAL 23/ Balay, S. et al. *PETSc Web page*. 2023. URL: https://petsc.org/ (visited on 2023-10-18) (cit. on pp. III, 49, 72, 129).

/BAL 97/ Balay, S., Gropp, W. D., Curfman McInnes, L., and Smith, B. F. "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries". In: *Modern Software Tools in Scientific Computing*. Ed. by Arge, E., Bruaset, A. M., and Langtangen, H. P. Birkhäuser Press, 1997, pp. 163–202 (cit. on p. 68).

/BAR 89/ Barcus, M. *Beschreibung und Diskussion des FEBE-Algorithmus*. Tech. rep. TN-BAM-89-01. Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH, June 1989 (cit. on p. 49).

/BOC 83/ Bock, H. G. "Recent Advances in Parameteridentification Techniques for O.D.E". In: *Numerical Treatment of Inverse Problems in Differential and Integral Equations*. Ed. by Hairer, E. and Deuflhard, P. Springer eBook Collection Mathematics and Statistics. Boston, MA: Birkhäuser Boston, 1983, pp. 95–121 (cit. on p. 10).

/BSC 23/ Barcelona Supercomputing Center. *Extrae*. 2023. URL: https://tools.bsc.es/extrae (visited on 2023-10-10) (cit. on p. 88).

/BUC 18/ Buchholz, S. et al. *EASY Integrale experimentelle und analytische Nachweise der Beherrschbarkeit von Auslegungsstörfällen allein mit passiven Systemen*. German. GRS 527. Köln; Garching b. München; Berlin; Braunschweig: Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH, Aug. 2018 (cit. on pp. 7, 23).

/BUT 16/ Butcher, J. C. *Numerical Methods for Ordinary Differential Equations*. 3rd ed. John Wiley & Sons, Ltd., 2016 (cit. on p. 2).

/BUT 90/ Butcher, J. C. and Cash, J. R. “Towards Efficient Runge-Kutta Methods for Stiff Systems”. In: *SIAM J. Numer. Anal.* 27.3 (June 1990), pp. 753–761 (cit. on p. 30).

/COL 83/ Coleman, T. F. and More, J. J. “Estimation of Sparse Jacobian Matrices and Graph Coloring Problems”. In: *SIAM J. Numer. Anal.* 20 (1983), pp. 187–209 (cit. on p. 68).

/COR 13/ Corless, R. M. and Fillion, N. *A graduate introduction to numerical methods: From the viewpoint of backward error analysis*. Mathematics. New York, NY and Heidelberg: Springer, 2013 (cit. on p. 2).

/GIT 24a/ GitLab Inc. *GitLab Docs*. 2024. URL: `https://docs.gitlab.com/ee` (visited on 2024-02-29) (cit. on p. 2).

/GIT 24b/ GitLab Inc. *GitLab Web page*. 2024. URL: `https://about.gitlab.com/` (visited on 2024-02-29) (cit. on p. 63).

/GUS 94/ Gustafsson, K. “Control theoretic techniques for stepsize selection in implicit Runge-Kutta methods”. In: *ACM Trans. Math. Softw.* 4 (1994), pp. 496–517 (cit. on p. 52).

/HAI 93/ Hairer, E., Nørsett, S. P., and Wanner, G. *Solving Ordinary Differential Equations I: Nonstiff Problems*. 2nd ed. Vol. 8. Springer Series in Computational Mathematics. Springer Berlin Heidelberg, 1993 (cit. on pp. 2, 10, 31).

/HAI 96/ Hairer, E. and Wanner, G. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. 2nd ed. Vol. 14. Springer Series in Computational Mathematics. Springer Berlin Heidelberg, 1996 (cit. on pp. 2, 29 sqq., 49, 52).

/HUN 00/ Hunt, A. and Thomas, D. *The Pragmatic Programmer: From Journeyman to Master*. USA: Addison-Wesley Longman Publishing Co., Inc., 2000 (cit. on p. 64).

/HUN 03/ Hundsdorfer, W. and Verwer, J. G. *Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations*. Vol. 33. Springer Series in Computational Mathematics. Springer Berlin Heidelberg, 2003 (cit. on p. 30).

/ICL 23/ Innovative Computing Laboratory, University of Tennessee, Knoxville. *Performance Application Programming Interface (PAPI)*. 2023. URL: https://icl.utk.edu/papi (visited on 2023-10-10) (cit. on p. 88).

/INT 23/ Intel Corp. *Developer Reference for Intel® oneAPI Math Kernel Library for C*. 2023 (cit. on p. 69).

/JAC 23a/ Jacht, V. *MMA – MPI for Multiple Applications*. 2023. URL: https://gitlab.com/nordfox/mma (visited on 2023-10-19) (cit. on p. 72).

/JAC 23b/ Jacht, V., Scheuer, J., Schöffel, P., and Wielenberg, A. *ATHLET 3.4.0 – Programmer's Manual*. GRS-P-1 / Vol. 2 Rev. 11. Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH. Nov. 2023 (cit. on pp. 83, 92).

/KAR 23/ Karypis, G. and Kumar, V. *Metis Web page*. 2023. URL: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview (visited on 2023-10-20) (cit. on p. 68).

/KEI 99/ Keil, F., Mackens, W., Voß, H., and Werther, J., eds. *Scientific Computing in Chemical Engineering II: Simulation, Image Processing, Optimization and Control*. Springer Berlin Heidelberg, 1999 (cit. on p. 158).

/KIT 23/ Kitware. *CMake Web page*. 2023. URL: https://cmake.org/ (visited on 2023-02-02) (cit. on pp. 63, 70).

/KLE 23/ Klein-Heßling, W. et al. *COCOSYS 3.2.0 – Models and Methods*. GRS-P-3 / Vol. 2. Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH. Nov. 2023 (cit. on pp. 4, 13 sqq.).

/LIP 12/ Lippman, S., Lajoie, J., and Moo, B. *C++ Primer*. 5th. Addison-Wesley Professional, 2012 (cit. on p. 2).

/LLVM 23/ The LLVM Team. *LLVM Clang Web page*. 2023. URL: https://clang.llvm.org/ (visited on 2023-12-14) (cit. on p. 77).

/LOT 98/ Lotka, A. J. *Analytical Theory of Biological Populations*. The Springer Series on Demographic Methods and Population Analysis. Boston, MA: Springer, 1998 (cit. on p. 39).

/MAR 03/ Martin, R. C. *Agile Software Development: Principles, Patterns, and Practices*. USA: Prentice Hall PTR, 2003 (cit. on p. 64).

/MAR 17/ Martin, R. C. *Clean Architecture*. Harlow: Pearson Education, 2017 (cit. on p. 2).

/MEY 97/ Meyer, B. *Object-Oriented Software Construction*. 2nd ed. USA: Prentice-Hall, Inc., 1997 (cit. on p. 64).

/MPI 23/ Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*. Nov. 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf (cit. on p. 2).

/MUM 23/ Mumps Technologies. *MUMPS Web page*. 2023. URL: http://mumps-solver.org (visited on 2023-10-20) (cit. on p. 68).

/POI 78/ Pointner, W. *Startrechnung für den Blowdowncode DRUFAN*. Tech. rep. TN-POI-78. Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH, Dec. 1978 (cit. on p. 102).

/POP 23/ POP Centre of Excellence. *POP Web page*. 2023. URL: https://pop-coe.eu (visited on 2023-10-09) (cit. on pp. 63, 88).

/ROS 22/ Rose, M. *NuT – performance audit: (POP2_AR_162)*. 2022 (cit. on p. 89).

/SCH 23a/ Schöffel, P. et al. *ATHLET 3.4.0 – Models and Methods*. GRS-P-1 / Vol. 4 Rev. 8. Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH. Nov. 2023 (cit. on pp. 13 sqq., 28 sq., 96, 117, 124, 130 sq.).

/SCH 23b/ Schöffel, P. et al. *ATHLET 3.4.0 – User's Manual*. GRS-P-1 / Vol. 1 Rev. 11. Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH. Nov. 2023 (cit. on p. 91).

/SCO 20/ Scott, C. *Professional CMake : A Practical Guide*. 7th ed. 2020. URL: https://crascit.com/professional-cmake/ (visited on 2020-10-23) (cit. on p. 71).

/SÖD 03/ Söderlind, G. "Digital filters in adaptive time-stepping". In: *ACM Trans. Math. Softw.* 29 (2003), pp. 1–26 (cit. on p. 52).

/STE 17a/ Steinhoff, T. "Providing a Single Point Spectrum for Runge-Kutta Schemes of High Stage Order Based on Perturbed Collocation". In: *AIP Conference Proceedings*. Vol. 1863. July 2017 (cit. on pp. 31, 36).

/STE 17b/ Steinhoff, T. and Jacht, V. *Ausbau und Modernisierung der numerischen Verfahren in den Systemcodes ATHLET, ATHLET-CD, COCOSYS und ASTEC*. German. GRS 469. Köln; Garching b. München; Berlin; Braunschweig: Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH, July 2017 (cit. on pp. IV, 1, 30 sqq., 39, 52).

/STE 20/ Steinhoff, T. and Jacht, V. *Weiterentwicklung und Ausbau numerischer Strukturen in den AC2-Programmen ATHLET und COCOSYS*. German. GRS 600. Köln; Garching b. München; Berlin; Braunschweig: Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH, Aug. 2020 (cit. on pp. IV, 1, 41).

/STE 23a/ Steinhoff, T. *QA plan for NuT*. Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH, Dec. 2023 (cit. on p. 83).

/STE 23b/ Steinhoff, T. and Jacht, V. *NuT 2.0.2 Numerical Toolkit – User's Manual*. GRS-P-10 / Vol. 1 Rev. 7. Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH. 2023 (cit. on pp. 90, 99).

/UNI 23a/ Univ. of Tennessee, Univ. of California, Berkeley, Univ. of Colorado Denver, and NAG Ltd. *LAPACK*. 2023. URL: https://www.netlib.org/lapack (visited on 2023-10-17) (cit. on p. 68).

/UNI 23b/ Univ. of Tennessee, Univ. of California, Berkeley, Univ. of Colorado Denver, and NAG Ltd. *ScaLAPACK*. 2023. URL: https://www.netlib.org/scalapack (visited on 2023-10-17) (cit. on p. 68).

/WEY 23/ Weyermann, F. et al. *AC2 2023.0 – User Manual*. GRS-P-15 / Vol. 1 Rev. 1. Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH. Dec. 2023 (cit. on pp. III, 1, 69).

Acronyms

AIG	ATHLET input graphics
BC	Building condenser
CD	Continuous delivery
CI	Continuous integration
CSC	Compressed sparse column
CSR	Compressed sparse row
CV	Control volume
EC	Emergency condenser
FiterRK	Finite iteration Runge–Kutta [method/scheme]
GCSM	General control simulation module within ATHLET
HCV	Heat conduction volume
HECU	Heat conduction module within ATHLET
HSZG	Zittau/Görlitz University of Applied Sciences
INKA	Integraltestst and Karlstein
IVP	Initial value problem
MPI	Message passing interface
NC	Non-condensable
NEUKIN	Neutron kinetics module in ATHLET
ODE	Ordinary differential equation
PC	Priority chain
PETSc	Portable, Extensible Toolkit for Scientific Computation

RK	Runge–Kutta [method/scheme]
ROW	Rosenbrock–Wanner [method/scheme]
SBTL	Spline-based table look-up
SIWAP	Verbesserungen der Simulation von Siedevorgängen in AC2 bei lokalem Eintrag von Wärme in Wasserpools
SSC	Steady state calculation
TDV	Time dependent volume
TFD	Thermo-fluid dynamics module within ATHLET
TFO	Thermo-fluid dynamic object

List of Figures

List of Tables

List of Codes

A Appendix on details of WP1

A.1 Applying the chain rule for calculating the derivatives in (2.6)

Considering the notation from Fig. 2.3 the matrices A and T are easy to come by since these are the matrices that the codes are able to determine anyway for their single code computations. The problem to be solved is the calculation of the matrix values of the off-diagonal submatrices LL and UR. The finally implemented procedure for this purpose is described in Section 2.2.1. However, prior to this implementation, a different approach was pursued, namely the application of the chain rule for calculating the matrix values. Details of the idea behind this approach and why it was discarded eventually are briefly discussed in the following.

For the sake of simplicity, the focus is on the UR matrix as the LL matrix can be build analogously. The UR matrix describes the first order influence of perturbations in the THY solution variables on the ODEs of the ATHLET system:

$$UR = \frac{\partial f}{\partial u}.$$

Applying the chain rule, it can be expressed as

$$\frac{\partial f}{\partial u} = \frac{\partial f}{\partial \alpha_{ATH}} \cdot \frac{\partial \alpha_{ATH}}{\partial \alpha_{THY}} \cdot \frac{\partial \alpha_{THY}}{\partial u} \approx \frac{\Delta f}{\Delta \alpha_{ATH}} \cdot \frac{\Delta \alpha_{ATH}}{\Delta \alpha_{THY}} \cdot \frac{\Delta \alpha_{THY}}{\Delta u}. \tag{A.1}$$

Note that all factors of the product are matrices. The quantities f, u, and α are defined as in Section 2.1. For (A.1), α is further partitioned into α_{THY} (the coupling variables as they are *provided by THY*) and α_{ATH} (the coupling variables as they are *accepted and used by ATHLET*). Between both of them exists a – simple or complex – transformation function.

Regarding the determination of the three factors in (A.1), it can be stated that

- the first factor $\frac{\Delta f}{\Delta \alpha_{ATH}}$ can be completely calculated in the ATHLET process,
- the last factor $\frac{\Delta \alpha_{THY}}{\Delta u}$ can be completely calculated in the THY process,
- the calculation of the central factor $\frac{\Delta \alpha_{ATH}}{\Delta \alpha_{THY}}$ needs information from both processes.

In an early stage of the project, this approach was partially implemented in ATHLET and COCOSYS development branches. However, for the calculation of the difference quotient $\frac{\Delta \alpha_{ATH}}{\Delta \alpha_{THY}}$, knowledge of the internals of the transformation function is needed. It finally turned out that the approach of calculating the derivatives by combining the idea of internal numerical differentiation with finite differences – as described in Section 2.2.1 – is simpler in that respect and therefore favorable.

A.2 Linear systems in a monolithic approach by means of a TBN ansatz

The Tangent Block Newton (TBN) ansatz discussed in /KEI 99/ may be applied to the linear systems that arise in a monolithic approach. The idea is to solve the linear systems related to the single ODE systems as usual and then to apply a correction to reflect the influence of the overall system. Still, the full Jacobian information is required but no system of the dimension of the overall system needs to be solved.

The drawback, however, is poor scalability since the number of required linear systems to solve is proportional to the number of coupling quantities in α and β of (2.3) combined. With n_c denoting the number of coupling interfaces the number of coupling quantities is in the range of $\mathcal{O}(10 \cdot n_c)$. This is considerable and impacts performance. Furthermore, after establishing the composition of a matrix out of four submatrices in NuT, see Section 2.3.2.1, the linear systems to solve can be treated exactly the same way as before.

In summary, the drawbacks of a TBN approach makes it inferior to the implemented logic. Hence, the idea was not pursued any further.

A.3 On the dependency of coupling variables

An example is given to illustrate the fact that the coupling input β from (2.3) does not solely depend on y, but also on α.

Given the situation that a mixture of NC gases and vapor is flowing from the THY into the ATHLET simulation domain, the component enthalpy flow rates – such as the vapor enthalpy flow rate $\dot{H}_{vapor}$ – on the coupling junction are calculated by ATHLET and provided to THY as boundary conditions; obviously, $\dot{H}_{vapor}$ is a component of β. For the sake of simplicity, let

$$\beta = \dot{H}_{vapor}.$$

Since ATHLET applies an upwind (a. k. a. donor cell) scheme for the determination of flow quantities, among other things, $\dot{H}_{vapor}$ depends on the specific enthalpy of vapor in the coupling CV:

$$\beta = \dot{H}_{vapor}\left(h_{vapor}\right).$$

The specific vapor enthalpy is calculated as a function of temperature and partial vapor pressure in the coupling CV:

$$\beta = \beta\Big(h_{vapor}\left(T_{copl}, p_{vapor,copl}\right)\Big).$$

The functional dependency $h_{vapor}\left(T_{copl}, p_{vapor,copl}\right)$ is evaluated in ATHLET by a non-trivial fluid property subroutine. The partial vapor pressure is calculated from the static pressure in the coupling CV and the volume fraction of vapor:

$$\beta = \beta\Big(h_{vapor}\left(T_{copl}, p_{copl}, x_{vapor,copl}\right)\Big) = \beta\left(T_{copl}, p_{copl}, x_{vapor,copl}\right).$$

The quantities T_{copl}, p_{copl}, and $x_{vapor,copl}$ are all provided by THY and are therefore components of α. T_{copl} is a component of u, too, and $x_{vapor,copl}$ can be simply derived from u. The quantity p_{copl}, however, is calculated in THY as the sum of all partial pressures, which in turn depend on u, but in a non-trivial way (the non-trivial dependency is implemented in a THY fluid property subroutine). This yields:

$$\beta = \beta\left(T_{copl}, p_{copl}, ...\right) = \beta\Big(T_{copl}, \sum \underbrace{p_{part}\left(u\right)}_{\substack{\text{non-trivial dependency,}\\ \text{calculated in THY}}}, ... \Big).$$

And this means

$$\beta = \beta\left(\alpha\right).$$

Of course, $\beta = \dot{H}_{vapor}$ depends not only on h_{vapor}, but on $\dot{m}$, the total mass flow rate on the junction, too. Since $\dot{m}$ is a component of y, one obtains

$$\beta = \beta\left(\dot{m}, h_{vapor}\right) = \beta\left(y, \alpha\right).$$

B Appendix on details of WP3

The iteration of enthalpy and mass quality is described in Section 4.4.1. Here, the focus is on the discussions of the functions roots are sought for. It is pointed out again that x_m, h_{dom}, p and $\dot{m}$ are the solution variables of the 4-equation model that is used in the SSC.

B.1 Müller's method

Müller's method is used to iteratively find the roots of an equation. Thermal equilibrium is assumed as well as that both phases (liquid and vapor) have saturation conditions. The variable quantity for Müller's method is x_m. The value $x_m^{\text{Müller}}$ that is found by the method is afterwards used as initial value for Newton's method.

B.1.1 Initial value

Müller's method is a modified bisection method. Initial values are the endpoints of the interval $x_m^{\langle 0,low\rangle}$ and $x_m^{\langle 0,up\rangle}$:

$$x_m^{\langle 0,\mathrm{low}\rangle} = 10^{-12}, \tag{B.1a}$$

$$x_m^{\langle 0,\mathrm{up}\rangle} = \frac{e_{\mathrm{tar}} - h'(p)}{\Delta h_v(p)}. \tag{B.1b}$$

B.1.1.1 Why is $x_m^{\langle 0,\mathrm{up}\rangle}$ a reasonable initial value for the upper bound?

In Section 4.4.1 it is shown that

$$\dot{x}_{m,SR}(x_m) = \frac{e_{\mathrm{tar}} - h'}{h'' - h'} = \frac{e_{\mathrm{tar}} - h'}{\Delta h_v} \tag{B.2}$$

would be a root of the function (B.6). Furthermore, it is claimed that usually $\dot{x}_{m,SR}(x_m) > x_m$ holds true so that (B.1b) is not a root of (B.6), but an upper bound. Below, it is briefly demonstrated why this claim is usually true (and in which cases it is not).
In mechanical non-equilibrium, the vapor and liquid phases usually do not have the same flow velocities. The ratio of vapor velocity compared to liquid velocity is called the slip ratio SR:

$$SR = \frac{v_{\mathrm{vap}}}{v_{\mathrm{liq}}}. \tag{B.3}$$

The slip ratio has to be taken into account when formulating the flow quality as a function of the static void fraction α:

$$\dot{x}_{m,SR} = \frac{1}{1 + \frac{\rho_{\mathrm{liq}}}{\rho_{\mathrm{vap}}} \cdot \frac{1-\alpha}{\alpha} \cdot \frac{1}{SR}}. \tag{B.4}$$

A slip ratio of 1 corresponds to mechanical equilibrium so that the flow quality equals the static quality:

$$\dot{x}_{m,SR}\,(SR = 1) = x_m = \frac{1}{1 + \frac{\rho_{\mathrm{liq}}}{\rho_{\mathrm{vap}}} \cdot \frac{1-\alpha}{\alpha}}. \tag{B.5}$$

Usually, the velocity of the vapor is higher than that of the liquid which means that the slip ratio is larger than one, which again means that $\dot{x}_{m,SR} > x_m$, as can be seen in (B.4) or, graphically represented, in Fig. B.1.

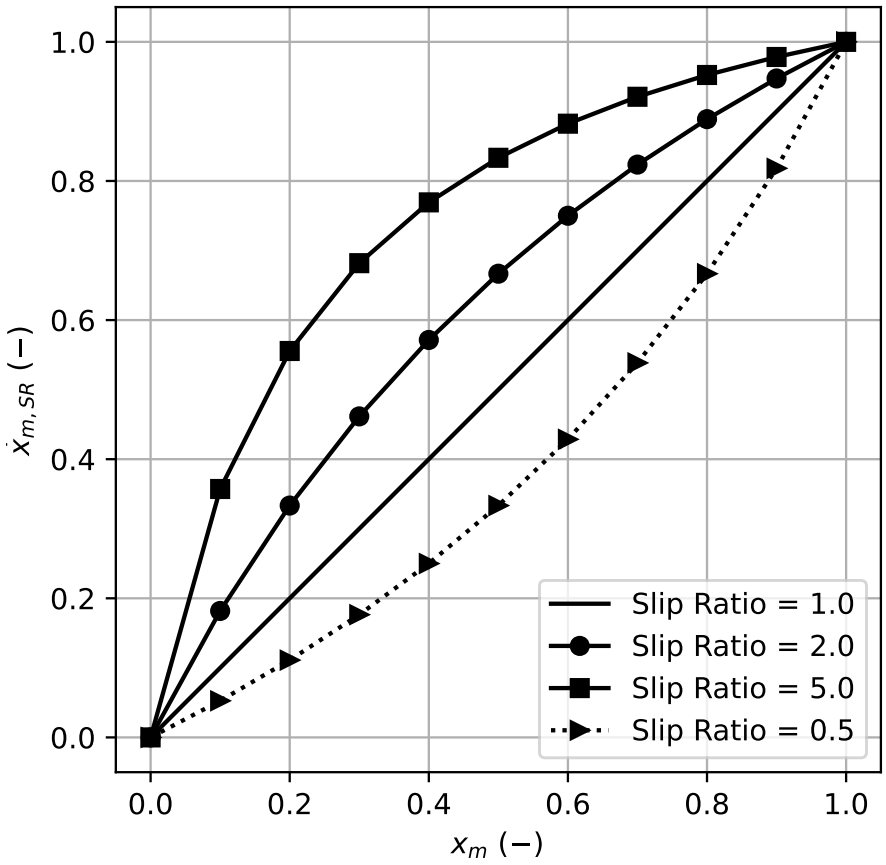

Fig. B.1 $\dot{x}_{m,SR}$ over x_m for various slip ratios. Note that the slip ratio is typically larger than one. Vapor and liquid density were chosen as steam/water properties at 100 °C and saturation pressure.

This in turn means that if $x_m = \frac{e_{\mathrm{tar}} - h'}{\Delta h_v}$ as given in (B.1b), the corresponding $\dot{x}_{m,SR}(x_m)$ will *usually* be larger, thus yield an $e_{\mathrm{cur}} > e_{\mathrm{tar}}$, and serve as an upper bound for the bisection.

Of course, problems are expected in the case of counter-current flow ($SR < 0$) or if the liquid flows faster than the vapor ($SR < 1$). Apparently, the ATHLET developers tried to deal with this issue in the past as can be concluded from deactivated code lines in the respective routines `DSLIP` and `DFCTXM`. The way the SSC in its current state treats these cases (which it obviously does successfully), needs more investigation.

B.1.2 Function $f_M\left(x_m\right)$

The root of the following function is searched for:

$$f_M\left(x_m\right) = \frac{e_{\mathrm{tar}} - e_{\mathrm{cur}}}{\Delta h_v\left(p\right)}. \tag{B.6}$$

With

$$e_{\mathrm{tar}} = \mathtt{HPKI}(I) \tag{B.7}$$

from `DENTH` (resulting there from the solution of a system of linear equations) and

$$\begin{aligned} e_{\mathrm{cur}} =& \dot{x}_{m,SR} \cdot \left[h''\left(p\right) + e_{\mathrm{kin}}\left(\dot{m}, p, \dot{x}_{m,SR}\right)\right] + \\ &+ \left(1 - \dot{x}_{m,SR}\right) \cdot \left[h'\left(p\right) + e_{\mathrm{kin}}\left(\dot{m}, p, \dot{x}_{m,SR}\right)\right]. \end{aligned} \tag{B.8}$$

(B.6) becomes

$$\begin{aligned} f_M\left(x_m\right) = & \frac{\mathtt{HPKI}\left(I\right) - \dot{x}_{m,SR} \cdot \left[h''\left(p\right) + e_{\mathrm{kin}}\left(\dot{m}, p, h'\left(p\right), \dot{x}_{m,SR}\right)\right]}{\Delta h_v\left(p\right)} + \\ &+ \frac{\left(1 - \dot{x}_{m,SR}\right) \cdot \left[h'\left(p\right) + e_{\mathrm{kin}}\left(\dot{m}, p, h'\left(p\right), \dot{x}_{m,SR}\right)\right]}{\Delta h_v\left(p\right)}. \end{aligned} \tag{B.9}$$

Here, $\dot{x}_{m,SR}$ (SR stands for slip ratio) is no ATHLET solution variable and depends in a non-trivial way on x_m:

$$\begin{aligned} \dot{x}_{m,SR} &= \dot{x}_{m,SR}\left(x_m, SR\right) \\ &= \dot{x}_{m,SR}\Big(x_m, SR\left(x_m, p, h'\left(p\right), h''\left(p\right), \dot{m}\right)\Big). \end{aligned} \tag{B.10}$$

The quantity $\dot{m}$ is the mass flow out of a CV. It is, contrary to the other quantities, junction related, not CV related. The junctions are staggered to the CV.
Note: If $\mathtt{CC} > 0.0$ is set in the ATHLET input under the control word `MISCELLAN`, the calculation of SR does not only depend on the conditions of the donor CV but also on the conditions of the downstream CV. This results in even more complex dependencies compared to (B.9). Furthermore, due to the successive procedure "CV by CV", quantities are used for the calculation of SR that have not yet been updated. For the sake of simplicity, only the donor cell mode $\mathtt{CC} = 0.0$ is treated here.

B.2 Newton's method

The system under consideration consists of two scalar functions, namely, $f_N\left(x_m, h_{\mathrm{dom}}\right)$ and $g_N\left(x_m, h_{\mathrm{dom}}\right)$. Newton's method is applied together with the 4-equation model, which means that one phase has saturation conditions and the other phase (the dominant phase) may deviate from saturation conditions. For simplicity, it is assumed here that the liquid phase is the dominant phase. For systems with dominant vapor phase the procedure is analogous. The variable quantities for Newton's method are x_m and h_{dom}.

B.2.1 Initial values

The initial values of Newton's method are as follows:

- $x_m^{\langle 0\rangle} = x_m^{\text{Müller}}$,
- $h_{dom}^{\langle 0\rangle} = h'$ (if liquid phase dominant; else $h_{dom}^{\langle 0\rangle} = h''$).

B.2.2 Function $f_N\left(x_m, h_{\mathrm{dom}}\right)$

The root of the following function is searched for:

$$f_N\left(x_m, h_{\mathrm{dom}}\right) = \frac{e_{\mathrm{tar}} - e_{\mathrm{cur}}}{\Delta h_v\left(p\right)}. \tag{B.11}$$

Contrary to the definition in (B.8) e_{cur} now also depends on h_{dom}:

$$\begin{aligned} e_{\mathrm{cur}} =& \dot{x}_{m,SR} \cdot \left[h''\left(p\right) + e_{\mathrm{kin}}\left(\dot{m}, p, h_{\mathrm{dom}}, \dot{x}_{m,SR}\right)\right] \\ &+ \left(1 - \dot{x}_{m,SR}\right) \cdot \left[h_{\mathrm{dom}} + e_{\mathrm{kin}}\left(\dot{m}, p, h_{\mathrm{dom}}, \dot{x}_{m,SR}\right)\right]. \end{aligned} \tag{B.12}$$

With (B.7), (B.11) becomes:

$$\begin{aligned} f_N\left(x_m, h_{\mathrm{dom}}\right) = & \frac{\texttt{HPKI}\left(I\right) - \dot{x}_{m,SR} \cdot \left[h''\left(p\right) + e_{\mathrm{kin}}\left(\dot{m}, p, h_{\mathrm{dom}}, \dot{x}_{m,SR}\right)\right]}{\Delta h_v\left(p\right)} + \\ & + \frac{\left(1 - \dot{x}_{m,SR}\right) \cdot \left[h_{\mathrm{dom}} + e_{\mathrm{kin}}\left(\dot{m}, p, h_{\mathrm{dom}}, \dot{x}_{m,SR}\right)\right]}{\Delta h_v\left(p\right)}. \end{aligned} \tag{B.13}$$

Note the similarity of (B.13) and (B.9). As for Newton's method, thermal equilibrium is not assumed and one phase (here the liquid phase) may deviate from saturation conditions, h' has become h_{dom}.

$\dot{x}_{m,SR}$ is still no ATHLET solution variable and depends in a non-trivial way on x_m and h_{dom}:

$$\begin{aligned} \dot{x}_{m,SR} &= \dot{x}_{m,SR}\left(x_m, SR\right) \\ &= \dot{x}_{m,SR}\Big(x_m, SR\left(x_m, p, h_{\mathrm{dom}}, h''\left(p\right), \dot{m}\right)\Big). \end{aligned} \tag{B.14}$$

Regarding $\dot{m}$, note the comment in Section B.1.2.

B.2.3 Function $g_N\left(x_m, h_{\mathrm{dom}}\right)$

The root of the following function is searched for:

$$g_N\left(x_m, h_{\mathrm{dom}}\right) = \dot{x}_{m,SR} - x_{m,\mathrm{MB}}. \tag{B.15}$$

Here $\dot{x}_{m,SR}$ denotes the mass flow quality of the junction with outflow of the considered CV. The mass flow quality results from the drift model and its calculation is according to (B.14). $x_{m,\mathrm{MB}}$ is the mass quality of the considered CV, which results from mass balance. For mass balance the model equations of Sideman and of Plesset & Zwick are applied for the calculation of evaporation or condensation rate. The calculation of $x_{m,\mathrm{MB}}$ in CV I is as follows:

$$\begin{aligned} x_{m,\mathrm{MB}}\left(I\right) &= \frac{\dot{m}_{\mathrm{evap}} + \sum \dot{m}_{\mathrm{in,vap}}}{\sum \dot{m}_{\mathrm{in,vap}} + \sum \dot{m}_{\mathrm{in,liq}}} \\ &= \frac{\dot{m}_{\mathrm{evap}}\left(x_m, h_{\mathrm{dom}}, p\right) + \sum \dot{m}_{\mathrm{in,vap}}}{\sum \dot{m}_{\mathrm{in}}} \\ &= \frac{\dot{m}_{\mathrm{evap}}\left(x_m, h_{\mathrm{dom}}, p\right) + \sum \dot{m}_{\mathrm{in}} \cdot \dot{x}_{m,SR}\left(K\right)}{\sum \dot{m}_{\mathrm{in}}} \\ &= \frac{\dot{m}_{\mathrm{evap}}\left(x_m, h_{\mathrm{dom}}, p\right) + \sum \dot{m}_{\mathrm{in}} \cdot \dot{x}_{m,SR}\Big(x_m\left(K\right), h_{\mathrm{dom}}\left(K\right)\Big)}{\sum \dot{m}_{\mathrm{in}}}. \end{aligned} \tag{B.16}$$

As can be seen from (B.14) and (B.16), the used models (evaporation or condensation rate and drift) are functions of x_m and h_{dom}.
If the dependencies are shown in a very simplified way, (B.15) becomes

$$\begin{aligned} g_N\left(x_m, h_{\mathrm{dom}}\right) = \dot{x}_{m,SR}&\left(x_m, h_{\mathrm{dom}}\right) \\ &- x_{m,\mathrm{MB}}\left(x_m, h_{\mathrm{dom}}, \sum x_m\left(K\right), \sum h_{\mathrm{dom}}\left(K\right)\right). \end{aligned} \tag{B.17}$$

Here, x_m and h_{dom} are abbreviated forms for $x_m\left(I\right)$ and $h_{\mathrm{dom}}\left(I\right)$ and stand for the mass quality or specific enthalpy of the dominant phase in the considered CV I, while $x_m\left(K\right)$ and $h_{\mathrm{dom}}\left(K\right)$ are the corresponding physical quantities from all junctions that are connected with CV I and that are located upwind of that CV. These values are not changing within Newton's iteration for CV I, but during the outer iteration loop of the SSC.

Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH

Schwertnergasse 1
50667 Köln
Telefon +49 221 2068-0
Telefax +49 221 2068-888

Boltzmannstraße 14
85748 Garching b. München
Telefon +49 89 32004-0
Telefax +49 89 32004-300

Kurfürstendamm 200
10719 Berlin
Telefon +49 30 88589-0
Telefax +49 30 88589-111

Theodor-Heuss-Straße 4
38122 Braunschweig
Telefon +49 531 8012-0
Telefax +49 531 8012-200

www.grs.de

ISBN 978-3-910548-49-7