Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

# Performance Optimization of Atmospheric Model ECHAM6 through Component Concurrency

**Dissertation**

zur Erlangung des akademischen Grades

Dr. rer. nat

an der Fakultät für Mathematik, Informatik und Naturwissenschaften

der Universität Hamburg

eingereicht am

Fachbereich Informatik

durch

**Mohammad Reza Heidari**

Hamburg, May 2022

**Gutachten:**

Prof. Dr. Thomas Ludwig

Prof. Dr. Ali Ebnenasir

**Datum der Disputation:** March 17th, 2023

## Abstract

This dissertation aims at providing a solution and support for optimizing the performance of the atmospheric model ECHAM6. The special prominence of this research is due to the application of the model in the German climate modeling initiative PalMod for simulating a complete glacial cycle from the last interglacial to the Anthropocene. The model, however, suffers from poor scalability at low resolution, as used in this paleoclimate study, due to the limited number of grid points. As a consequence, the potential of the existing high-performance computing architectures cannot be utilized for such experiments at full scale. Endeavors to adopt a higher optimized model is, thus, opportune for the PalMod research. Our investigation reveals that radiative transfer is a relatively expensive atmospheric process in ECHAM6, accounting for approximately 50% of the total simulation time. This current level of cost is achieved by performing radiation calculations only once every two simulation hours.

In response, this dissertation reports on a twofold research effort to alleviate such a computational burden in order to render the paleoclimate simulations viable. It first presents the idea of the concurrent radiation scheme for extending the available concurrency in ECHAM6 further by running the radiation component in parallel with other atmospheric processes. This solution also offers a way forward to improve the accuracy of the simulations by increasing the physical consistency between atmospheric states. To implement this scheme, a novel program analysis approach, i.e. Component Isolation, is then introduced for performing the following tasks: extracting a component from a Fortran program, detecting the input and output global variables of the component, making shared source code of the component non-shared. This approach not only benefits the implementation of the concurrent radiation scheme, but it also provides support for another optimization solution envisaged in PalMod.

Furthermore, the high accuracy of the approach of Component Isolation and the implementation of the concurrent radiation scheme is demonstrated using careful qualification and validation. Moreover, a thorough analysis is investigated to show the impact of the new radiation scheme on the performance of the atmospheric model ECHAM6. The experiments show that ECHAM6 can achieve a speedup of over 1.9× using the concurrent radiation scheme while becoming almost twice more scalable. The scientific results from the new scheme are, however, evaluated through an independent investigation by a climate scientist.

In the nutshell, the approach of Component Isolation offers an unprecedented solution for preparing any arbitrary components of scientific models (in Fortran) for a better organization scheme such as component concurrency or any individual optimization technique such as mixed-precision arithmetic in order to improve the performance of the model. In addition, the successful example of the concurrent radiation scheme in the atmospheric model ECHAM6 can encourage similar optimization research in scientific computing whenever scalability becomes challenging.

## Zusammenfassung

Ziel dieser Dissertation ist es, eine Lösung und Unterstützung für die Optimierung der Leistung des atmosphärischen Modells ECHAM6 zu entwickeln. Die besondere Bedeutung dieser Forschung ergibt sich aus der Anwendung des Modells in der deutschen Klimamodellierungsinitiative (PalMod) für die Simulation eines kompletten glazialen Zyklus von der letzten Zwischeneiszeit bis zum Anthropozän. Aufgrund der begrenzten Anzahl von Gitterpunkten verfügt dieses Modell über schlechte Skalierbarkeit bei niedriger Auflösung, wie es in dieser Paläoklimastudie gezeigt wird. Folglich kann das Potenzial der bestehenden Hochleistungsrechnerarchitekturen für solche Experimente nicht in vollem Umfang genutzt werden. Es besteht der Bedarf das Modell für die PalMod-Forschung höher zu optimieren. Unsere Untersuchung zeigt, dass der Strahlungstransport ein relativ teurer atmosphärischer Prozess in ECHAM6 ist, der ungefähr 50 % der gesamten Simulationszeit ausmacht. Dieses derzeitige Kostenniveau wird dadurch verursacht, dass alle zwei Simulationsstunden Strahlungsberechnungen nur einmal durchgeführt werden.

Um eine solche Rechenlast zu verringern und dadurch die Paläoklima-Simulationen realisierbar zu machen, wird in dieser Dissertation ein zweiteilges Lösungskonzept entwickelt. Sie stellt zunächst die Idee des gleichzeitigen Bestrahlungsschemas zum Erweitern der verfügbaren Parallelität in ECHAM6 weiter vor, indem die Bestrahlungskomponente parallel zu anderen atmosphärischen Prozesse ausgeführt wird. Diese Lösung verbessert auch die Genauigkeit der Simulationen, indem die physikalische Konsistenz zwischen atmosphärischen Zuständen erhöht wird. Um dieses Schema zu implementieren, wird dann ein neuartiger Programmanalyseansatz, genannt Komponentenisolation, eingeführt. Er umfasst die folgenden Aspekte:: Extrahieren einer Komponente aus einem Fortran-Programm, Erfassen der Eingabe und Ausgabevariablen der Komponente, und Trennung der gemeinsam genutzten Quellcodes der Komponente. Dieser Ansatz kommt nicht nur der Implementierung des simultanen Strahlungsschemas zugute, sondern bietet auch Unterstützung für eine andere Optimierungslösung, die in PalMod vorgesehen ist.

Darüber hinaus wird die hohe Genauigkeit des Ansatzes der Komponentenisolierung und der Implementierung des gleichzeitigen Strahlungsschemas durch sorgfältige Qualifizierung und Validierung demonstriert. Die Analyse untersucht, welche Auswirkungen das neue Strahlungsschema auf die Leistung des Atmosphärenmodells ECHAM6 hat. Die Experimente zeigen, dass ECHAM6 mit dem simultanen Strahlungsschema eine Beschleunigung von über 1,9x erreichen kann und dabei fast doppelt so skalierbar wird.

Der Ansatz der Komponentenisolierung bietet eine neuartige Lösung, um beliebige Komponenten wissenschaftlicher Modelle (in Fortran) für ein besseres Organisation-

sschema wie Komponentennebenläufigkeit oder beliebige individuelle Optimierungstechniken wie Mixed-Precision-Arithmetik vorzubereiten, und dadurch die Leistung des Modells zu verbessern. Darüber hinaus kann das Beispiel des simultanen Strahlungsschemas im Atmosphärenmodell ECHAM6 ähnliche Optimierungsforschung im wissenschaftlichen Rechnen anregen, wann immer die Skalierbarkeit ein Problem darstellt.

# Acknowledgments

I would like to express my sincere gratitude and appreciation first and foremost to Prof. Dr. Thomas Ludwig for his unceasing support and invaluable feedback ever since I applied for an admission to the PhD program, without which I would not have been able to complete my study.

I also feel a deep sense of gratitude towards Prof. Dr. Ali Ebnenasir, whose expert guidance was an essential and intangible asset to the correct direction of the static program analysis presented in this dissertation.

Moreover, I wish to thank Dr. Hendryk Bockelmann for giving me the opportunity to join the PalMod project and enabling this research. Without his help and wise guidance, the direction of this dissertation could not have been the same.

I would also like to extend my special thanks to Jörg Behrens, who was a great and consistent source of support throughout this research and implemented the interface to the YAXT library for the concurrent radiation scheme. His inspiring advice and overall insights augmented the quality of the performance analyses presented in this dissertation.

I should also pay tribute to Prof. Dr. Julian Kunkel and Prof. Dr. Michael Kuhn, who generously helped me to embark on this PhD study in the first place and were always willing to support enthusiastically.

Further, I am so thankful to Deutsches Klimarechenzentrum (DKRZ) and the Department of Informatics of Universität Hamburg as well as their members' staff, in general, for their wonderful support in this endeavor. In particular, I am extremely grateful for Ms. Anna Leffler's never-ending support, who patiently guided me through the submission process.

Last, but by no means least, I would like to sincerely thank Prof. Dr. Ahmad Zeinolebadi and his lovely family for their wonderful support during my PhD study.

Part of this research was funded by the Federal Ministry of Education and Science (BMBF) under grant number FKZ:01LP1515A.

This work used resources of the Deutsches Klimarechenzentrum (DKRZ) granted by its Scientific Steering Committee (WLA) under project ID bk0993 and ku0598.

# Contents

# 1. Introduction

*The revolution of numerical weather and climate modeling requires combined scientific and computational advances to be made. A deep understanding of the shortcomings and limitations as well as new opportunities in climate modeling and computing technologies is thus a prerequisite. This chapter provides a valuable insight into the (computational) challenges of (future) climate simulation which motivate the research conducted in this dissertation. It also sets the goals of this thesis clearly and defines the problems that will be addressed in the the following chapters.*

## 1.1. Climate Models

The term "*weather*" denotes the condition of the atmosphere at certain point in time which can change within minutes. In contrast, "*climate*" mainly describes the condition of weather at a specific location over long-time scales. *Climate change* is, therefore, the statistical deviation of the climate condition in the long-run, mostly due to human activities (Kalkuhl and Wenz, 2020; Meurant, 2012). Over the last few years, scientific tools and observations have been combined to confirm this fact. The tools for predicting future climates are *climate models*, which investigate the response of the climate system to various forcings. They are used for making climate predictions on seasonal to decadal time scales and for making projections of future climate over the coming century and beyond. Climate modeling can address important issues of future climate change, especially global warming and its impacts.

Models used in climate research range from simple energy balance models to complex *Earth System Models (ESMs),* requiring state of the art high-performance computing technology (Flato et al., 2014; Wang et al., 2010), and they are chosen based on the target research questions (Flato et al., 2014; Held, 2005). *Atmosphere–Ocean General Circulation Models (AOGCMs)* are used for studying climate system and predicting greenhouse gases and aerosol forcing (Flato et al., 2014). ESMs are the most comprehensive tools that were developed on *AOGCMs* to account for biogeochemical cycles and study the impact of external forcing on the climate system either in the past or in the future (Bonan and Doney, 2018; Flato et al., 2014).

#### Component organization in ESMs

Climate system is a multidimensional space–time process and consists of multiple components including atmosphere, ocean, land and sea ice. Each component

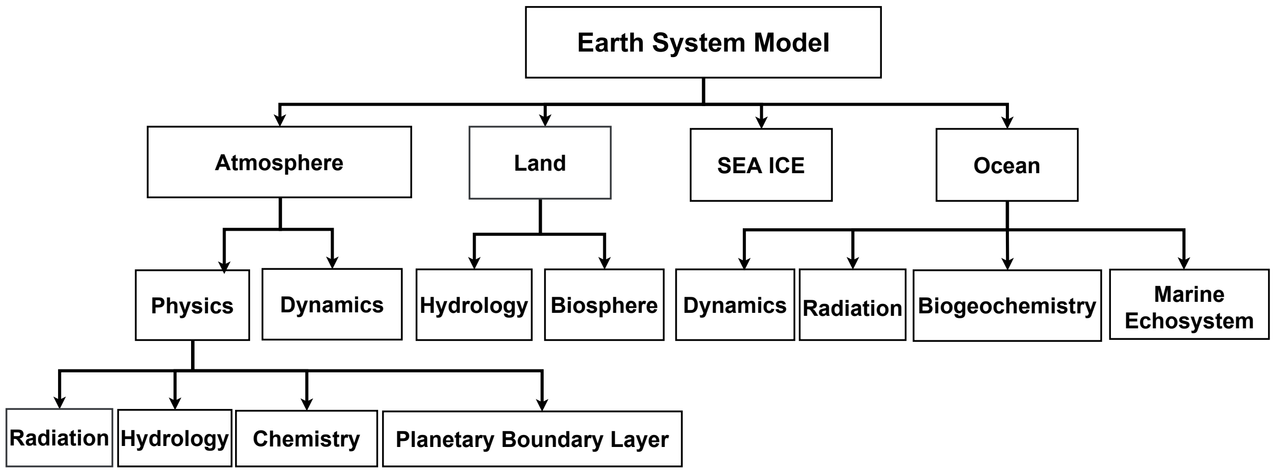


**Figure 1.1.:** A notional architecture of Earth System Model

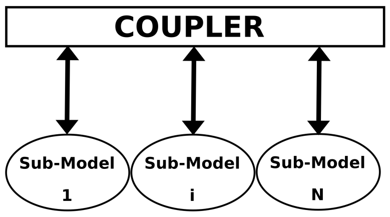


**Figure 1.2.:** An abstract architecture of a typical Earth System Model (ESM)

is driven by several physical processes. The notion of a physical component of the climate system is translated into a software component within climate models. Figure 1.1 shows the hierarchical component organization of a typical *ESM* (Balaji et al., 2016). The architecture of *ESMs* are quite diverse and they embody these components differently in their implementations (Balaji et al., 2016; Alexander and Easterbrook, 2015). Depending on the model, the components are organized inside one or multiple software programs. Generally speaking, an *ESM* is composed of various submodels that are separate (but mutually dependent) computer programs as shown in Figure 1.2. Each submodel is in charge of calculating a number of physical processes that have tight data dependencies to each other and solve within a smaller time steps. Due to the data dependency between components, submodels may exchange data at larger time steps through a coupler mechanism.

Figure 1.3 shows an overview of the components of the traditional Earth system model of *Max Planck Institute*, *MPI-ESM* (Müller et al., 2018; Giorgetta et al., 2013a),

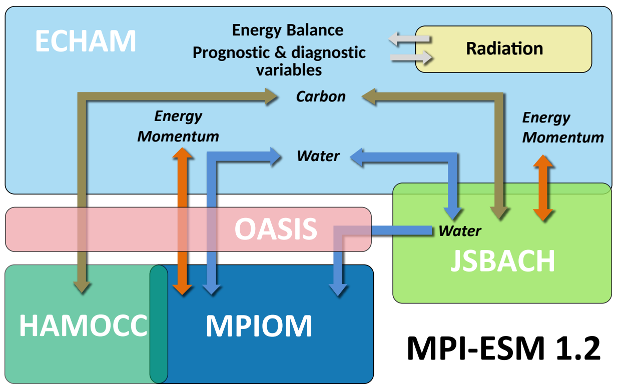


**Figure 1.3.:** MPI-ESM Earth System Model Architecture (courtesy of Deutsches Klimarechenzentrum (last access: 4 May 2022)), as described by Giorgetta et al. (2013a).

and their interactions . It is based on the components of *ECHAM* (Giorgetta et al., 2013b) for atmosphere and *MPIOM* (Jungclaus et al., 2013) for ocean as well as *JSBACH* (Lasslop et al., 2018) for terrestrial biosphere and *HAMOCC* (Ilyina et al., 2013) for the ocean´s biogeochemistry. The coupling of atmosphere and land on the one hand and ocean and biogeochemistry on the other hand is made possible by a separate coupling program called *OASIS* (Craig et al., 2017).

## 1.1.1. The Atmospheric Model ECHAM6

The atmospheric model plays a central role in every forecasting system and calculates the evolution of physical processes in the atmosphere. A stand-alone atmospheric model can be used to pursue specific research. Atmospheric modeling consists of two fundamental components: *dynamics* and *physics* (Rasp, 2019). The dynamics refer to the equations that represent the flow of air on a spatial grid and calculate the atmospheric states in every time step based on the previous time step (Gordon et al., 2016; Durran, 2010; Holton, 1973). Certain physical processes in the atmosphere such as radiation, however, take place locally on scales smaller than current grid resolutions of atmospheric models and thus cannot be calculated using finite differencing schemes. These processes are referred to as subgrid processes. Such processes play a pivotal role in the evolution of the model states, thus their effect on other physical processes that take place across the grid must be effectively approximated. These approximations are referred to as the "*parameterizations*" or, in general, the "physics" of atmospheric models (Rasp, 2019).

This dissertation concentrates on the performance optimization of *ECHAM6*, which is the sixth generation of the atmospheric general circulation *ECHAM* (Stevens et al., 2013; Giorgetta et al., 2013b). The model was developed at the *Max Plank Institute for Meteorology (MPI-M)* in Hamburg. It is the traditional atmospheric component of *MPI-ESM*, as indicated in Figure 1.3 and described by (Giorgetta et al., 2013a) . *ECHAM6* benefits from spectral and finite difference methods in five different grid resolutions, ranging from the coarse (*CR*) and low resolution (*LR*) to the very high resolution (*XR*). The *CR* or *T031* corresponds to a horizontal spatial resolution of 96×48 points in longitude and latitude while the *LR* or *T063* corresponds to 192×96 points (Stevens et al., 2013). *ECHAM6* is a parallelized and vectorized Fortran program in which the calculations can take place on several processors.

The special prominence of *ECHAM6* in this research is due to its application in the long-time paleoclimate simulations conducted within the German climate modeling initiative "*PalMod*" (PalMod webpage, last access: 17 January 2022), which is the main motivation to this dissertation. Paleoclimate Modeling is a branch of climate science whose research focus is on the prevalent climate conditions and natural climate change in the past. Lack of direct observations from the past climate is however a major obstacle in paleoclimate studies, and, thus climate scientists have to resort to the environmental records (or proxy data in climate science jargon) found in nature that cover different time periods (Sweeney et al., 2018). Paleoclimatology fosters a better understanding of how slow components in the Earth system operate. The new insights into the Earth system dynamics gained from paleoclimate research offers a practical technique for understanding the uncertainty in climate predictions and hence a useful means for testing and calibrating climate models (Sweeney et al., 2018; Stocker, 2014; Li et al., 2010; Haslett and Parnell, 2008).

### 1.1.2. Computational Challenges of Climate Modeling

One objective in climate research is to increase the realism of the simulated physical processes (Alizadeh, 2022; Washington et al., 2009). Despite the remarkable progress, climate modeling still suffers from stubborn errors (Rasp, 2019). And sizeable biases yet remain in climate predictions (Washington et al., 2009). These problems are caused by either our incomplete process understanding of the climate system or our modeling capability (Rasp, 2019). Climate has a chaotic nature which leads to uncertainty in climate modeling (Rasp, 2019). It is a complex system in which a vast number of fine-grained details interact (Wang et al., 2010). Hence, climate research needs to model a large number of physical processes on which the evolution of such a system depends. The level of details involved in modeling the climate system has a direct impact on the uncertainties inherent in the simulation results (Wang et al., 2010).

In addition, stochastic parameterization is a major contributor to errors in climate models (Rasp, 2019). This technique was introduced to better represent subgrid processes (that happen below the resolved scale). As observational studies reveal, a climate model must have a particular resolution to resolve some certain phenomena more realistically. There is a well-founded hope that increasing resolution enables climate models to replace physical parameterization with an explicit treatment and thus inherent uncertainty in climate change projections is gradually reduced (Schär et al., 2020).

Climate modeling is an immense task. It is compared to the simulation of the human brain and of the evolution of the early universe (Bauer et al., 2015). Due to the limited computing capabilities, climate models were initially quite simple. However, they have demonstrated a continual increase, in terms of vertical, horizontal and temporal resolution in parallel with the advancement in the computing industry (Lawson et al., 2021; Wedi, 2014). We are now capable of simulating the complex climate system at an unprecedented level of realism. The choice of the spatial and temporal resolutions affects the computational costs of climate simulations (Alizadeh, 2022; Wang et al., 2010). Doubling the horizontal resolution increases the computational cost by a factor of 24 (Rasp, 2019). To resolve crucial features more realistically, increasing resolution becomes an indispensable element of more accurate climate models at the expense of a higher computational power (Tabari et al., 2016; Washington et al., 2009). High resolution models can however become too expensive to be useful for long simulations. Since computational resources are limited, a compromise in spatial and time resolutions is usually made in order to perform realistic experiments in a reasonable time.

Due to the end of Moore's law (Theis and Wong, 2017; Mann, 2000), the future high-performance computing technology cannot continue the historical evolution as the energy cost becomes prohibitive and it has to be reduced (Bauer et al., 2015). Instead, the heterogeneous HPC systems are expected to combine massive parallelism (numerous low-power processors at lower clock rates) with classical CPUs (with large memory and a fast data interface). Challenges in science and technology are interdependent in many ways. The new paradigm in computing technology will exert a major impact on the design and development of future climate models. In particular, it is expected that the new trend will give rise to the adoption of higher scalable climate models (Bauer et al., 2015).

Although it is an old custom to adapt scientific challenges to the prevailing computational performance, the issue of higher scalability adds a new dimension to optimization of climate models. A mounting concern is aroused by the sequential organization of components within climate models which fundamentally restricts the benefit of mass parallelism in future heterogeneous HPC systems. Traditionally, climate models resolve physical processes sequentially and scalability is only achieved by taking advantage of domain decomposition and implementing data parallelism within internal components. With the emergence of new computing technology, this classical approach to concurrency seems insufficient in achieving scalability and, thus, the code design in climate models must be adapted in order to benefit from various computing architectures simultaneously. This requirement will pose a major challenge to legacy climate codes which contain millions lines of codes. As Peter Bauer states (Bauer et al., 2015), improving scalability is among the top priorities of weather and climate modeling in the next 10 years. A recent viable solution proposed by Balaji (Balaji et al., 2016) suggests achieving higher scalability through the re-organization of components in climate models and implementation of concurrency between higher and lower components. This approach allows model developers to offload various components into different accelerators and calculate multiple physical processes concurrently, thus improving the scalabilty of climate models. On this account, the following implications are expected:

- components may run concurrently on multiple resources
- components may run concurrently in shared or distributed memory model
- components may run concurrently on different architectures (such as CPU and GPGPU)

## 1.2. Motivation

This dissertation is inspired by two motivations, which are as follows:

- Primary motivation
- General motivation

### 1.2.1. The primary motivation

This research is primarily motivated by the performance optimization requirements of the PalMod initiative in the atmospheric model ECHAM6. PalMod aims at simulating a complete glacial cycle (i.e., about 120,000 years) from the last interglacial to the Anthropocene. There, however, remains a serious caveat as to the feasibility of such an ambitious project which should be acknowledged in advance. In particular, a major concern has been raised over the poor performance of ECHAM6 suffering from the limited number of grid points at the setups (including the CR spatial resolution) used in paleoclimate simulations. For this reason, the performance optimization of the model is instrumental in ensuring the viability of such long-time simulations. Hence, the following solutions have been proposed to reduce the negative impact of the poor performance of the model on the simulation time. These solutions are as follows:

- Solution I: Concurrent radiation calculations
- Solution II: Single-precision radiation calculations

These solutions target the calculation of radiative transfer in the model. In Chapter 3, it will be shown that the radiation component is one of the most expensive components in ECHAM6. Thus, reducing the impact of the high computational profile of this component will be a quantum leap towards expediting the paleoclimate experiments performed with this model. However, the solutions above are confronted with some inherent obstacles in the model which will be described individually.

#### 1.2.1.1. Solution I: Concurrent radiation calculations

In the classical radiation scheme, the model resolves the radiative transfer and other atmospheric processes sequentially, which prolongs the time-to-solution of the model dramatically. Solution I is based on the idea of *component concurrency* introduced by Balaji (Balaji et al., 2016) and suggests modifying the classical scheme and running the radiation component concurrently with the other components inside the model. A full account of this approach will be provided in Chapter 3. Briefly speaking, however, the concurrency is implemented using distributed memory and, due to the inherent dependency, the radiation component and the main model have to synchronize periodically during the course of simulation. Though frowned upon in software engineering, such dependency is created in part through some global variables that are shared between the radiation component and the main model. In the original model, components run in a shared address space and thus they have access to the same copy of a shared variable. In the concurrent scheme, however, the radiation component and the main model run in different address spaces and they, therefore. see different copies of the shared variables. To create a correct memory consistency, synchronizing different copies of the shared variables is absolutely necessary. There are, however, two main obstacles in this regard, which are as follows:

- unknown shared variables of the radiation component:
  It is remotely possible to find the shared variables between the radiation component and the main model without performing a thorough static code analysis. As a prerequisite, we need to have the entire source code of the component.
- the unknown scope of the radiation component:
  Although the component has been defined inside the model, the source code of the radiation component is not clearly known and a rigorous assessment is required to constraint its entire source code among the definitions of other components. This is due to the following problems:
  - code sharing between components due to cross-cutting concerns
  - code sharing between components due to sharing the same concerns
  - the large code base of the model (containing almost 167,000 lines of code), which hinders a quick analysis.

On this account, it is necessary to perform the following tasks before implementing a synchronization between the concurrent components:

- Extracting the radiation component from the model: A thorough (static code) analysis is required to find the entire source code of the component among other components in the large code base of model.
- Extracting the shared variables between the radiation component and the main model: An additional static code analysis is required to find the global variables that are shared between the radiation component and the rest of the model.

#### 1.2.1.2. Solution II: Single-precision radiation calculations

The second solution, on the other hand, suggests optimizing the model by applying single-precision arithmetic to the calculation of radiative transfer to reduce the required computations and thus the overall simulation time of the model. To implement this solution, the following steps are required:

- Converting calculations to single-precision:
  All the calculations (and thus variables) inside the radiation component are converted from high-precision to single- precision. Hence, a major code refactoring takes place across the component. This requires the source code of the component to be separated from the main model in advance to prevent any negative impacts on other calculations.
- Creating an explicit type-casting interface:
  As mentioned in Section 1.2.1.1, the radiation component and the main model exchange data through shared variables. Thus, an explicit type-casting (from high-precision to single-precision and vice versa) must be implemented to make sure that each part receives correct data. On this account, the prerequisite for this step is collecting the shared variables between the radiation component and the main model and creating a import or export data interface from or to the component.

### 1.2.2. The general motivation

The prerequisites for Solution I and II also extend to other components inside the atmospheric model ECHAM6 and even to other legacy climate models. This gives significant rise to the general motivation for this dissertation. As explained in Section 1.1.2, climate models are expected to demonstrate a continual increase in terms of complexities and spatial and temporal resolutions in the future, stimulating an enormous demand for higher computational resources. It was also discussed that a change of paradigm is expected to place much more emphasis on higher scalability in climate models due to heterogeneity and massive parallelism of future

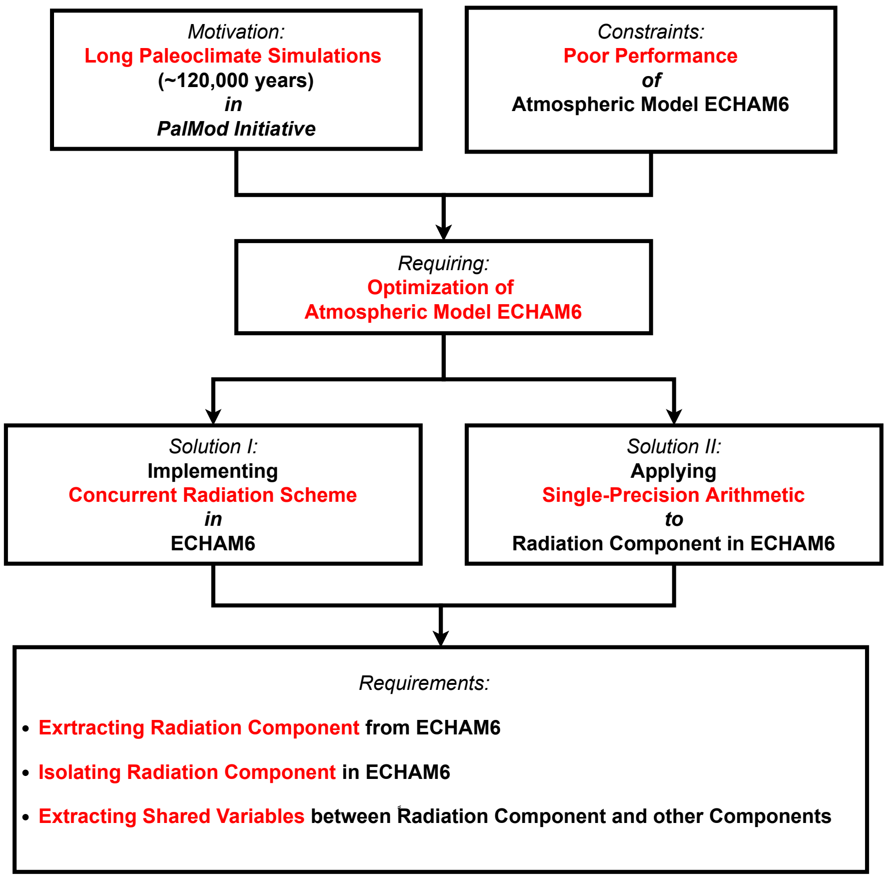

Motivation:
Long Paleoclimate Simulations
(~120,000 years)
in
PalMod Initiative
Constraints:
Poor Performance
of
Atmospheric Model ECHAM6
Requiring:
Optimization of
Atmospheric Model ECHAM6
Solution I:
Implementing
Concurrent Radiation Scheme
in
ECHAM6
Solution II:
Applying
Single-Precision Arithmetic
to
Radiation Component in ECHAM6
Requirements:
• Exrtracting Radiation Component from ECHAM6
• Isolating Radiation Component in ECHAM6
• Extracting Shared Variables between Radiation Component and other Components

**Figure 1.4.:** The primary motivation of this dissertation arises from the long-run paleoclimate simulations (around 120,000 years) within PalMod projects. The poor performance of the atmospheric model ECHAM6 is however a major concern as to the feasibility of such long simulations. The radiation component is an expensive component of the model. Single-precision arithmetic and concurrent radiation component are two solutions to reduce the impact of the high computational profile of the component on the simulation time. This requires the extraction of the component, separation of the source code of the component from the main model and the extraction of the shared variables between the component and the main model.

high-performance computing systems. As a result, it is expected various components will be offloaded to different architectures (which match best their computational profiles) and run concurrently in a distributed-memory model. This approach is, however, confronted with the same obstacles discussed in Section 1.2.1. Generally speaking, a component in legacy climate models is entitled to the following problems:

- sharing source code with other components
- the unknown scope of the component
- the unknown shared variables between the components

Since components are expected to run concurrently, the same prerequisite for Solution I is applied here as well. Additionally, the components are expected to run on different architectures available in heterogeneous HPC systems. Hence, the source code of each component must be adapted to the target architecture. This approach, therefore, resembles Solution II and requires the source code of the component to be separated from others to prevent any modifications from affecting other components. In the nutshell, the prerequisites for improving the scalability of future climate models are as follows:

- Extracting the shared variables between the components
- Isolating components from each other

Figure 1.5 summarizes the general motivation to this dissertation.

## 1.3. Goals of this Thesis

This dissertation strives for achieving multiple goals: the primary goals and the general goal.

### 1.3.1. The primary goals

As shown in Figure 1.6, the primary goals of this thesis are as follows:

- Primary Goal 1: building a new version of the atmospheric model ECHAM6 with *the isolated radiation component.*
- Primary Goal 2: building a new version of the atmospheric model ECHAM6 with *the concurrent radiation scheme.*

Primary Goal 1 prepares the ground for implementing Solution I and II (discussed in Section 1.2) by creating *the isolated radiation component* in the atmospheric model ECHAM6 and extracting the shared variables between the radiation component and other parts of the model. Roughly speaking, *isolating a component in a Fortran*

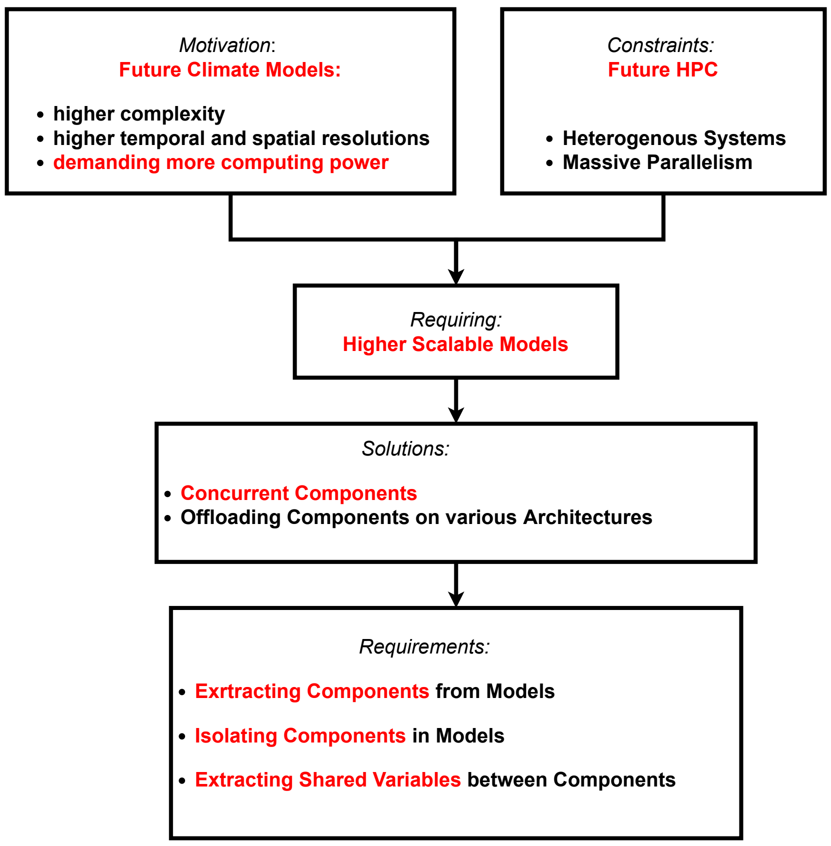

Motivation:
Future Climate Models:
• higher complexity
• higher temporal and spatial resolutions
• demanding more computing power
Constraints:
Future HPC
• Heterogenous Systems
• Massive Parallelism
Requiring:
Higher Scalable Models
Solutions:
• Concurrent Components
• Offloading Components on various Architectures
Requirements:
• Exrtracting Components from Models
• Isolating Components in Models
• Extracting Shared Variables between Components

**Figure 1.5.:** The general motivation for this dissertation arises from future climate models that are expected to become more complex with a higher temporal and spatial resolutions, requiring a higher computational power. Since future HPC systems will provide massive parallelism on heterogeneous technologies such as GPGPU, it requires higher scalable climate models in the form of concurrent components. The indispensable prerequisites to implementing such concurrency includes the extraction of components from models, separating the source code of components from models and extracting the shared variables between components.

*program* refers to a code refactoring practice that separates the source code of the component from the source code of the other parts of the program such that they share no source code any longer. Extracting the component from the program is, nevertheless, a prerequisite to make the source code of the component clearly known in advance. In Chapter 6, this procedure will be described precisely.

Primary Goal II is, however, to implement *the concurrent radiation scheme* in ECHAM6 using the results from Primary Goal I. In addition, *the single-precision radiation scheme* also benefits from Primary Goal 1, but its full implementation is beyond the scope of this dissertation. This solution was pursued by an external team albeit taking the advantage of the support of this dissertation.

### 1.3.2. The general goal

This dissertation also paves the way for improving the scalability of (legacy) climate models on future heterogeneous HPC systems. Thus, to fulfill the prerequisites discussed in Section 1.2, this thesis generally aims at an approach called *Component Isolation* for performing the following tasks on an arbitrary component in a Fortran program:

1. Extracting the component from the Fortran program.
2. Extracting shared variables between the component and the other parts of the program.
3. Isolating the component in the program.

Hence, the outputs of this approach are as follows:

- the shared variables
- *the isolated component*
- *the carvedout program*

The list of shared variables are the global variables shared between the component and the other parts of the program. In Chapter 6, we will define the isolated component and the carvedout program precisely. Roughly speaking, however, the isolated component is similar to the original component, but it does not share any source code with the other parts of the program. The carvedout program is almost the same as the original program, but it does not share any source code with the isolated component. Figure 1.7 depicts the general goal of this dissertation.

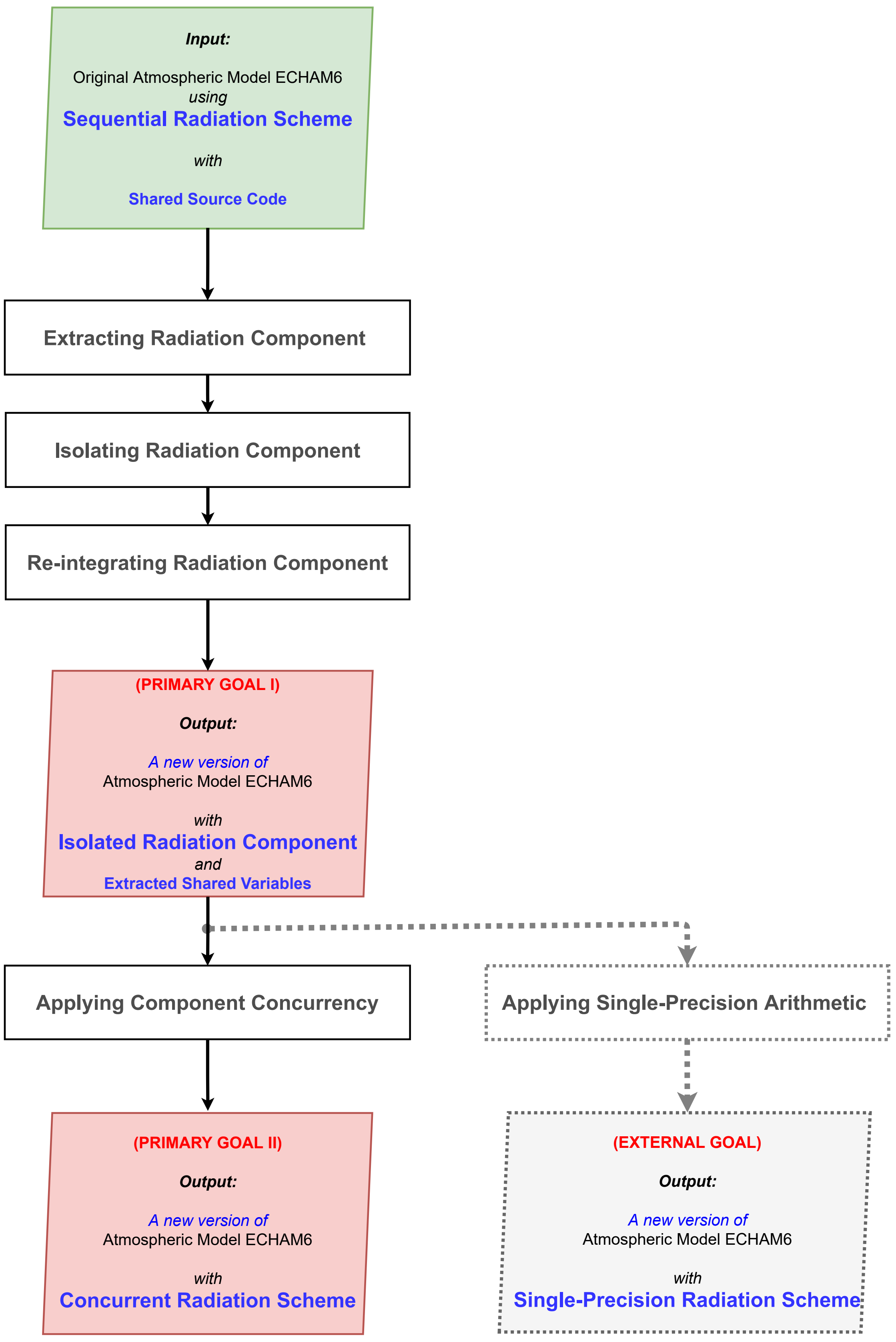

Input:
Original Atmospheric Model ECHAM6
using
Sequential Radiation Scheme
with
Shared Source Code
Extracting Radiation Component
Isolating Radiation Component
Re-integrating Radiation Component
(PRIMARY GOAL I)
Output:
A new version of
Atmospheric Model ECHAM6
with
Isolated Radiation Component
and
Extracted Shared Variables
Applying Component Concurrency
Applying Single-Precision Arithmetic
(PRIMARY GOAL II)
Output:
A new version of
Atmospheric Model ECHAM6
with
Concurrent Radiation Scheme
(EXTERNAL GOAL)
Output:
A new version of
Atmospheric Model ECHAM6
with
Single-Precision Radiation Scheme

**Figure 1.6.:** Primary Goal I is to create *the isolated radiation component* in the atmospheric model ECHAM6 and extract the shared variables between the radiation component and other parts of the model. This goal is achieved by extracting the radiation component from the model and separating its source code from the source code of the model. Primary Goal II is to implement *the concurrent radiation scheme* in ECHAM6 using the results from Primary Goal I. In addition, *the single-precision radiation scheme* is also created (by an external team) using the isolated radiation component.

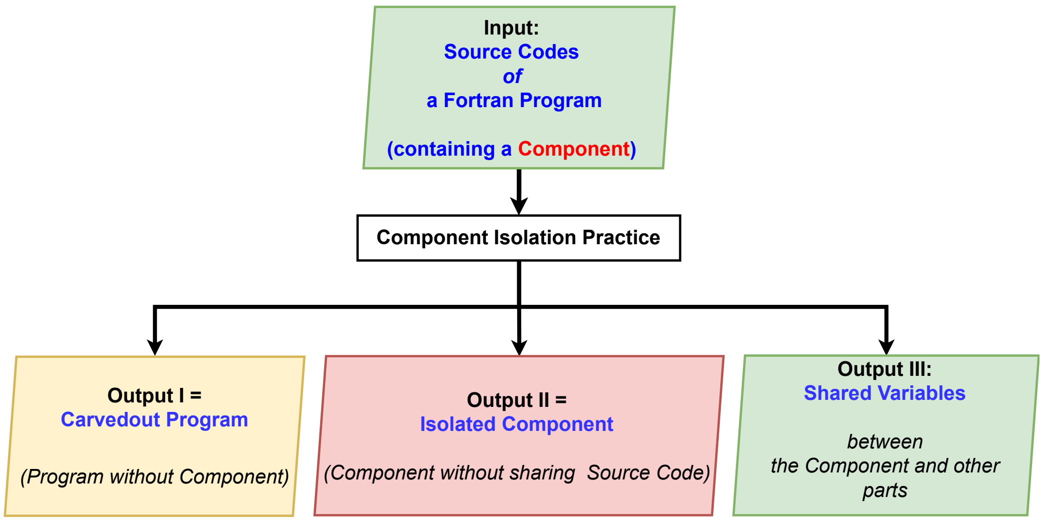


**Figure 1.7.:** The general goal of the dissertation is to introduce an approach called *Component Isolation* for extracting a component from a Fortran program, *isolating* it from the program and extracting the shared variables between the component and the program. The shared variables are the the global variables shared between the component and the Fortran program. The *isolated component* is the same as the original component, but it does not share any source code with the other parts of the program. The *carvedout program* will be defined in Chapter 6 precisely. Roughly speaking, however, it is the same as the original program without sharing any source code with the component.

## 1.4. Outline of the thesis

- Chapter 2 is a literature review of the similar works to this dissertation. It discusses the related research in two parts:
  - The first part provides an insight into the available concurrency inside the existing climate models.
  - The second part gives an overview of the software engineering research and tools that relate to the general goal of the dissertation. The chapter indicates the shortcomings of the previous works and justifies the work presented in this thesis.
- Chapter 3 describes the technique of the concurrent radiation scheme that was applied to the atmospheric model ECHAM6 within this dissertation. It will explain the importance of the component in the atmospheric simulation and describes why the radiative transfer is a real computational bottleneck. The chapter also discusses why the proposed solution should be a right choice to overcome the challenges confronting the primary motivation of this dissertation.
- Chapter 4 discusses the implementation procedure regarding Primary Goal 1 and Primary Goal 2.
- Chapter 5 provides a thorough performance analysis of the classical and concurrent radiation scheme of the atmospheric model ECHAM6 and highlights the achievements regarding Primary Goal 2.
- Chapter 6 describes a novel static program analysis approach to achieve the general goal of this dissertation.
- Chapter 7 provides some techniques for validating the implementation procedures (described in Chapter 4 and Chapter 6) to achieve the goals of this dissertation.
- Chapter 8 provides a summary of the discussions throughout the chapters and conclude the dissertation.
- Chapter 9 proposes a couple of follow-up research works to improve the solutions provided in this dissertation.

## 1.5. Chapter Summary

This chapter presented the (general and primary) motivations and goals of this dissertation. It was stated that climate models are the means for studying climate change, but more accurate simulations require more optimized models which can benefit from the current and future heterogeneous high performance computing (HPC) systems. Such optimization in legacy climate models is, however, hindered by sharing source code between different components inside the models. Removing such a problem is, therefore, the general motivation of this dissertation. Hence, the general goal of the dissertation is as follows:

- Presenting a novel approach for performing the following static program analysis tasks:
  1. Extracting a component from a Fortran program.
  2. Extracting shared variables between the component and the other parts of the program.
  3. Isolating the component in the program.

The primary motivation of this dissertation is, however, to optimize the radiation scheme of the atmospheric model ECHAM6 in order to expedite the paleoclimate simulations required by the PalMod project. Hence, Primary Goal 1 of this dissertation is as follows:

- Building a new version of the atmospheric model ECHAM6 with the isolated radiation component.

In this new version of the model, the shared source code and variables of the radiation component and the other parts of the model are separated from each other. Primary Goal 2, on the other hand, aims at the following goal:

- Building a new version of the atmospheric model ECHAM6 with the concurrent radiation scheme.

In this new version of the model, radiative transfer is calculated concurrently with other atmospheric processes in order to improve the scalability of the model. Finally, the external goal of this dissertation is as follows:

- Building a new version of the atmospheric model ECHAM6 with the single-precision arithmetic radiation scheme.

In this version, a single-precision arithmetic scheme is applied to the calculations of radiative transfer by benefiting from the results from Primary Goal 1. However, this goal is beyond the scope of this dissertation and is pursued by an external project.

# 2. Background and Related Works

Einstein:
*A hundred times every day, I remind myself that my inner and outer life depends on the labors of other men, living and dead, and that I must exert myself in order to give in the measure as I have received and am still receiving.*

*This chapter provides a solid background and the related works to the discussions in this dissertation. The chapter thus is divided in two parts: the first part presets the topics concerning the primary goals and the second part is dedicated to the general goal of the dissertation.*

## 2.1. Background and related works to the primary goals

It was explained in Section 1.3.1 that ECHAM6 suffers from low scalability in paleoclimate simulations (pursued in the PalMod project) due to the limited number of grid points at the CR resolution. The primary goal of this dissertation is to implement a concurrent radiation scheme to increase the level of concurrency in the model. Hence, this chapter first gives an overview on the available concurrency in ECHAM6 and then describes the idea of *Coarse-grained Component Concurrency (CCC)* proposed by Balaji (Balaji et al., 2016) as the basis of the solution proposed by this dissertation. Before delving into details, however, it should be helpful to clarify the difference between two recurring terminologies in this dissertation here: "concurrency" and "parallelism".

#### Concurrency vs. Parallelism

Concurrency denotes the order of the execution of multiple tasks (Grossman and Anderson, 2012). Concurrent tasks can start, run, and complete in any order (Wikipedia, last access: 31 March 2022a; Kreowski, 1986) even on a single processor. The results of the concurrent tasks do not change if they execute in different orders as they do not have any dependency (Shatnawi et al., 2017; Wilde, 1990) on each other. Parallelism, on the other hand, denotes simultaneous execution of multiple concurrent tasks on multiple processors. In other words, if task A and B are concurrent and they run at the same time on different processors, they are also parallel tasks.

### 2.1.1. Concurrency in ECHAM6

Climate models historically implement concurrency at different levels and leverage various parallelism techniques to reduce time-to-solution of climate simulations. ESMs, in general, benefit from both fine and coarse-grained concurrency and enact them within a combination of data parallelism and task parallelism paradigms. Task and data parallelism are two fundamental parallel programming paradigms and mixing them often yields better speedups in handling expensive computational applications (Wikipedia, last access: 31 March 2022b; Suter, 2007). Providing the implementation support for the mixed-parallel computing at various levels has already been addressed in the literature (Khaldi et al., 2012; Aida and Casanova, 2009; N'Takpe et al., 2007; Radulescu et al., 2001; Radulescu and Van Gemund, 2001; Bal and Haines, 1998). MPI-ESM is a good example of climate models that benefit from such a hybrid computing paradigm, which will be explained further below.

#### Task Parallelism in MPI-ESM

Introduction of the multi-processor technology in 1980s allowed for the simultaneous execution of multiple programs on several processors. This became a milestone in parallel computing as it enabled task parallelism in form of coarse-grained concurrency for the first time and led to the MPMD (multiple-program, multiple-data) parallel framework (Wikipedia, last access: 17 January 2022; Balaji et al., 2016). In MPMD, multiple autonomous processors execute several programs simultaneously. The emergence of distributed-memory computing gave rise to climate models which combine fine-grained and coarse-grained concurrency. Most of the ESMs in the world today are MPMD applications that allow multiple submodels to run concurrently and exchange data through a coupler mechanism as shown in Figure 1.2. MPI-ESM implements task parallelism using the MPMD framework between the atmospheric model ECHAM6 and the ocean model MPIOM. ECHAM6 and MPIOM are composed of several components, each of which calculate some physical processes. The land model (JSBACH) is now part of ECHAM6. By the same token, the biogeochemical model (HAMOCC) is part of MPIOM. The components in ECHAM6 and MPIOM are organized sequentially, but they implement data parallelism internally.

#### Data Parallelism in ECHAM6

Data parallelism in ECHAM6 is implemented at three levels: SIMD, OpenMP threads and SPMD. These levels are explained below:

- Level I:
  The first level of data parallelism in ECHAM6 is implemented using the SIMD (single-instruction multiple data) architecture (Wikipedia, last access: 17 January 2022) (also known as vectorization) of modern processors in program loops and arrays of data (Balaji, 2015). SIMD was the first introduction of fine-grained concurrency in which the same sequence of instructions could be

applied to each element of a data stream simultaneously (Balaji et al., 2016) (Balaji, 2015).

- Level II:
  Further data parallelism in ECHAM6 is achieved through OpenMP programming model by generating parallel threads that run on shared-memory architectures (i.e. multi-processors, multi-core machines) and divide loops and partition large data streams between multiple parallel threads and processors.

- Level III:
  The third level of data parallelism in ECHAM6 is implemented using domain decomposition (DDM webpage, last access: 4 April 2022). It discretizes the total physical input space and divides it into multiple separate domains. ECHAM6 is an example of SPMD (single-program, multiple-data) applications (Wikipedia, last access: 4 April 2022; Shipman, 2016), in which multiple autonomous processors simultaneously execute the same program on partitions of a data stream. The model leverages Message Passing Interface (MPI) (MPI official webpage, last access: 4 April 2022; Nielsen, 2016) and spawns several MPI processes to calculate each domain on a different processor across a distributed-memory machine.

### 2.1.2. Coarse-grained Component Concurrency: A new level of parallelism

In the previous section, it was explained that the scalability of a model is affected by the level of concurrency that is available within the model. There are several motivations for increasing the level of concurrency in climate models. For example, low-resolution simulations normally suffer from a low scalability due to the limited number of grid points. In addition, as discussed in Chapter 1, climate models need to achieve a higher scalability in order to cope with the future scientific and technological challenges.

On this account, Balaji suggests a new approach called *coarse-grained component concurrency (CCC)* to increase the rather modest amount of concurrency among ESM components (Balaji et al., 2016). He suggests that, from 10 decisive factors in achieving the goal of future high scalable climate models, one comes from component reorganization (Balaji et al., 2016). Coarse-grained concurrency can be implemented inside each submodel by running lower or higher level components in parallel with each other.

It was pointed out earlier that components of a submodel traditionally run sequentially in respect to each other (as shown in Algorithm 3.1). Since the physical processes are calculated one at a time in this scheme, each component receives feedback from previous calculations before it takes turn to run in every time step. As a result, every component has a direct contribution to the overall runtime of a submodel and fast components have to wait long for slow components. To reduce the long simulation time, however, coarse-grained component concurrency suggests resolving multiple physical processes simultaneously by running several (expensive) components concurrently. As a result, this approach increases the available concurrency in ESMs by implementing more task parallelism within submodels. Traditionally, task parallelism is mainly available between submodels, but coarse-grained component concurrency extends it to the components within submodels as well. In addition, if the concurrent component has a higher scalability, it can potentially adopt finer domain decomposition and allocate a larger number of parallel processes. As a result, this feature can increase the scalability of the whole model proportionally.

### Components coupling

Coarse-grained component concurrency may sound non-intuitive as there is a tight coupling dependency between components of the climate system. Traditionally, climate models resolve the physical processes sequentially in respect to each other to prevent instability in the model. Consider the simplest case in which two components CompX and CompY run sequentially and have a dependency on each other at the boundary, as shown in Figure 2.1. The call sequence of the component can thus be schematically represented as below:

$$\mathrm{CompY}^{t+1} = \mathrm{CompY}^{t} + f(\mathrm{CompX}^{t}, \mathrm{CompY}^{t})$$
$$\mathrm{CompX}^{t+1} = \mathrm{CompX}^{t} + g(\mathrm{CompY}^{t+1}, \mathrm{CompX}^{t})$$

where f() and g() represent the feedback from the other component, and the superscript represents a discrete time step. In this scheme, CompX is able to access the updated state of $\mathrm{CompY}^{t+1}$ in the second step. As described by Balaji (Balaji et al., 2016) and other text books on numerical computing (e.g. (Durran, 2013)), this is formally equivalent to Euler forward-backward time integration or Matsuno time stepping.

In a parallel scheme, the components however execute concurrently (as shown in Figure 2.1) and CompX has access only to the lagged state $\mathrm{CompY}^{t}$. The coupling between the parallel components can be described as below:

$$\mathrm{CompY}^{t+1} = \mathrm{CompY}^{t} + f(\mathrm{CompX}^{t}, \mathrm{CompY}^{t})$$
$$\mathrm{CompX}^{t+1} = \mathrm{CompX}^{t} + g(\mathrm{CompX}^{t}, \mathrm{CompY}^{t})$$

The new scheme, however, introduces a change in the operator splitting technique and the potential effects of which needs to be systematical assessed as the results will not be identical to the sequential case any longer. A general formal stability analysis requires the forms of f and g to be available (Balaji et al., 2016). Although this coupling algorithm is formally unconditionally unstable (like the Euler forward

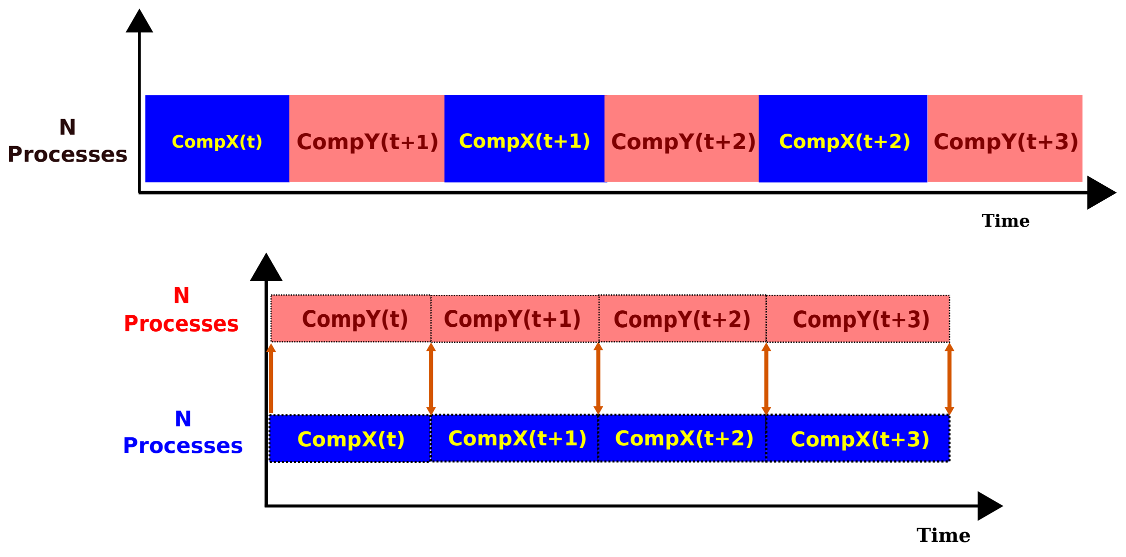


**Figure 2.1.:** Sequential and concurrent coupling sequences between two components (CompX and CompY) in a climate modeling, with time on the X-axis and processors on the Y-axis. Note that, in the sequential coupling sequence (on the top), CompX$^{t+1}$has access to the update state CompY$^{t+1}$. However, in the concurrent coupling sequence (at the bottom), CompX$^{t+1}$only has access to the lagged state CompY$^{t}$.

method), it works in practice in climate modeling (Balaji et al., 2016). As Balaji states (Balaji et al., 2016), this is a good example to indicate the opportunistic nature of performance engineering that tries to take advantage of a successful practical solution even though theoretically unfeasible. This is due to the inherent sources of stability within the climate system as well as some computing methods aimed at reducing instability (Balaji et al., 2016). This issue is nevertheless a good motivation for further investigations.

### Properties of the concurrent component

A suitable candidate for coarse-grained component concurrency in a legacy climate model should have the following properties:

- The candidate component is expensive, i.e. it is much slower than the rest of the model.
- The expensive component is configured to run at a different timescale than the rest of the model in order to reduce the negative impacts of its high computational profile on the overall simulation time. In this scheme, the component is executed at a larger time step ($Ts_{large}$) and will not be called

during multiple shorter time steps ($Ts_{short}$) in which other physical processes are resolved. $Ts_{large}$ is usually multiple times larger than $Ts_{short}$ (i.e. $Ts_{large} = n * Ts_{short}$).

- $Ts_{large}$ ,however, creates inaccuracies in the model. This is because the results from the expensive component is not updated during the shorter time steps ($Ts_{short}$). As a result, the other components use a lagged state of the expensive component during the shorter time steps, which leads to inaccuracies in simulations. However, using a larger time step requires re-tuning of the model to achieve the required accuracy.
- Nevertheless, the candidate component is far more scalable than the rest of the model, but the sequential organization of the main model prevents this benefit by enforcing a similar domain decomposition setup across all components.

#### Advantage of coarse-grained component concurrency

Considering the properties of a candidate component that was mentioned above, coarse-grained component concurrency can bring the following benefits:

- Since the other components do not expect updated results from the expensive components during the shorter time steps, this provides an ample opportunity to run the expensive component in parallel with the other calculations. In consequence, the model does not wait long for the expensive component when the next $Ts_{large}$ arrives, thus expediting the overall simulation time.
- In addition, the concurrent scheme can give the expensive component freedom to scale independently from the main model. Hence, the expensive component can benefit from finer problem decomposition and allocate more computational resources independent from the main model. This property improves the scalability of the model, which can potentially reduce the overall simulation time.
- The last but not the least, coarse-grained component concurrency is an opportune to remove the discrepancy between $Ts_{large}$ and $Ts_{short}$ (i.e. $Ts_{large} = Ts_{short}$) and executing the expensive component preferably in every time step. This can happen by assigning enough resources to the expensive component.

## 2.2. Background and related works to the general goal

This section presents a background and the related works regarding the general goal of this dissertation. In Chapter 1, it was explained that the general goal of this dissertation includes the following tasks:

1. extracting a component from a Fortran program
2. isolating the component in the program

3. extracting the shared variables between the component and the other parts of the program

As far as an extensive search by this dissertation is concerned, neither previous works nor tools address the above tasks fully. However, there are some similar efforts in the past and some helpful tools that can assist regarding the general goal of this thesis.

## 2.3. Reusable Component Extraction

In software engineering, software component reuse denotes the search for components that provide required functionality for a new software application (Thapar and Sarangal, 2020; Singh and Tomar, 2014). The cost of developing software from scratch and maintaining can be reduced by identifying and extracting the reusable components from legacy software (Ampatzoglou et al., 2018; Ahmaro et al., 2014). The authors in (Gholamshahi and Hasheminejad, 2019) provide an overview of research in this regard. Some of the most relevant works are, however, presented below and their difference with our work is explained. We refer to these papers as P1, P2, P3 and P4 for a later discussion.

### P1: Conceptual Module

Authors in (Baniassad and Murphy, 1998) introduce an approach (and a tool called Conceptual Module) to transform an arbitrary set of lines of code (selected by the user from a C program) to an independent component. The entry point of the component is thus formed as a logical unit around these lines. The dependencies of the entry point on the original source code are then explored using the SUIF(Wilson et al., 1995) compiler's intermediate representation (IR) of a multifile software system. These dependencies include variables, subprograms and etc. that are used inside the entry point but their definitions are somewhere else in the original code. This approach generates only one slice, which is the extracted component.

### P2: Extract Component

The authors in (Washizaki and Fukazawa, 2005) present an approach (and a tool called Extract Component) to extract all possible reusable (independent) components from a Java source code for reuse in other applications. The tool builds a dependency graph called Class Relation Graph (CRG) and uses the reachability techniques to collect all possible clusters that have no dependency on the elements outside the cluster. The tool modifies the original program to use the extracted components.

### P3: ComponentExtractor

Authors in (Marx et al., 2010) report an approach (and a visual development tool called ComponentExtractor) that interactively supports developers to extract a (dependent) component from Java source codes. Although this is not a fully automated tool for component extraction, it visualizes the dependencies between the component and the original program and guides the developers to collect the codes of the component from the program step by step. The output of the process of component extraction is two slices of the original program: the slice of the extracted component and the slice of the remaining parts of the original program. The tool uses BCEL library (Byte Code Engineering Language) (Cap, 2013), which works on Java byte code, to collect the dependency information (albeit overestimated). The extracted component will be remained dependent on the second slice (i.e. the rest of the program), but the goal is to partition the original program in such a way that the dependency between the two slices becomes minimal. This approach is useful for outsourcing a component (of a large application) to an external team for further development and create a minimum interaction between the internal and external developers.

### P4: Kernel GENerator (KGEN)

Authors in (Kim et al., 2016) introduce a Python-based open source tool (called Kernel GENerator (KGEN)) to extract a computational kernel from a Fortran application and run it as a standalone executable. The motivation for this research is providing an opportunity for the independent optimization of computational kernels without dealing with the complexity of running the main application. KGEN generates only one slice of the original program which is the extracted component. The tool picks (at least) one Fortran subroutine and collects all the source code supporting the subroutine using a static dependency analysis on Abstarct Syntax Tree (AST) (Jones, 2003) of the original program. KGEN captures the input and output data to the subroutine as well as the state of the global variables while the original application is running. The extracted kernels may carry wrong pieces of software from the main application though it may not affect the target objectives of KGEN.

### Discussions

The general goal of this dissertation has similarities and major differences with the methods discussed in P1, P2, P3 and P4. The main features of our approach are as follows:

- Our approach is aimed at generating two slices from a Fortran program (which contains a component). These two slices are denoted by the *isolated component* and the *carvedout program*.
- Novelty I:
  Both slices are independent from each other, meaning that they compile successfully as stand-alone codes.
- Novelty II:
  Our approach separates shared source code between the component and the program by creating mirror copies of the shared parts for each slice.

| Features | P1 | P2 | P3 | P4 | Dissertation |
|---|---|---|---|---|---|
| **Target Programming Language** | C | Java | Java | Fortran | Fortran |
| **Number of Extracted Slices** | 1 | >2** | 2* | 1 | 2* |
| **Extracted Component Compilable** | Yes | Yes | No | Yes | Yes |
| **Carvedout Program Compilable** | —[1] | —[1] | No | —[1] | Yes |
| **Integrating Carvedout Program & Component** | —[1] | —[1] | No | —[1] | Yes |
| **Extraction Granularity** | Statement | Class | Class | namespace | Combined[2] |
| **Mirroring Shared Namespaces** | —[1] | No | No | —[1] | Yes |
| 1: Not applicable as the carvedout program is not extracted. | | | | | |
| 2: Extraction at both namespace and statement level. | | | | | |
| *: One slice is the isolated component and one slice is the carvedout program. | | | | | |
| **: The tool can extract multiple components and modifies the original code to re-use the components. | | | | | |

**Table 2.1.:** Comparing four component extraction methods (in P1, P2, P3 and P4) with the proposed solution in this dissertation.

- Novelty III:
  Our approach also extracts the shared variables between the two slices.
- The motivation to our approach is to give freedom to the developer to modify a component without affecting the other parts of the original program (through the shared source code). For example, the mixed precision arithmetic can be applied to the component or it can be adapted to the required modifications for offloading some computations on accelerators without affecting the other parts of the original program.

A comparison between our approach regarding the general goal of this dissertation with the methods discussed in P1, P2, P3 and P4 is presented in Table 2.1. Note that in this dissertation, we refer to the definition of a Fortran module or a Fortran procedure in a Fortran program as a Fortran namespace.

## 2.4. Program Analysis

In software engineering, a software program can be analyzed statically or dynamically. Static program analysis denotes an analysis without an actual execution of the target program as opposed to dynamic analysis that is performed when the program is running (Kaur and Nayyar, 2020; Egele et al., 2008). Static program analysis usually implies that the code analysis is conducted using automated tools during the assessment process. A code analysis, in general, pursues a wide variety of goals and thus requiring different techniques and tools to be able to perform the required tasks. In Chapter 6, we introduce an approach to achieve the general goal

| | Forcheck | plusFort | Flint | Understand | Phasar |
|---|---|---|---|---|---|
| **Call Graph** | Yes | Yes | Yes | Yes | Yes |
| **Cross-Reference Table** | Yes | Yes | Yes | Yes | Yes |
| **(Automatic) Component Extraction** | No[1] | No[1] | No[1] | No[1] | No[1] |
| **(Automatic) Isolated Program** | No[1] | No[1] | No[1] | No[1] | No[1] |
| **(Automatic) Shared Variables Extraction** | No[1] | No[1] | No[1] | No[1] | No[1] |
| **1: Lots of post processing (using the call graph and cross-reference table) is required.** | | | | | |

**Table 2.2.:** Five static program analysis tools in Fortran that can help with the approach introduced in Chapter 6 to achieve the three tasks (extracting a component, isolated program, extracting shared variables) of the general goal of this dissertation.

of this dissertation in multiple steps. There are some static code analysis tools that can however be potentially helpful in implementing some of these steps. These tools are as follows:

- Forcheck
- Cleanscape FortranLint
- Understand
- PlusFort
- Phasar

None of these tools nonetheless offers a fully-automated implementation of the approach proposed in Chapter 6. These tools can mainly provide the call graph as well as the cross-reference table of a Fortran program. As shown in Table 2.2, such information can be used to do a considerable post-processing to perform the three task tasks of the general goal of this dissertation based on the algorithm described in Chapter 6.

### Forcheck

Forcheck (Codework webpage, last access: 17 January 2022) is mainly aimed at locating bugs and generating reliable codes. It performs a static analysis on Fortran programs to get a fast insight into the code at various stages of the development process and verify the conformance to the Fortran standards. It can compose optimal documentations of a project with cross-reference tables of each program unit as well as a call-tree. Such information can be helpful in various steps of extracting a component from a Fortran program as well as collecting the shared variables between the component and the program.

### FortranLint (Flint)

FortranLint (Flint)(Cleanscape webpage, last access: 17 January 2022) is a proprietary Fortran static source code analysis tool from Cleanscape. It is mainly aimed at automatically identifying problems in Fortran codes prior to compiling or executing programs and has 1000+ analysis of F77-F03 source code (PhASAR webpage, last access: 17 January 2022). Flint is useful for refactoring and source browsing and generates different reports including the call tree and cross reference table of the program. The call tree is composed of a graphical diagram that shows the calling structure of the source code. Cross reference table displays every symbol used in the Fortran program in a table along with how it is used on each line of code. The cross reference also implements powerful filter capabilities to exclude variables that should not be visible in the results (for example, all single-letter variables, which are often just loop indices or other temporary variables). Such information can be helpful in various steps of extracting a component from a Fortran program as well as collecting the shared variables between the component and the program.

### *plus*Fort

plusFORT (Codework webpage, last access: 17 January 2022), from Polyhedron Solutions, is a multipurpose suite of tools for analyzing and improving Fortran programs. It can generate a call tree and a comprehensive cross-reference links, which can be helpful in extracting a component from a Fortran program as well as collecting the shared variables between the component and the program.

### Understand

Understand (SciTools, last access: 17 January 2022), designed and maintained by scitools company, is a proprietary integrated development environment (IDE) and platform for the static program analysis of large code bases written in multiple languages. It mainly provides supports for automatically reducing unnecessary or risky interdependencies during code modifications. Detailed cross-referencing is the core of Understand. This tool generates a bi-directional reference (such as "depends on" and "depended on by") for every name in a program and creates different types of references such as "Calls/CallBys" and "Include/Includeby". Such information can be used to visually track the side effects of every change that are made to a source code. The call-graph of a program can also be extracted using the detailed cross-reference. However, this tool does not provide a fully automated solution to the task of the general goal of this dissertation. It can nonetheless be helpful with the approach introduced in Chapter 6.

### PhASAR

PhASAR (PhASAR webpage, last access: 17 January 2022) is a LLVM based (LLVM webpage, last access: 17 January 2022; Lopes and Auler, 2014; Lattner, 2008) static analysis framework that allows for solving data-flow problems based on the LLVM intermediate representation (IR). It offers algorithms to compute points-to, type hierarchy, call-graph, and data-flow information. To generate the call graph of a Fortran program, the following steps are required:

- **Generating intermediate representation code:** It is necessary to generate the LLVM intermediate representation (LLVM IR) of the Fortran program in the first step. In practice, there is a tool called WLLVM(Ravitch, 2022) that can help in this regard.
- **Finding reachable methods:** Once the IR code is available, it should be possible to extract the call graph of the program and the component. In order to find which subroutines and modules are needed by the program and the component, PhASAR provides a call graph algorithm that implements all the means to find out which methods are reachable starting at the specified entry points of the program or the component.
- **Methods demangling:** Each of the reachable methods that the call-graph algorithm determines must then be demangled (use the c++filt tool) to find the actual names of the subprograms in the Fortran program.

The resulting call graph can be used in Section 6.5.1.1 to extract the code coverage of the component. This however requires a further processing. For example, a shell script should be used to grep for the definition of each of the resulting subprogram names on all of the modules of the program in order to find the namespaces of the target component. As this procedure clearly shows, PhASAR does not automatically provide a solution to the tasks of the general goal of this dissertation.

## 2.5. Program Slicing

Program slicing is a source-to-source transformation technique. A program slicer extracts those statements that potentially affect some target variables at a particular point of the original program. In other words, program slicing filters the program statements that are not relevant to a chosen computation called a slicing criterion, which is composed of some target variables at a particular point of the program (Nguyen et al., 2015; Gallagher and Binkley, 2008; Weiser, 1984). A slice of a program is thus the remaining statements afterwards. The original program and its slice generate the same output regarding the target variables at the target point (Mastroeni and Zanardini, 2017; Weiser, 1984). The prerequisite to program slicing is the slicing criterion. User must know the detail of the target program in advance to specify the target variables. Therefore, this technique does not provide a complete solution to the general goal of this dissertation as the source code and thus the target variables of the target component are considered unknown to the developer. In Chapter 6, we introduce an approach that does not require such details (e.g. role/intention of the variables of the target component) to be known in advance. However, some techniques of program slicing can assist with the implementation of this approach in Section 6.5.4.4. These techniques are as follows:

- Interprocedural slicing (Masud and Lisper, 2021; Asăvoae et al., 2014)
- Dynamic slicing (Lin et al., 2018; Sasirekha et al., 2011)
- Backward slicing (Srinivasan and Reps, 2016)

There are several program slicing tools, some of which have been summarized in Table 2.3. Despite an extensive search, this study could, however, detect only one program slicer for Fortran source codes as described below.

**Fortran Program Slicer Schatz:**

Schatz (Hoffner, 1995) is a program slicer that works on Fortran source codes and was developed by Ottenstein and Ottenstein on PDGs (Ito, 2018; Makka and Sagar, 2016; Ottenstein and Ottenstein, 1984) to study program slicing. It has a very simple user interface and works on a graph representation of the code. Schatz is intended to work with PAT which is an interactive FORTRAN parallelizing assistant tool (Smith and Appelbe, 1989; Smith, 1988). The following problems are, however, the main obstacles to use Schatz for the general goal of this dissertation:

- **Only intraprocedural slicing**. The tool offers only intraprocedural slicing. However, the target Fortran applications of this dissertation (i.e. climate models) contain subprograms massively and thus they require the interprocedural slicing technique.
- **No side effect:**. Subroutine calls must be free of side effects and are not expected to modify their parameters. However, our problem must deal with shared variables between subroutines.
- **Not available:** As far as this dissertation is concerned, this tools is not available.

## 2.6. Chapter Summary

This chapter presented the background and related works concerning the goals of this dissertation. It was explained that different levels of parallelism traditionally exists in the earth system models (ESMs). Similarly, MPI-ESM benefits from task parallelism between its submodels, namely the atmospheric model ECHAM6 and the ocean model MPIOM. In addition, ECHAM6 takes advantage of data parallelism by implementing domain decomposition across its components. The primary goals of this dissertation are, however, in pursuit of adding a new level of parallelism in the model by applying component concurrency to the radiation component.

Furthermore, this chapter presented the similar research and available tools regarding the general goal of the dissertation. In particular, it was shown that the majority

| **Program Slicer Tools** | **Language** | **Slicing Direction** | **Type of information** | **Scope of Slicing** |
|---|---|---|---|---|
| **Schatz**[1] | **FORTRAN** | Backward | static slicing | Intraprocedural |
| **Giri**[2] | **LLVM** | Backward | static slicing | Intraprocedural |
| **CodeSurfer**[3] | **C** | Backward | Dynamic | Interprocedural |
| **Spyder**[4] | | Forward & Backward | Dynamic | Interprocedural |
| **Unravel**[5] | | Backward | static | Interprocedural |
| **WALA**[6] | **JAVA** | Backward | static | Interprocedural |
| **JSlice**[7] | | Backward | Dynamic | Interprocedural |
| 1: (Ottenstein and Ottenstein, 1984) | | | | |
| 2: (Liu, 2013) | | | | |
| 3: (GrammaTech webpage, last access: 22 Mar 2022) | | | | |
| 4: (Hoffner, 1995) | | | | |
| 5: (Lyle and Wallace, 1997) | | | | |
| 6: (SourceForge webpage, last access: 17 January 2022b) | | | | |
| 7: (SourceForge webpage, last access: 17 January 2022a) | | | | |

**Table 2.3.:** Six program slicing tools which were studied by this dissertation to examine their potentials for the general goal of this dissertation. The extensive search, however, revealed only Schatz as a program slicer for Fortran source codes albeit not useful for the approach introduced in Chapter 6.

of the previous reusable component extraction works concern other programming languages than Fortran, which is the focus of this dissertation. In contrast to previous research, the approach described in Chapter 6 generates two slices from a Fortran program at both statement and namespace levels. Moreover, it was explained that the existing program analysis tools cannot fulfill all the requirements of this dissertation, which are extracting the source code of an arbitrary component from a Fortran program, extracting the global variables shared between the component and the program as well as separating the source code shared between both sides. Hence, the novel program analysis approach introduced in this dissertation is still required to benefit either from the existing tools or to build new tools to achieve the general goal of this dissertation.

# 3. The Concurrent Radiation Scheme

*This chapter describes the concurrent radiation scheme in the atmospheric model ECHAM6, which was implemented within this dissertation to increase the scalability and the computational performance of the model. The discussion starts by a short description of the role of radiative transfer in atmosphere and continues by explaining why the classical calculation of this process in ECHAM6 is a major stumbling block in reducing the overall simulation time.*

## 3.1. Radiative transfer in atmosphere

Radiative transfer is one of the most expensive parts in atmospheric simulations (Balaji et al., 2016). This process is resolved to respond to the changing state of the chemical species which interact with the radiation (Balaji et al., 2016; Salby, 1996; Wallace and Hobbs, 2006). Solar energy is the driving force for the atmosphere through radiative transfer, which is the only physical process that is capable of exchanging energy between a planet like the Earth and the rest of the universe (Bai and Zong, 2021; Wallace and Hobbs, 2006). Energy transfer in the atmosphere involves electromagnetic radiation in two separated wavelengths: *shortwave*, emitted by the sun, and *longwave*, emitted by the earth's surface and the atmosphere (Iqbal, 2012; Wallace and Hobbs, 2006; Salby, 1996). There are several atmospheric processes - including greenhouse gases, aerosols and clouds - that interact with electromagnetic radiation through the mechanisms of absorption, scattering and emission. The level of interaction strongly depends on the state of the atmospherics particles (evolving by advection, cloud processes and chemistry) and the optical properties (the wavelengths and intensity) of the incident radiation (Myers, 2017; Wallace and Hobbs, 2006).

In principle, the absorption of solar radiation by the atmosphere and the earth's surface must be balanced by the longwave emission to space from the terrestrial radiation (Parkhomenko, 2018; Salby, 1996). It is crucial for atmospheric models to accurately represent the radiative transfer process (Rasp, 2019). Solving the problem is in essence straightforward (Balaji et al., 2016; Wallace and Hobbs, 2006). However, this can be quite computationally demanding in practice, despite the simplifying approximations adopted in the radiation component (Balaji et al., 2016; Wallace and Hobbs, 2006). As a result, it is too expensive to calculate radiative transfer in every time step and grid point and, thus, in many climate models around the world (such as the ECMWF Integrated Forecasting System (IFS)), radiative transfer is calculated infrequently in time and/or on a reduced spatial grid than the rest of atmospheric physics - entirely pursuing a performance improvement rather than fulfilling any other technical objective (Balaji et al., 2016; Hogan and Bozzo, 2015; Morcrette et al., 2008).

## 3.2. Classical Radiation Scheme in ECHAM6

The calculation of radiative transfer in ECHAM6 is represented with PSrad/RRTMG (a postscript to the Rapid Radiative Transfer Model for GCMs(Pincus and Stevens, 2013) for both shortwave and longwave parts of the electromagnetic spectrum (Stevens et al., 2013). The radiation component is one of the most expensive components in ECHAM6. Figure 3.1 suggests that this component (the red curve) may take up to over 80% of the total simulation time while the calculation of other atmospheric processes (the blue curve) is below 20%. In this figure (and also throughout the dissertation), *RAD* denotes the calculation of radiative transfer and *ATM* denotes the calculation of the atmospherics processes except for radiative transfer. The relative cost of calculating radiative transfer is affected by the following factors:

- The number of MPI processes
- The temporal and spatial resolutions

### 3.2.1. Impact of MPI processes on the radiation cost

As Figure 3.1 suggests, the relative time contribution of the radiation component varies with the number of MPI processes assigned to the model. As the model keeps scaling, the cost of resolving radiative transfer is reduced. This is an indication of higher scalability of the radiation component in comparison to the whole model. Separate experiments confirm the same different scaling profiles . Figure 3.2 shows the total runtime of the model and compares it to the pure execution time of the radiation component. This red curve clearly shows that the calculation of radiative transfer scales almost perfectly while the blue curve shows the other components (the main model) tend to become flat towards the end as they fail to benefit from more available resources. The higher scalability of the radiation component can be attributed to the columnwise organization of the atmospheric physics and the embarrassingly parallel nature of the workload. In other words, since the individual columns (iterating the k index in an (i, j, k) discretization) have no cross-dependency in (i, j), it allows for fine-grained parallelism (Balaji et al., 2016). This is the reason

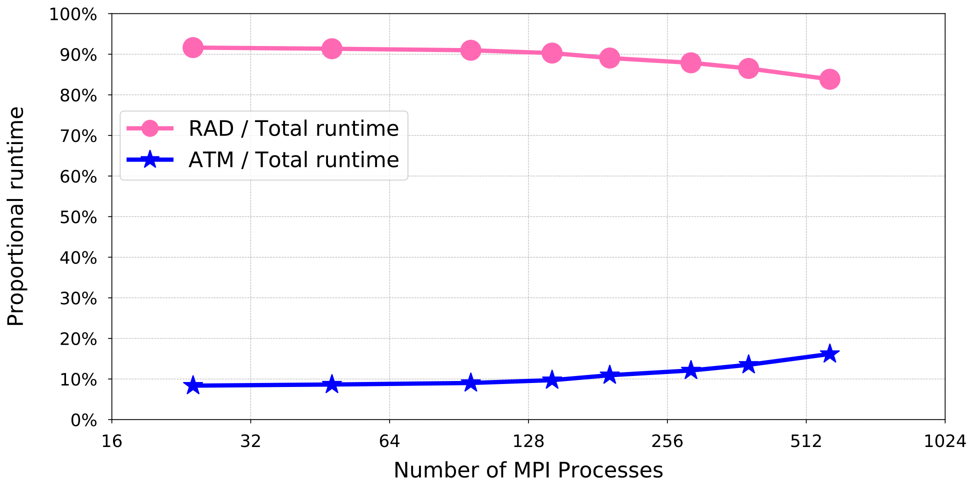


**Figure 3.1.:** The (expected) relative time contribution of the radiation calculations in ECHAM6 to the total runtime if radiative transfer is calculated in every time step. *RAD* denotes the calculation of radiative transfer and *ATM* denotes the calculation of the atmospherics processes except for radiative transfer.

why the radiation component can intrinsically scale better beyond the limitations of the main model.

Allocating the number of MPI processes for the simulations must follow a strict valid settings options controlled by the spatial resolution and the inherent configurations and the model. Experiments also indicate that the model is not entitled to running on more resources much further beyond what the curves in Figure 3.1 and Figure 3.2 indicate. As a negative result, using higher scalability of the radiation component is impeded by the lower scalability of the whole model.

### 3.2.2. Impact of temporal resolution on the radiation cost

Intuitively, radiative transfer is expected to be calculated in each time step of the model. In Figure 3.1, the radiation component and the main model were assumed to adopt the same time resolution, i.e. $\Delta t_{RAD} = \Delta t_{ATM}$. In other words, radiative transfer was calculated in every time step as the other atmospheric processes. In reality, however, this is different and ECHAM6 benefits from a larger radiation time step to improve the performance of the model. Figure 3.3 shows the organization of the classical radiation scheme in ECHAM6. In this figure, the red parts show

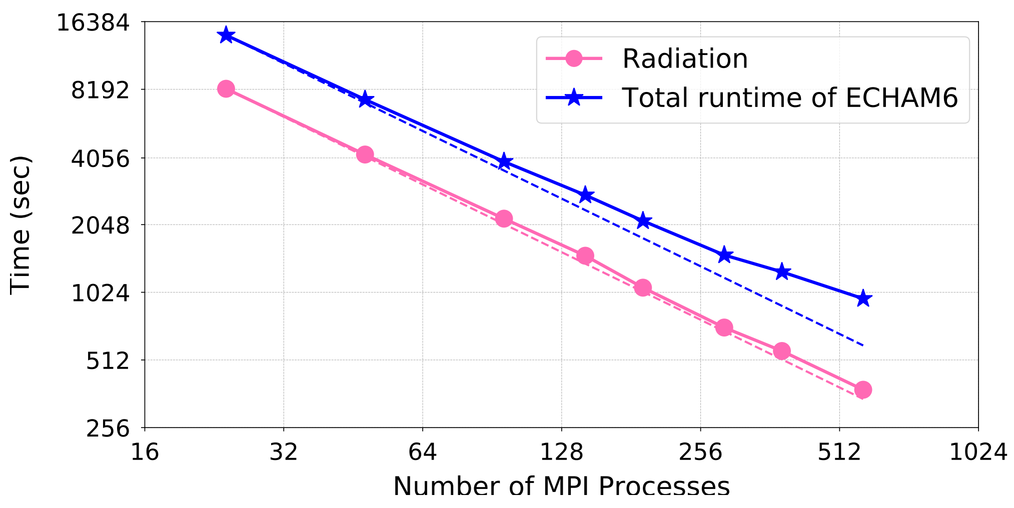


**Figure 3.2.:** The scaling curve of the radiation component vs. the whole atmospheric model ECHAM6 shows that the radiative calculations can keep scaling almost linearly while the main model fails to benefit from a higher number of resources towards the end.

the calculation of radiative transfer and the blue parts show the calculation of other atmospheric processes.

As it is apparent, the radiation component adopts a coarser time resolution than the rest of the model, i.e. radiative transfer is calculated once in every multiple atmospheric time steps, i.e. $\Delta t_{RAD} = n * \Delta t_{ATM}$. In its flagship configuration, ECHAM6 updates the optical properties of radiation every two hours ($\Delta t_{RAD} = 8 * \Delta t_{ATM}$ and $\Delta t_{ATM}$= 15min) except for very high-resolution (T255) simulations where the radiation is calculated hourly. Although the choice of larger radiation time steps ($\Delta \mathrm{t}_{\mathrm{RAD}} > \Delta \mathrm{t}_{\mathrm{ATM}}$) evidently reduces the overall runtime of simulations, radiative portion is yet considered relatively high for some configurations in ECHAM6. As shown in Figure 3.3, the radiative calculations take up from almost 40% to 58% of the total simulation time at the CR resolution, depending on the number of MPI processes assigned to the model.

### 3.2.3. Impacts of larger temporal and spatial resolutions on accuracy

Adopting larger temporal or spatial resolutions for calculating radiative transfer may, however, have negative impacts on the model accuracy (Balaji et al., 2016; Pincus and Stevens, 2013; Morcrette, 2000). At coarser radiation time steps, as

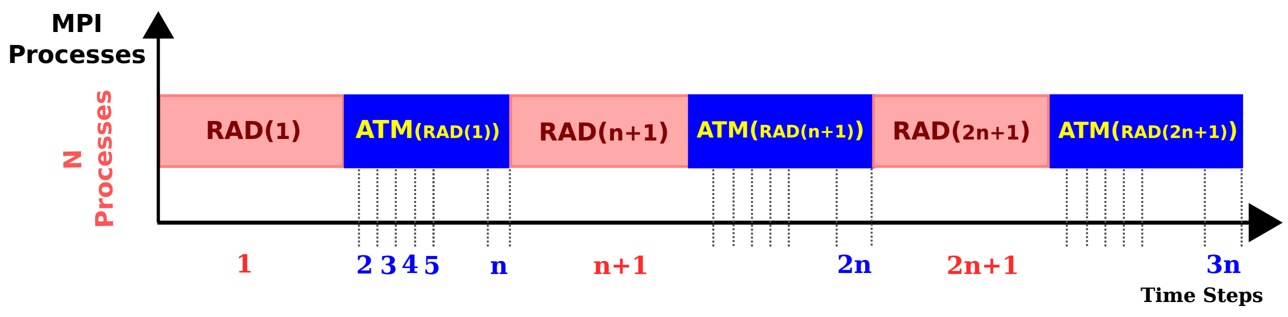


**Figure 3.3.:** The organization of the classical radiation scheme in ECHAM6: radiative transfer is resolved sequentially with respect to the other atmospheric physics and dynamics and it is stepped forward at a slower rate than the other atmospheric processes.

shown in Figure 3.3, there are multiple normal atmospheric time steps between two consecutive radiation time steps in ECHAM6. Thus, the radiation results are used beyond the state of the input tracers, which may be as much as $\Delta t_{RAD}$- $\Delta t_{ATM}$ (approximately two hours) behind. In consequence, the atmospheric calculations are provided with old feedback from the radiation component within the normal time steps. This scheme has a negative impact on the accuracy of the model. These infrequent calculations of the radiative heating may result in numerical instability in climate models. The use of the lagged state can be viewed as a potential source of discrepancy between the cloud field and the "cloud shadow field" seen by the radiation component. This, therefore, introduces numerical errors in atmospheric models and becomes considerably worse at higher resolutions (Balaji et al., 2016; Pauluis and Emanuel, 2004).

Some solutions have been proposed to mitigate the negative impacts of infrequent radiation calculations. While some techniques focus on optimizing the calculations for calling the radiation component more frequently without affecting the computational costs, some others suggest rescaling old radiation feedback wherever the new results are not available (Hogan and Hirahara, 2016; Hogan and Bozzo, 2015; Morcrette et al., 2008). To improve the model's accuracy in ECHAM6, longwave irradiance is rescaled based on the surface temperature while shortwave irradiance is rescaled by the zenith angle on non-radiation time steps in between an update of the optical properties (Stevens et al., 2013).

### 3.2.4. Sequential Component Organization: Root of problems

The entire crux of the problem in the classical radiation scheme can be attributed to the sequential organization of the components inside the model, as suggested by Algorithm 3.1. Although the actual code is far more complex, this abstract representation shows how atmospheric processes inside the model are stepped forward one at a time in a time stepping loop. "*ATM_part_1*" and "*ATM_part_2*"

**Algorithm 3.1** An abstract representation of the sequential component organization in the atmospheric model ECHAM6. As it suggests, the atmospheric processes inside the model are stepped forward one at a time in a time stepping loop and thus their calculations cost contribute directly to the overall simulation time.

```
Program ECHAM6

  call Initialization

  while (time steps < enough)

    call ATM_part_1

    if  (radiation time step) call radiation(arguments)

    call ATM_part_2

  end while

  call cleanup

End Program ECHAM6
```

in this algorithm denote the calculation of all the atmospheric processes except for radiative transfer, which is handled by "*radiation*" subroutine. In this scheme, the computation time of each component including the radiation directly contributes to the overall simulation time. In particular, the radiation component significantly delays the following calculation of other atmospheric physics and dynamics during the entire course of simulation. As it is apparent in Figure 3.3, this architecture prolongs the radiation time step in proportion to the high computational cost of the radiation component. It will be shown in the next section that this long response time of the radiation calculations is not, however, inevitable and can be avoided by reorganizing the component inside the model.

Moreover, the sequential organization of components in the classical ECHAM6 creates another obstacle that hinders the optimization of the model. In fact, ECHAM6 traditionally benefits from MPI and implements domain decomposition parallelism to expedite the computations. As explained before, the radiation calculations display a higher scalability than the main model in this framework. Such a higher scalability is instrumental in reducing the high computational cost of the radiation calculations. However, the sequential architecture of the classical ECHAM6 restrains the benefit by forcing the radiation component to use the same computational resources as the rest of the model. As a consequence, the component is hindered by the limited scalability of the whole model.

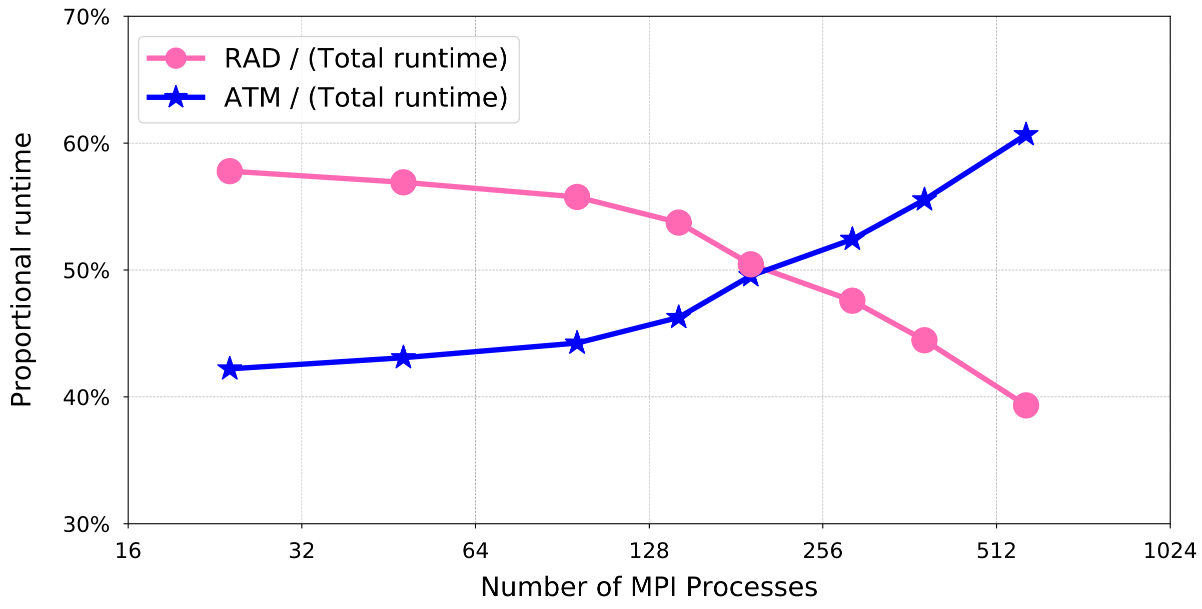


**Figure 3.4.:** The relative time contribution of the radiation calculations in ECHAM6 (using the classical radiation scheme) at the CR resolution.

## 3.3. Concurrent Radiation Scheme

It was discussed in the previous section that, even in light of larger radiation time steps, radiation calculations yet impose a daunting cost on the atmospheric simulation at coarse (CR) resolution in ECHAM6. It was also shown that the sequential treatment of resolving atmospheric processes is implicated in the long response time of the radiation component in every radiation time step and restricts the benefit of the higher scalability of the radiation calculations. This dissertation proposes to modify the sequential component organization to resolve radiative transfer concurrently with the other atmospheric calculations, as shown in Figure 3.5. This new scheme is denoted by *the concurrent radiation scheme.* In Section 2.1.1, it was explained that there are three levels of parallelism in ECHAM6. The concurrent radiation scheme implements an additional level of parallelism inside the model by applying coarse-grained component concurrency (Balaji et al., 2016) to the radiation calculations. This approach eliminates the high response latency from the radiation time step and paves the way for a higher scalable model. In contrast to the classical scheme, the concurrent radiation scheme starts resolving radiative transfer much earlier before the next radiation time step arrives. As a result, the main model receives feedback from the radiation component much faster upon the request. This technique minimizes the response latency of the component and reduces the overall simulation time. In this approach, radiative transfer is calculated concurrently with other atmospheric processes along the course of normal time steps. Due to the

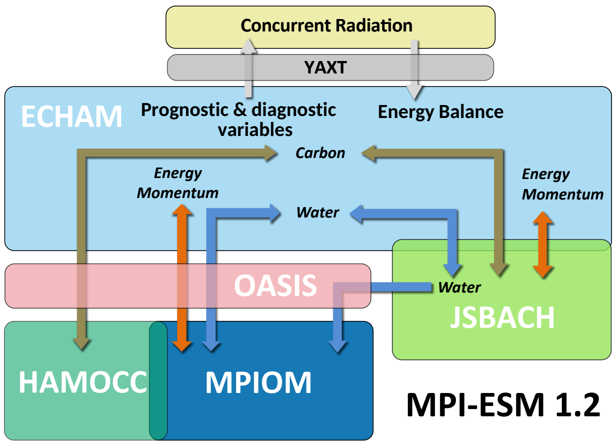


**Figure 3.5.:** The concurrent radiation scheme adds a new level of parallelism inside the atmospheric model ECHAM6 and increases coarse-grained component concurrency in the coupling architecture of the Earth system model MPI-ESM (courtesy of Deutsches Klimarechenzentrum (last access: 4 May 2022)).

dependency between components, the coupling fields are also exchanged between the radiation component and the main model within the radiation time steps but without the typical delay experienced in the classical scheme.

Figure 3.6 describes a method for casting radiative transfer as a concurrent component using distributed memory computing. In the concurrent scheme, the radiation component and the main model are organized on two separate sets of MPI processes which enable parallel calculations of radiative transfer and other atmospheric processes. In this approach, the radiation component runs within the same binary code of the model, but the allocated resources are split between the main model and the component. ECHAM6 classically benefits from MPI to implement data parallelism using domain decomposition and, thus, this techniques is applied to all the components inside the model including the radiation - due to the traditional organization of the components. In the concurrent scheme, the radiation keeps benefiting from the old data parallelism through domain decomposition but on a separate set of MPI processes. An MPI interprocess communicator enables the communication between the component and the main model. In the concurrent radiation scheme, the radiation component and the main model intuitively follow the same domain decomposition and thus they are assigned the same number of processes. However, it will be explained shortly that the radiation component can adopt coarser or finer domain decomposition and allocate a different number of MPI processes than the main model.

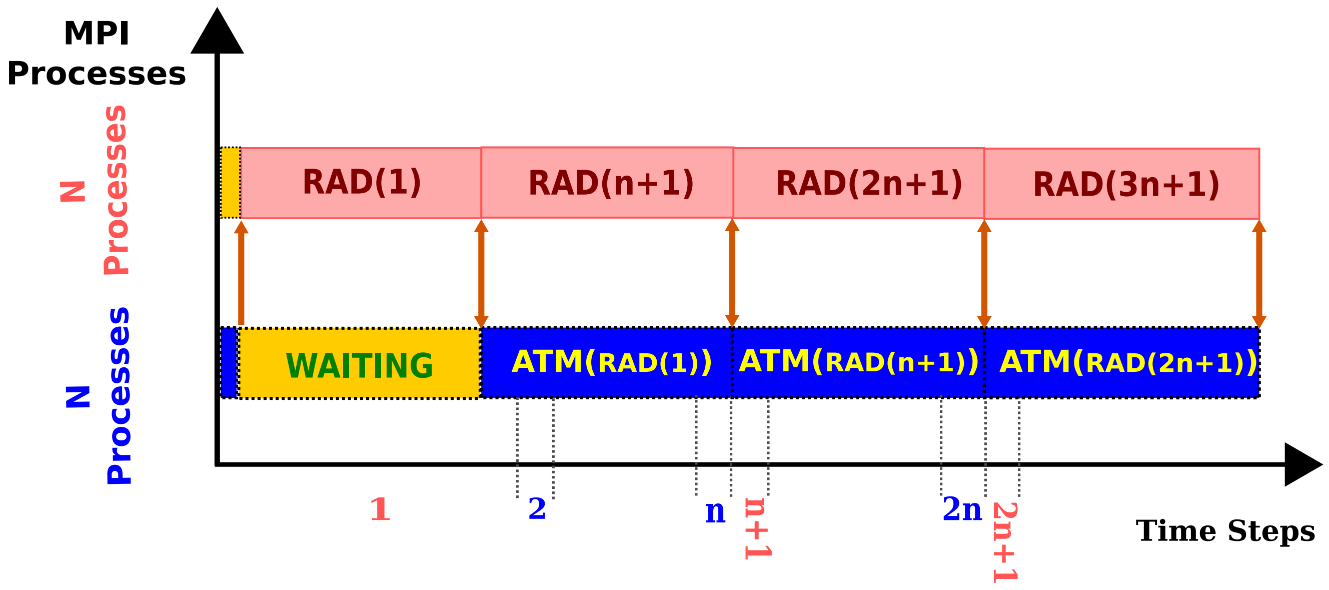


**Figure 3.6.:** The re-organization of the radiation component in parallel with the rest of atmospheric physics and dynamics in ECHAM6. In the first (radiation) time step, ATM sends the input data to RAD, but it has to wait long until it receives the results from RAD. In the following radiation time steps, however, the data exchange takes place immediately one after the other (i.e. ATM first receives the results of radiation calculations and immediately provides the input data to RAD for the next radiation calculations.). This way, ATM is supposed to experience a minimum idle time when it interacts with RAD.

## 3.4. Arbitrary Domain Decomposition

It was explained that the radiation component is far more scalable than the main model in the atmospheric model ECHAM6. This feature could potentially be exploited to reduce the high computational profile of radiative transfer in the classical radiation scheme if the component were capable of choosing the best choice of domain decomposition freely regardless of the configuration of the main model. In the classical model, however, this is not possible due to the following problems:

1. the sequential component organization:
   This scheme does no allow the component and the main model to run on different sets of resources.
2. sharing the model states and configuration:
   This problem does not allow the component and the main model adopt different (domain decomposition) setups as they use shared variables over some of the model's states and configuration.

These two problems are solved in the concurrent radiation scheme as the radiation component and the main model become concurrent processes, thus having different address spaces. As a result, they can adopt different (domain decomposition) setups and allocate different number of MPI processes. In general, there are three conceivable domain decomposition arrangements that can be adopted for the radiation component in comparison to the main model's setups. These options are as follows:

1. identical domain decomposition:
   In this scheme, the radiation component and the main model are assigned the same number of MPI processes and there is a one-to-one communication between the processes of both sides.
2. coarser domain decomposition:
   In this scheme, the radiation component allocates a lower number of MPI processes than the main model. As a result, one process of the radiation component calculates radiative transfer for multiple processes of the main model.
3. finer domain decomposition:
   In this scheme, the radiation component allocates a higher number of MPI processes than the main model. As a result, multiple processes of the radiation component calculate radiative transfer for one process of the main model.

This approach is a means solely aimed at improving the overall performance of the model and creating a load-balance between the MPI processes assigned to the main model and the component. It is noteworthy that the accuracy of the model will not however be affected if the radiation component and the main model adopt different domain decomposition. As long as the temporal and spatial resolutions of the model and the radiation are remained intact, the simulations results are expected to remain bit-wise identical.

## 3.4.1. Benefits

Arbitrary domain decomposition in the concurrent radiation scheme can potentially offer the following benefits:

- higher scalability
- load balancing
- model consistency

#### 3.4.1.1. Load balancing.

It was explained before that the concurrent radiation scheme makes it possible for the radiation component and the main model to adopt different domain decomposition from each other. This feature can be exploited in load-balancing in the model as explained below.

**Impacts of identical domain decomposition:**

In identical domain decomposition, the main model and the radiation component span the same number of MPI processes. In addition, for each MPI process assigned to the main model, there is only one dedicated peer process assigned to the radiation component for calculating radiative transfer. In another word, MPI process 0 from ATM exchanges data with MPI process 0 from RAD and so on. However, it is expected that this setup results in a load imbalance in the model. In Chapter 5, such a negative impact of this setup will be shown with some explicit experiments. Figure 3.4, however, implies the same message. It is clear from the curves in this figure that, in some configurations (when the whole model allocates 24, 48, 96 or 144 MPI processes), RAD is more expensive than ATM. Hence, it is expected that, in the concurrent radiation scheme in the respective configurations, ATM keeps waiting for RAD, resulting in a load imbalance between ATM and RAD. Figure 3.7 schematically shows that the load imbalance takes place repeatedly in the coarse of simulation before the next radiation time step arrives.

By the same token, it is clear from Figure 3.4 that if the model allocates 288, 384 or 576 MPI processes in the classical radiation scheme, RAD will be less expensive than ATM. This implies that, in the concurrent radiation scheme in the respective configurations, an idle time will be inflicted on RAD, resulting in a load imbalance between ATM and RAD. In Chapter 5, some experiments will be performed to

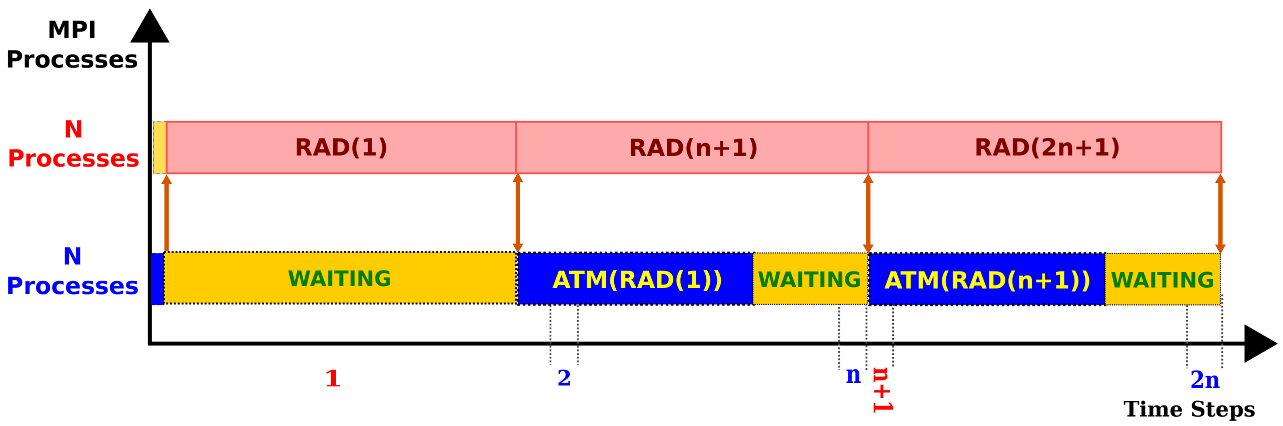


**Figure 3.7.:** Identical domain decomposition is expected to cause ATM to experience an idle time in some configurations in the concurrent radiation scheme, resulting in a load imbalance between ATM and RAD. For example, this can happen when ATM and RAD individually adopt 16, 24, 96 or 144 MPI processes. In these configurations, the whole model allocates (16+16=) 32, (24+24=) 48, (96+96=)192 or (144+144=) 288 MPI processes.

indicate the idle time explicitly. Figure 3.8 schematically shows the idle time in some setups in the concurrent radiation scheme.

### Impacts of finer domain decomposition

Finer domain decomposition at the radiation component can bring two advantages to the concurrent radiation scheme. These benefits are as follows:

- higher scalability
- load balancing

In Figure 3.7, it was shown that identical domain decomposition can potentially result in an idle time at the MPI processes assignmed to ATM. Adopting finer domain decomposition can expedite the calculation of radiative transfer, thus reducing the idle time inflicted on the MPI processes assigned to ATM. Hence, this setup can ideally remove the load-imbalance between ATM and RAD, as shown in Figure 3.9, while improving the scalability of the model further in comparison to identical domain decomposition.

### Impacts of coarser domain decomposition

In Figure 3.8, it was shown that identical domain decomposition can potentially result in an idle time at the MPI processes assignmed to RAD. Adopting coarser

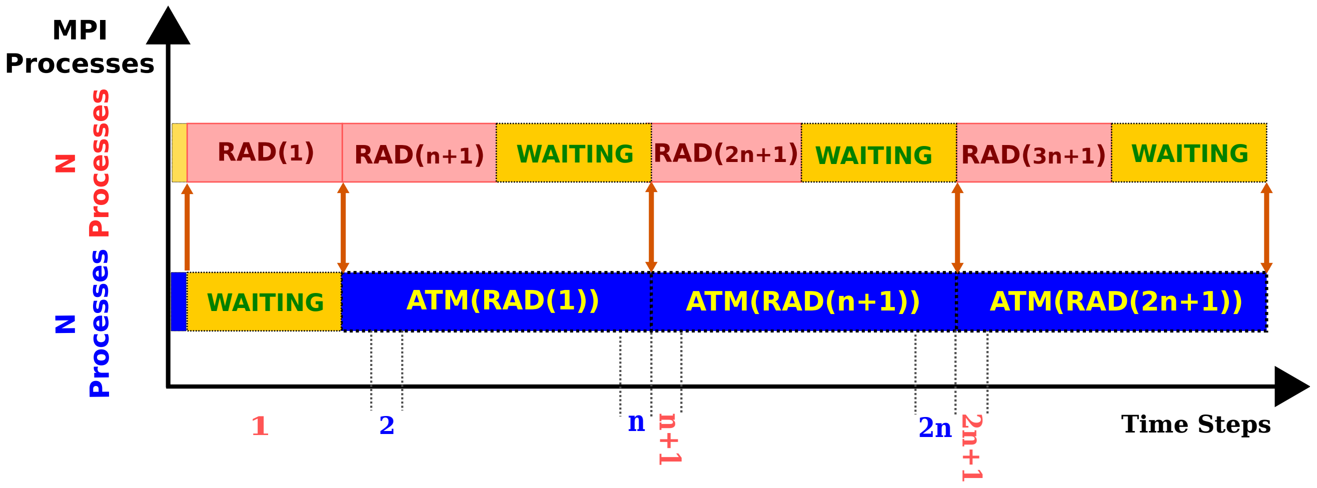


**Figure 3.8.:** Identical domain decomposition is expected to cause the radiation component experiences an idle time in some configurations in ECHAM6 using the concurrent radiation scheme, resulting in a load imbalance between ATM and RAD. For example, this can happen when both ATM and RAD individually adopt 288, 384 or 576 MPI processes. In these configurations, the whole model allocates (288+288=) 576, (384+384=) 768, (576+576=)1152 MPI processes.

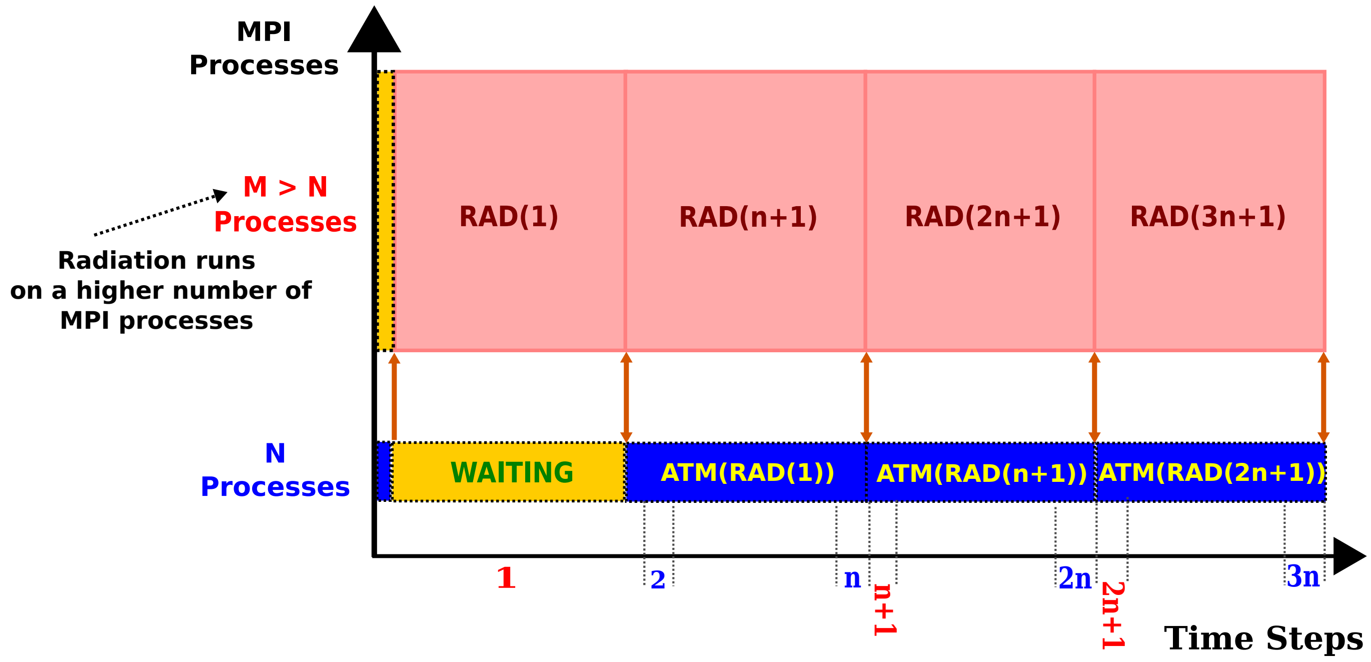


**Figure 3.9.:** Finer domain decomposition scheme can potentially remove the waiting time shown in Figure 3.7 and thus achieves a higher scalable model with an improved workload distribution between MPI processes assigned to ATM and RAD.

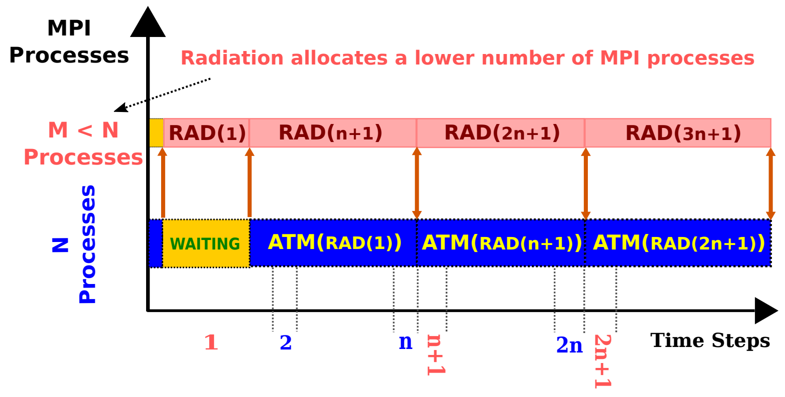


**Figure 3.10.:** Coarser domain decomposition scheme can remove the waiting time shown in Figure 3.8 and thus improves the workload distribution between MPI processes assigned to ATM and RAD.

domain decomposition can remove the idle time inflicted on the MPI processes assigned to RAD at the expense of slowing it down. Hence, this setup can ideally remove the load-imbalance between ATM and RAD, as shown in Figure 3.10.

#### 3.4.1.2. Model Consistency

In Section 3.2, it was discussed that the radiation component is stepped forward in a larger time resolution in order to reduce the negative impact of the high computational profile of radiative transfer. This however results in inaccuracy in the model. The concurrent radiation scheme may offer a way forward to decrease the discrepancy between $\Delta t_{RAD}$ and $\Delta t_{ATM}$, and bring us toward more physical consistency between the radiative and physicochemical atmospheric states (Balaji et al., 2016; Pauluis and Emanuel, 2004; Xu and Randall, 1995). Two solutions can be proposed in this regard, which are as follows:

- taking advantage of idle times
- adopting finer domain decomposition

**Taking advantage of idle times**

Figure 3.11 schematically illustrates a contrived configuration in which ATM and RAD adopt identical domain decomposition, but the radiation component is forced to remain idle almost half of the total runtime of the model. This is, however, an ample opportunity for reducing the radiation time step to a half and calculating radiative transfer twice as shown in Figure 3.12. This scheme thus creates a more accurate model without increasing the total simulation time or increasing the resource usage while removing the load-imbalance between ATM and RAD as well.

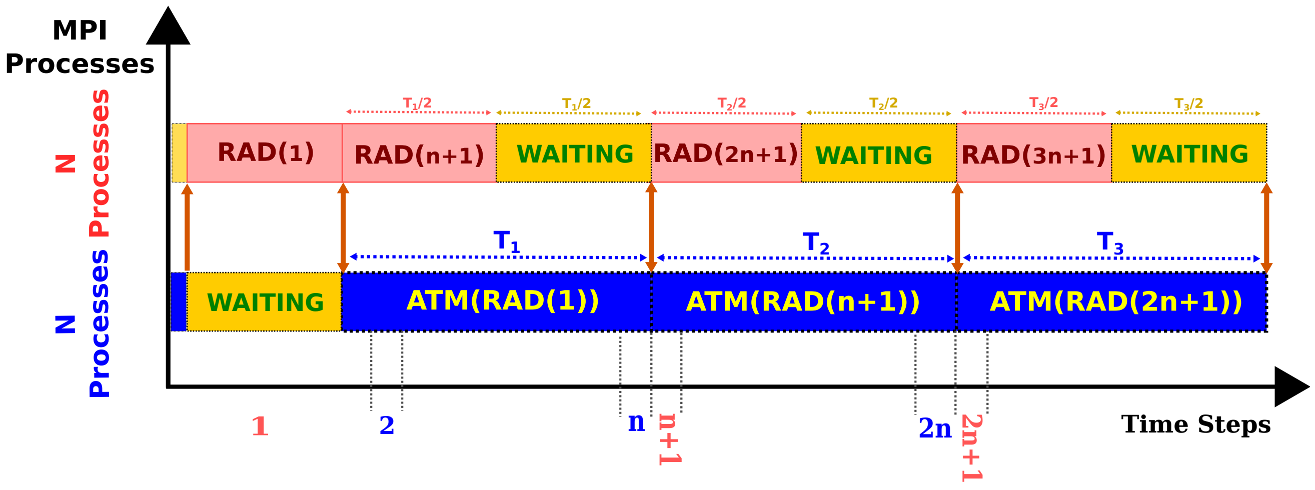


**Figure 3.11.:** A contrived model configuration in which the same domain decomposition is assigned to ATM and RAD, thus forcing the radiation component to remain idle almost half of the total runtime of the model and leading to a load-imbalance between ATM and RAD.

### Adopting finer domain decomposition

Generally speaking, however, the concurrent radiation scheme may offer a way forward towards more physical consistency in the model between the radiative and physicochemical atmospheric states. This feature can be achieved by assigning a higher number of MPI processes to the radiation component in order to calculate radiative transfer faster and reduce the gap between the radiation time step and the model (normal) time step (Balaji et al., 2016). Ideally, if the radiation component has enough resources, it should be possible to create a consistent model by calculating radiative transfer in each model (normal) time step ($\Delta t_{RAD} = \Delta t_{ATM}$). Figure 3.13 indicates a proposed configuration for the concurrent radiation in which the radiation time step is reduced to four normal time steps. Ideally, the radiation component should run at every time step without imposing any waiting time at the atmospheric calculations. This should be possible by letting the radiation component to allocate more resources and scale enough to achieve the necessary speedup to finish in one normal time step.

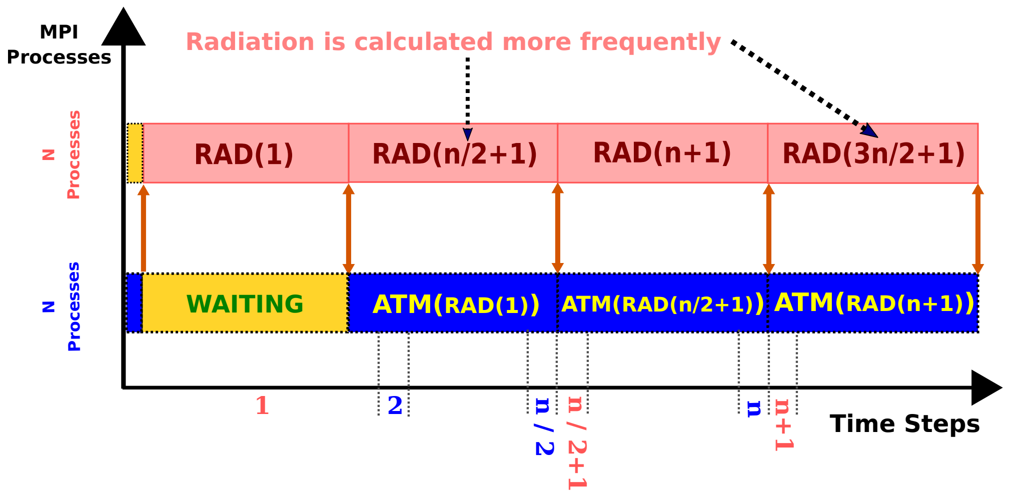


**Figure 3.12.:** A proposal for taking advantage of the idle time in Figure 3.11 to reduce the radiation time step to a half ($\Delta t_{\text{RAD}} = n/2 * \Delta t_{\text{ATM}}$), thus improving the model accuracy by creating a more consistent model. The advantage of this approach is that it does not require more resources and just takes advantage of the idle resources assigned to the radiation component.

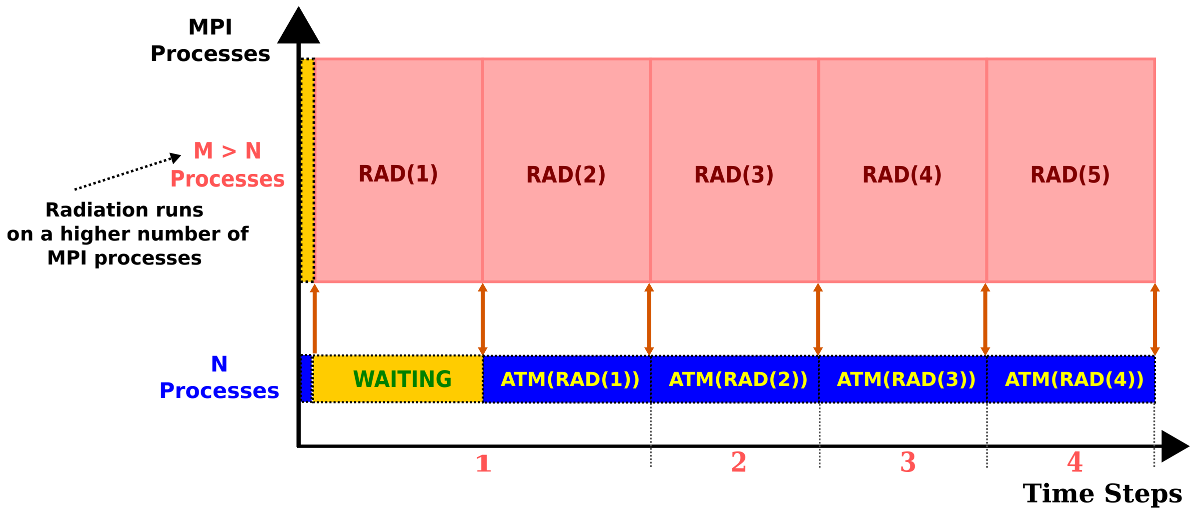


**Figure 3.13.:** This is a contrived example of a finer domain decomposition setup for the radiation component in which radiative transfer is calculated in every normal time step in order to close the gap between the radiation and the model time steps and ideally achieve $\Delta t_{\text{RAD}} = \Delta t_{\text{ATM}}$. This scheme can create consistency between the radiative and physiochemical atmospheric states in the model using the concurrent radiation scheme.

## 3.5. Chapter Summary

This chapter describes the concurrent radiation scheme (introduced) in the atmospheric model ECHAM6. It was explained that radiative transfer is one of the most expensive parts in the atmospheric simulation. In the classical radiation scheme in ECHAM6, it takes up to 58% of the total simulation time at the CR resolution to resolve this atmospheric process. It was indicated that the sequential organization of components in ECHAM6 is the major contributor to this problem as all the atmospheric processes are resolved one by one. The concurrent radiation scheme, thus, suggests running the radiation component concurrently with the main model to resolve radiative transfer in parallel with other atmospheric processes. This solution reduces the impact of the high computation profile of the component and improves the time-to-solution of the atmospheric simulation by increasing the scalability of the model. In addition, the concurrent radiation scheme allows the radiation component to adopt finer or coarse domain decomposition than the main model's setup. This feature offers a way forward to reduce the gap between the radiation time step and the model's normal time step, thus removing the inherent model inaccuracies inflicted by calculating radiative transfer at larger time steps in the classical radiation scheme. Hence, the concurrent radiation scheme can potentially bring the model towards more physical consistency between the radiative and physicochemical atmospheric states.

# 4. Implementation of the Concurrent Radiation Scheme

*This chapter presents the software engineering practice required to achieve the primary goals of this dissertation. As described in Chapter 1, this dissertation is primarily in pursuit of the following goals:*

- *Primary Goal 1: Building a new version of the atmospheric model ECHAM6 with the isolated radiation component.*
- *Primary Goal 2: Building a new version of the atmospheric model ECHAM6 with the concurrent radiation scheme.*

*This chapter describes the implementation procedures regarding each goal individually.*

## 4.1. Implementation procedure of Primary Goal 1

In Chapter 3, it was explained that the radiation calculations in the atmospheric model ECHAM6 have a high computational profile. As described in Chapter 1, two solutions (including the concurrent and single-precision radiation schemes) have been proposed to reduce the time-to-solution of the paleoclimate simulation in the PalMod project using the atmospheric model ECHAM6. However, the shared source code between the radiation component and the main model poses a major obstacle to both solutions. Sharing source code results in shared global variables between the radiation component and the main model, which may lead memory inconsistency depending on the concurrency scheme. Similarly, building a single-precision radiation scheme must prevent modifications in the radiation component from affecting the other calculations of the main model.

Hence, Primary Goal 1 aims at responding to these two concerns by building a new version of the atmospheric model ECHAM6 with using *the isolated radiation component*. The procedure to achieve Primary Goal 1 includes the following steps:

1. Extracting the radiation component from ECHAM6
2. Isolating the component in ECHAM6
3. Re-integrating the isolated component and carvedout model

### 4.1.1. Extracting the radiation component from ECHAM6

The atmospheric model ECHAM6 was developed in Fortran and is composed of several software components implemented in Fortran in almost 167,000 lines of code, including 249 Fortran modules, which are contained in 280 Fortran files. We would like to extract the following information regarding the radiation component from the model:

- Files and *namespaces* containing the code coverage of the component. Note that in this dissertation, we refer to the variable scope of a Fortran file, Fortran module or a Fortran procedure as a Fortran namespace.
- Shared files and namespaces between the component and the other parts of the model.
- Shared variables between the component and the other parts of the model.

The only information concerning the radiation component of this model (available to this dissertation) is the entry point of the component. Entry points of a component are the Fortran procedures of the component which are invoked from outside the component. The entry point of the radiation component is the Fortran subroutine " *radiation*" located in the namespace of Fortran module "*mo_radiation*". However, we have to extract the required information about the component from the original source code of the model. In Chapter 6, we will describe a novel approach for extracting a component from a Fortran program.

In this section, we present the result of applying this approach to the radiation component of the atmospheric model ECHAM6 and collect the required information. The procedure was performed mainly manually as the full automatic support tools were not available at the time of the implementation. The list of the namespaces containing the whole source code of the radiation component is shown in table Table 4.1. The first column in this table shows the namespaces that contain the source code of the component exclusively, thus denoted as *dedicated namespaces*, and they do not contain any codes from the other parts of the model. The second and third columns are shared namespace between the component and the other parts of the model. We will describe the differences between these two columns shortly. By the same token, the complete list of the shared variables between the component and the other parts of the model are shown in Table 4.2 and Table 4.3. The difference between the tables are also explained in the following sections.

| No. | Dedicated namespaces | Intact shared namespaces | Reduced shared namespaces |
|---|---|---|---|
| 1 | mo_aero_kinne.f90 | mo_kind.f90 | mo_control.f90 |
| 2 | mo_aeropt_stream.f90 | mo_interpo.f90 | mo_decomposition.f90 |
| 3 | mo_aero_volc.f90 | mo_math_constants.f90 | mo_echam_cloud_params.f90 |
| 4 | mo_aero_volc_tab.f90 | mo_namelist.f90 | mo_echam_convect_tables.f90 |
| 5 | mo_cld_sampling.f90 | mo_netcdf.f90 | mo_exception.f90 |
| 6 | mo_cloud_optics.f90 | mo_orbit.f90 | mo_filename.f90 |
| 7 | mo_lrtm_driver.f90 | mo_physical_constants.f90 | mo_gaussgrid.f90 |
| 8 | mo_lrtm_gas_optics.f90 | mo_radiation_forcing.f90 | mo_geoloc.f90 |
| 9 | mo_lrtm_kgs.f90 | mo_radiation_parameters.f90 | mo_greenhouse_gases.f90 |
| 10 | mo_lrtm_netcdf.f90 | mo_random_numbers.f90 | mo_io.f90 |
| 11 | mo_lrtm_setup.f90 | mo_read_netcdf77.f90 | mo_io_units.f90 |
| 12 | mo_lrtm_solver.f90 | mo_submodel_interface.f90 | mo_memory_cfdiag.f90 |
| 13 | mo_o3clim.f90 | mo_time_base.f90 | mo_memory_g3b.f90 |
| 14 | mo_o3_lwb.f90 | mo_time_control.f90 | mo_param_switches.f90 |
| 15 | mo_psrad_interface.f90 | mo_time_conversion.f90 | mo_submodel.f90 |
| 16 | mo_rad_fastmath.f90 | mo_util_string.f90 | mo_transpose.f90 |
| 17 | mo_radiation.f90 | mo_vphysc.f90 | |
| 18 | mo_rrtm_coeffs.f90 | | |
| 19 | mo_rrtm_params.f90 | | |
| 20 | mo_simple-plumes.f90 | | |
| 21 | mo_solar_irradiance.f90 | | |
| 22 | mo_spec_sampling.f90 | | |
| 23 | mo_srtm_driver.f90 | | |
| 24 | mo_srtm_gas_optics.f90 | | |
| 25 | mo_srtm_kgs.f90 | | |
| 26 | mo_srtm_netcdf.f90 | | |
| 27 | mo_srtm_setup.f90 | | |
| 28 | mo_srtm_solver.f90 | | |

**Table 4.1.:** The extracted namespaces from the atmospheric model ECHAM6 that contain the source code of the radiation component.

| No. | REAL | | | Derived Data Types |
|---|---|---|---|---|
| | 3D | 2D | 1D | |
| 1 | cisccp_cldemi3d | amu0_x(:,:) | ghg_cfcvmr(:) | ghg_co2mmr |
| 2 | cosp_f3d | rdayl_x(:,:) | vct(:) | ghg_ch4mmr |
| 3 | cisccp_cldtau3d | geosp(:,:) | | ghg_n2ommr |
| 4 | cosp_reffi | ghg_ch4(:,:) | | local_decomposition |
| 5 | cosp_reffl | | | vphysc |
| 6 | | | | global_decomposition |

| No. | Scalar | | | |
|---|---|---|---|---|
| | LOGICAL | INTEGER | | REAL |
| 1 | ldiag_aeropt | ico2 | nmw1 | wgt1 |
| 2 | ih2o | ich4 | nmw2 | wgt2 |
| 3 | lyr_perp | io3 | nmw1cl | wgt1_m |
| 4 | locfdiag | io2 | nmw2cl | wgt2_m |
| 5 | locosp | in2o | ndw1 | wgtdt |
| 6 | Lisccp_sim | icfc | ndw2 | wgtd2 |
| 7 | ghg_ch4 | ighg | nmw1_m | fco2 |
| 8 | lcouple | iaero | nmw2_m | cecc |
| 9 | lmidatm | lrce | cecc | cobld |
| 10 | ldebugio | nn | cobld | clonp |
| 11 | ldebugs | ngl | clonp | co2vmr |
| 12 | lrad | nlon | nmonth | ch4vmr |
| 13 | lco2 | nlev | isolrad | n2ovmr |
| 14 | lanysubmodel | nhgl | ldiur | cfcvmr |
| 15 | | ih2o | lradforcing | o2vmr |
| 16 | | nvclev | sw_gpts_ts | solc |
| 17 | | yr_perp | lw_spec_samp | flx_ratio_cur |
| 18 | | lyr_perp | sw_spec_samp | declination,initialized |
| 19 | | p_nprocs | lw_gpts_ts | earth_angular_velocity |
| 20 | | iaero_forcing | rad_perm | |

**Table 4.2.:** The shared variables creating flow data dependencies from the other parts of the model to the radiation component. These variables bring input data to the component.

| No. | 2D | Scalar | | |
|---|---|---|---|---|
| | REAL | LOGICAL | INTEGER | REAL |
| 1 | amu0_x(:,:) | lyr_perp | yr_perp | flx_ratio_cur |
| 2 | rdayl_x(:,:) | | nb_sw | declination,initialized |
| 3 | irlu(:,:,:) | | | co2mmr |
| 4 | srsu(:,:,:) | | | solc |
| 5 | irld(:,:,:) | | | |
| 6 | srsd(:,:,:) | | | |
| 7 | irlucs(:,:,:) | | | |
| 8 | srsucs(:,:,:) | | | |
| 9 | irldcs(:,:,:) | | | |

**Table 4.3.:** The shared variables creating flow data dependencies from the radiation component to the other parts the atmospheric model ECHAM6. These variables return the results of calculating radiative transfer to the main model.

### 4.1.2. Isolating the radiation component in ECHAM6

In the next step, the extracted radiation component must become *isolated* in the model. In Chapter 6, we will define the idea of *isolating a component from a Fortran program* precisely and describe a novel approach to perform this procedure clearly. *Isolating the radiation component* in the model, nevertheless, generates two slices from the original model which are called *the isolated radiation component* and *the carvedout model.* This procedure guarantees that these two slices share no source code or variables. This is because a separate copy of the shared namespaces (shown in the second and third column of Table 4.1) is created and assigned to the isolated radiation component exclusively.

It is noteworthy to emphasize that the isolated radiation component is still capable of generating bitwise identical results to the output of the original component. In addition, the carvedout program is the same as the original model, but it does not invoke the entry point of the component and does not contain the dedicated source code of the component. Since there is no code sharing, any code refactoring in the isolated radiation component does not affect the carvedout model. In addition, any change in the memory context of the component does not affect the memory view of the carvedout model and vice versa. This feature makes the component prepared for implementing both the concurrent and the single-precision radiation schemes.

Note that the shared namespaces in the second column in Table 4.1 are denoted by the *intact shared namespaces* since a copy of these namespaces are added to the isolated radiation component without any modification. However, the shared namespaces in the third column contain some codes that are not used by the radiation component. Thus, a copy of these namespaces is modified such that they contain only the relevant codes to the radiation component and then added to the isolated radiation component. Hence, these namespaces are denoted by the *reduced shared namespaces*.

The same practice must be applied to the carvedout model to collect its namespaces. However, the list of the intact and reduced shared namespaces of the model may differ from the component.

### 4.1.3. Re-integration of the isolated radiation component and carvedout model

Isolating the radiation component in ECHAM6 results in three predictable problems. In this section, we discuss these problems and some solutions for removing them. The process of implementing these solutions is referred to as *the re-integration of the isolated radiation component and the carvedout model*. After the re-integration, a new version of the atmospheric model ECHAM6 is created that is capable of regenerating bit-wise identical results to the output of the original model.

#### 4.1.3.1. Problems of the component Isolation

The first problem is the removal of the call-site to the entry point of the component from the carvedout model. This is not, however, a major issue, but the call-site must be re-activated to allow for the calculation of the radiative transfer again. The second problem concerns the memory inconsistency over the shared variables between both slices. This problem is largely due to the data dependencies between the radiation component and the main model. Among all different types, *true* data dependencies from the main model to the radiation component or vice versa result in an inconsistent memory between the isolated radiation component and the carvedout model. The third problem, however, stems from the memory allocation of some dynamic variables of the radiation component. These particular variables are shared between the component and the main model, but the allocation instructions of such variables are not added to the source code of the extracted shared namespaces during the component extraction procedure. This is because the extraction procedure is syntactic rather semantic and the missing instructions do not affect the workflow.

#### 4.1.3.2. Data flow analysis

A data flow analysis can be exploited to detect true data dependencies between the radiation component and the other parts of the model. In addition, the concerning memory allocation instructions discussed in Section 4.1.3.1 can also be explored using this technique. Performing a data flow analysis in the atmospheric model ECHAM6 is not a trivial task however, considering the large code base of the model. As shown in Algorithm 3.1, the software is composed of an initialization and a

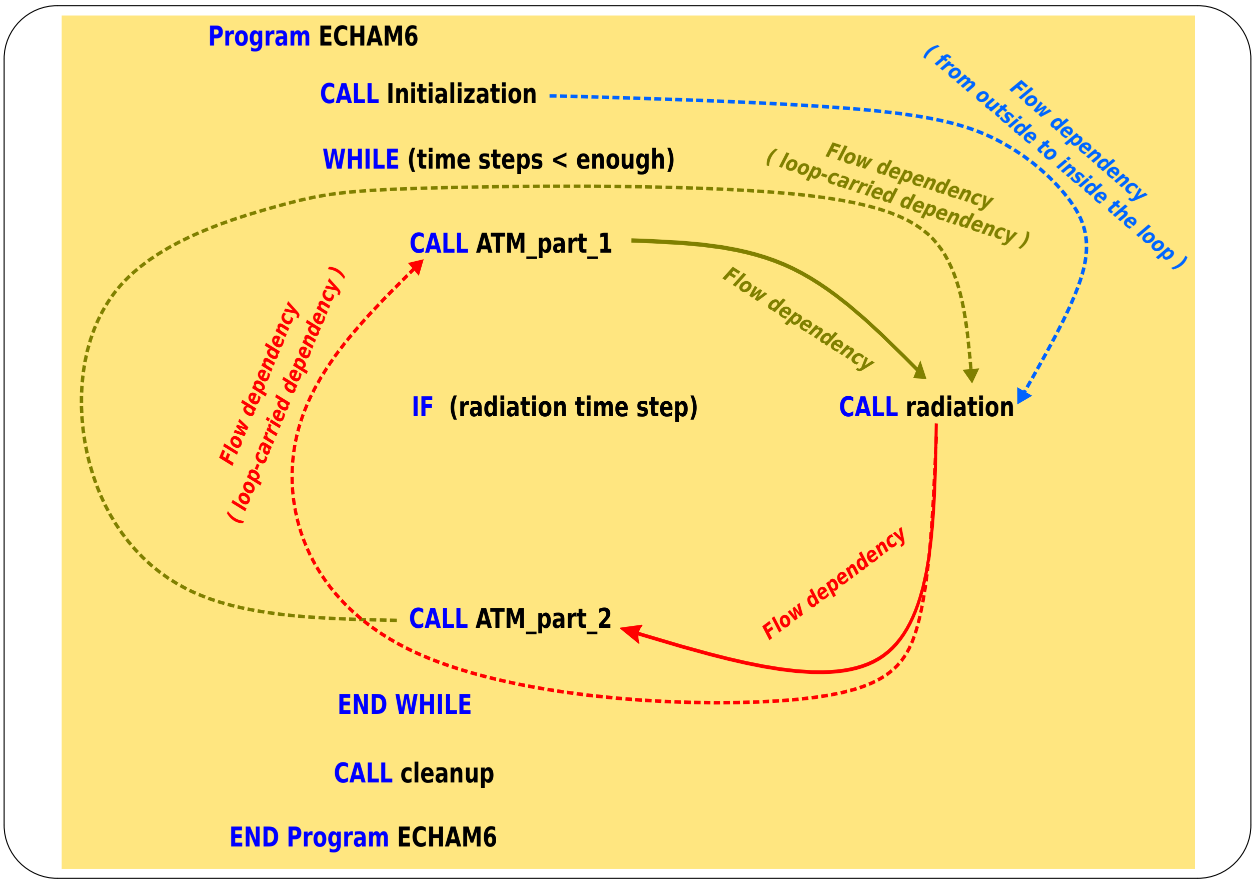


**Figure 4.1.:** A data flow analysis is carried out to extract the data dependency between the radiation component (shown by *CALL radiation*) and the other parts of the model (*ATM_part_1* and *ATM_part_2*) in the atmospheric model ECHAM6. This analysis only looks for flow dependencies as it is the only type of data dependency that affects the memory consistency between these two parts. Such a data dependency type occurs within the loop or throughout the loop-carried dependencies (as shown with the dotted lines).

time-stepping loop. Thus, we apply a data flow analysis to find the following data dependencies (as shown in Figure 4.1):

1. Flow dependencies inside "*Initialization*"
2. Flow dependencies between the "*Initialization*" and the time-stepping loop
3. Loop-independent dependencies
4. Loop-carried dependencies

In this dissertation, we found no (reliable) supporting tools for data flow analysis in Fortran source codes. Hence, the result of a manual data flow analysis in the atmospheric model ECHAM6 is shown in Table 4.2 and Table 4.3. The variables in Table 4.2 create a flow dependency from the other parts of the model to the component. Hence, they bring the input data to the component. Conversely, the variables in Table 4.3 create a flow dependency from the component to the other parts of the model, thus providing the results of calculating radiative transfer.

#### 4.1.3.3. Measures of re-integration

This section describes the measures for re-integrating the carvedout model and the isolated radiation component. An abstract overview of the re-integrated code is shown in Algorithm 4.1.

##### Adding the call-site of the component

The call-site of the entry point of the radiation component is removed from the model during the process of isolating the component. To enable the calculation of radiative transfer, the call-site must be added back to the carvedout model. In contrast to the original model (as shown in Algorithm 3.1) that makes a direct call to the entry point (*call radiation(arguments)*) and exchanges data through the formal arguments, a wrapper subroutine (*call RAD_entrypoint*) will be added to the carvedout model without any data exchange. This is because all the data exchange will take place explicitly through two new subroutines as will be explained shortly.

##### Adding missing memory allocations

After detecting the missing memory allocation instructions of the dynamic shared variables, they must be added to the extracted shared namespaces of the isolated radiation component. Hence, a subroutine for allocating each dynamic variable is added to the corresponding shared namespace of the isolated component. Such subroutines become new entry points to the isolated component and they must be called from the carvedout model.

##### Creating memory consistency

For each flow dependency between the radiation component and the other parts of the model in ECHAM6, the copies of the concerning shared variable in the isolated radiation component and the carvedout model must become consistent. To create memory consistency over a shared variable, an explicit data exchange is implemented between the isolated component and the carvedout model, which must be performed at a correct synchronization point. As shown in Algorithm 4.1, two interfaces (*export_inputdata_to_RAD* and *import_results_from_RAD*) are created to export and import data from or to the isolated component explicitly over all the shared variables collected in Table 4.2 and Table 4.3. These two interfaces will also make the implementation procedures to achieve Primary Goal 2 and External Goal easier.

#### Synchronization point

The memory allocation of dynamic shared variables and creating memory consistency between the isolated radiation component and the carvedout model must take place at correct synchronization points. During the initialization phase, we can luckily bundle all the data exchange and call-sites to the newly added entry points (to the isolated radiation component for the missing memory allocations) inside one subroutine ( *RAD_initialization*) while preserving the correct call sequence between them. The detailed information on the contents of this subroutine is skipped here, but it is available in the source code. Algorithm 4.1 shows that the correct call-site of the subroutine was detected at the end of the initialization phase of the carvedout model. The second and the third synchronizations (*call export_inputdata_to_RAD* and *call import_results_from_RAD*) take place just before and immediately after the call-site of the main entry point of the radiation component in the time-stepping loop (to minimize the data transfer overhead) as also shown in Algorithm 4.1.

## 4.2. Implementation procedure of Primary Goal 2

This section describes the implementation of the concurrent radiation scheme based on the results from the previous section. The new scheme is implemented in three steps:

1. Separating threads of execution
2. Choosing the concurrency model
3. Synchronization

### 4.2.1. Separating threads of execution

In the classical radiation scheme, ATM and RAD run sequentially. The reason is the radiation component is reachable from the main program unit of the model. As a result, the operating system cannot schedule the calculation of radiative transfer as an independent task. The concurrent radiation scheme must, therefore, make the component unreachable from the main program unit of the model to prepare the ground for creating concurrency between these two parts. To do this, all the entry points of the component must be detected and removed from the main thread of execution of the model (which starts from the main program unit). In addition, it must create another thread of execution for the component.

Luckily in Section 4.1, two separate slices of the original program (i.e. the carvedout model and the isolated radiation component) were generated, which are not reachable from each other (before the re-integration). The main program unit of the carvedout model provides the entry point to the first thread of execution.

**Algorithm 4.1** Re-integrating the carvedout model and the isolated radiation component of the atmospheric model ECHAM6. Subroutines *RAD_entrypoint*, *RAD_initialization*, *export_inputdata_to_RAD* and *import_results_from_RAD* are added to the carvedout model to build a new version of the atmospheric model ECHAM6 which is capable of generating bit-wise identical results to the output of the original model. *RAD_entrypoint* is a wrapper for the subroutine *radiation*. In contrast to the subroutine *radiation*, *RAD_entrypoint* does not exchange any data with the carvedout model and all the data exchange takes place through the import and export subroutines. *RAD_initialization* bundles all the data exchange over the shared variables between these two slices. It also contains all the call-sites for the new entry points of the isolated component for the missing memory allocation of some dynamic shared variables. The other two subroutines perform only data exchange between the two slices.

```
Program ECHAM6_CarvedoutModel

  call Initialization
  call RAD_initialization


  while (time steps < enough)

    call ATM_part_1

    if (radiation time step) then

        call export_inputdata_to_RAD
        call RAD_entrypoint
        call import_results_from_RAD

    end if

    call ATM_part_2

  end while

  call cleanup

End Program ECHAM6_CarvedoutModel
```

A second thread must, however, be created to run the isolated component. In Section 4.1, all the entry points of the component and their call sequence were also explored. This information is exploited in this section to generate a new thread of execution for the isolated radiation component. Algorithm 4.2 shows an abstract overview of separating the thread of execution of the radiation component from the main model in the atmospheric model ECHAM6. The subroutine *ATM_execution_thread* is responsible for starting the thread of execution of the main model and *RAD_execution_thread* starts the thread of execution of the radiation component. We will discuss this implementation in more detail shortly.

### 4.2.2. Choosing the concurrency model

After separating the thread of execution of the main model from the radiation component, it should be possible to create concurrency between these two parts. The concurrency can be implemented either in the shared memory (at the thread level using OpenMP threads, for example) or distributed memory (at the process level using the Message Passing Interface (MPI) framework, for example). This dissertation opts for creating the concurrency at the process level using MPI for the following reasons:

- The MPI framework can potentially provide a higher scalability to the radiation component (and thus to the whole model) as the component can freely allocate any number of (distributed) resources (regardless of the choice of the main model). In the OpenMP framework, however, RAD is bounded to run only on the resources assigned to the main model.
- In addition, the MPI framework lends itself much better for calculating radiative transfer at different domain decomposition from the main model while minimizing the overall resource usage. In fact, the feature of arbitrary domain decomposition (described in Section 3.4) requires a data reordering scheme between ATM and RAD to create a correct data communication. In the MPI implementation, we benefit from the communication library YAXT (Behrens et al., 2014) (which will be described in Section 4.2.3.3) to handle such a complexity. This option offers a major advantage over the OpenMP implementation in this dissertation.
- Furthermore, the load-balancing within the OpenMP framework cannot be handled properly since the workload per thread can only be modified within the local process data space. However, since ECHAM6 originally adopts the MPI framework to implement data parallelism, the load-imbalance problem has to be solved globally.
- Moreover, implementing (the task parallelism of) the concurrent radiation scheme in shared memory (e.g., OpenMP) would require the duplication of

**Algorithm 4.2** Creating two separate threads of execution for the carvedout model and the isolated radiation component (obtained from Section 4.1) and implementing a client-server model to manage the concurrency between these threads within consecutive radiation time steps.

**ATM Process**

```
Subroutine ATM_execution_thread

  call Initialization

  while (time steps < enough)

    call ATM_part_1

    if (radiation time step) then

        call send_command_to_RAD (RESULTS)
        call MPI_RECV (resultsdata, RAD, intercomm)

        call send_command_to_RAD (INPUT)
        call MPI_SEND (inputdata, RAD, intercomm)

    end if

    call ATM_part_2

  end while

  call send_command_to_RAD (FINISH)

  call cleanup

End Subroutine ATM_execution_thread
```

**RAD Process**

```
Subroutine RAD_execution_thread

  call RAD_Initialization

  while (TRUE)

    call receive_command_from_ATM

    select case (command)

        case (RESULTS)
            call MPI_SEND (resultsdata, ATM, intercomm)

        case (INPUT)
            call MPI_RECV (inputdata, ATM, intercomm)
            call RAD_entrypoint

        case (FINISH)
              exit

    end select

  end while

  call cleanup

End Subroutine RAD_execution_thread
```

a high amount of global data between ATM and RAD in every MPI process address space (as the model originally exploits the MPI framework for performing data parallelism in both ATM and RAD). This is because ATM and RAD would require the data of different time steps and thus one could overwrite the global data that the other would still use. Instead of keeping multiple copies of the data in the same process spaces, it would be easier to run ATM and RAD in different MPI processes so that they would have access to their own dedicated instances of the data.

- Finally, ECHAM6 already has the appropriate support for the MPI implementation as well as load-balancing in the model. This feature makes the MPI framework a more attractive solution.
- The main advantage of the OpenMP implementation that could, however, be argued is that it could avoid the MPI communication overhead between ATM and RAD. In Chapter 5, it will nevertheless be shown that this overhead is negligible in our experiments..

The other issue which is worth discussing here is that ATM and RAD could be deployed within separate Fortran programs. However, they are deployed within a new version of ECHAM6 but branch from each other upon the start. Algorithm 4.3 suggests an abstract overview of this scheme. Once an MPI process starts, it is decided whether the role of the process should be running the main model or the radiation component. The subroutine *ATM_execution_thread* is responsible for starting the thread of execution of the main model and *RAD_execution_thread* starts the thread of execution of the radiation component. In addition, an *MPI_COMM_SPLIT* groups all the processes that have similar roles and allows for their internal data communication within the domain decomposition scheme. MPI processes from different groups can also communicate within a task parallelism scheme. This is a clear example of mixing task and data parallelism using the MPI framework in the atmospheric model ECHAM6.

### 4.2.3. Synchronization

As explained in Section 4.1, the isolated radiation component and the carvedout model have an inconsistent view of the memory (before the re-integration). Running these two slices on different MPI processes also contributes to the inconsistency as the processes run in different address spaces. During the re-integration process in Section 4.1, two import and export subroutines were developed in order to create memory consistency between these two slices in pursuit of generating bit-identical results to the original model. It is, however, well-known that it is not possible to create such a persistent consistent memory in the concurrent radiation scheme. Nevertheless, due to the close time-dependency of radiation on evolving model's fields, the concurrency between the carvedout model and the isolated radiation component

**Algorithm 4.3** An abstract model for creating concurrency (in the concurrent radiation scheme) between the thread of execution of the main model and the thread of execution of the radiation component. The subroutine *RAD_execution_thread* is responsible for starting the thread of execution of the radiation component and the subroutine *ATM_execution_thread* is responsible for starting the thread of execution of the main model. In this scheme, all the MPI processes that have a rank less than the number of MPI processes required by the main model are dedicated to the main model (thus make an invocation to *ATM_execution_thread*) and the rest will be assigned to the radiation component (thus make an invocation to *RAD_execution_thread*). An *MPI_COMM_SPLIT* generates private communicators to group similar processes together.

```
Program Concurrent_ECHAM6

  call MPI_INIT
  call MPI_COMM_RANK(comm, rank)
  if (rank < model_number_of_processes) then
          has_radiation_role = FALSE
          color = 0
  else
          has_radiation_role = TRUE
          color = 1
  endif

  key = rank
  call MPI_COMM_SPLIT(comm, color, key, newcomm))

  if (has_radiation_role == TRUE) then
     call RAD_execution_thread
  else
     call ATM_execution_thread

End Program Concurrent_ECHAM6
```

is only enacted between consecutive radiation time steps and the memory of both slices must become consistent at synchronization points.

#### 4.2.3.1. Synchronization points

The isolated radiation component and the carvedout model must synchronize at each radiation time step and a consistent memory view must be created for both parts before they start new calculations (as shown in Figure 3.6). On this account, they perform the following interactions at the beginning of each radiation time step:

- sending radiation results:
  The isolated radiation component provides the results of the latest radiation calculation to the carvedout model. For example, the output from *RAD(2n+1)* (which was started at time step *(n+1)*) is exchanged at the time step *(2n+1)*.
- receiving input data:
  The isolated radiation component receives the input data from the carvedout model for the following calculation of radiative transfer. For example, the input data for calculating *RAD(2n+1)* is received at time step *(n+1)*.

#### 4.2.3.2. A client-server model

Algorithm 4.2 shows an abstract overview of a client-server model for implementing the synchronization between the threads of execution discussed in Section 4.2.1. It suggests an asymmetric communication in which one master process controls one worker process in the time stepping loop by sending three commands (requesting the results from the latest calculation of radiative transfer, providing input data for the following calculation of radiative transfer or terminating the simulation process). In an identical domain decomposition setup, one MPI process of the carvedout model controls a peer process assigned to the isolated radiation component and they exchange data. If the model is setup to allocate 1152 MPI processes, for example, there will exist 576 pairs of MPI processes (from both sides) interacting with each other. If the carvedout model adopts coarser domain decomposition, one MPI process of the carvedout model controls several MPI processes of the isolated radiation component and exchanges data with them. By the same token, if the isolated radiation component adopts coarser domain decomposition, one MPI process of the isolated radiation component is controlled by several MPI processes of the carvedout model and exchanges data with them.

#### 4.2.3.3. Creating memory consistency

For each flow dependency between the radiation component and the main model in ECHAM6, the copies of the concerning shared variable in the isolated radiation

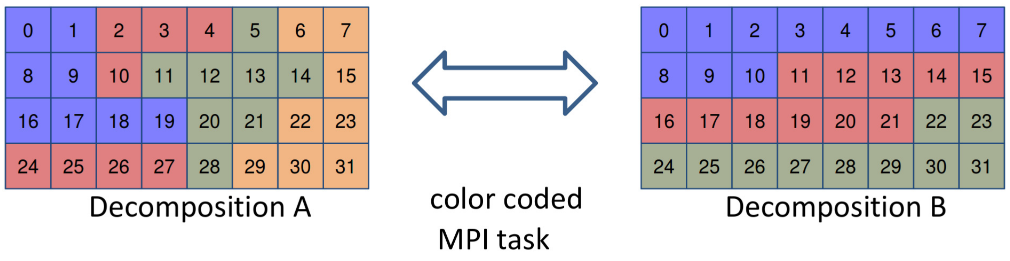


**Figure 4.2.:** YAXT library facilitates MPI communication between concurrent components with different domain decomposition layouts. (courtesy of Deutsches Klimarechenzentrum (last access: 4 May 2022))

component and the carvedout model must become consistent. The list of shared variables and the types of data dependencies that they create were already extracted in Section 4.1. To create memory consistency over a shared variable, an explicit data exchange is implemented between the isolated component and the carvedout model. Since these two slices run on different MPI processes, the data exchange will be implemented in distributed memory.

## Communication layout

Since the isolated radiation component and the carvedout model implement domain decomposition using the MPI framework, the data exchange over shared variables will take place between the peer MPI processes of both sides. If these two slices adopt identical domain decomposition, each MPI process from one side must synchronize only with its peer process from the other side. Otherwise, the number of processes contributing to the synchronization from both sides may differ and a more complex arrangement will be required. This feature, however, requires a reordering of data between the MPI processes assigned to the main model and the radiation component. To simplify the implementation, the concurrent radiation scheme benefits from the communication library YAXT (Behrens et al., 2014). This library and the required interface in the concurrent radiation scheme to this library were developed at Deutsches Klimarechenzentrum (DKRZ). The library provides an efficient and simplified platform for the formulation of communication problems and exchanging data. It is built based on the MPI framework and takes high level descriptions of arbitrary domain decomposition and derives an efficient collective data exchange. The automatic generation of MPI data types in YAXT provides direct access to the model's data for generating MPI messages and removes the need for additional copies of data or packing and unpacking. This feature makes the library suitable for exchanging bulk data (Heidari et al., 2021). Figure 4.2 shows the use of the YAXT library for coupling two concurrent components with different domain decomposition.

### 4.2.4. The full implementation

The full implementation of the concurrent radiation scheme is, however, more complex and requires more adaptation of the original code to the new scheme. For example, ECHAM6 benefits from a restart file mechanism to break long simulations into multiple consecutive experiments. These files contain the last state of the MPI processes from which the next experiment must continue. In the concurrent radiation scheme, this mechanism had to be improved to support the additional processes assigned to the radiation component. Moreover, ECHAM6 interacts with other components of MPI-ESM (described in Chapter 1). In the concurrent radiation scheme, a further modification is required to hide the MPI processes assigned to the radiation component from the coupler. Nevertheless, the detailed report on the implementation of the new restart file mechanism and interaction with the coupled model is skipped in this dissertation for the sake of brevity. The complete source code of the concurrent radiation scheme in the atmospheric model ECHAM6 is available under a permanent repository at (WDC-Climate, 2022).

## 4.3. The External Goal

The external goal of this dissertation is the implementation of the single-precision radiation scheme in the atmospheric model ECHAM6 based on the results from Primary Goal 1. The private namespaces of the isolated radiation component allows for applying single-precision arithmetic to the radiation calculations without affecting the other calculations of the model. In addition, the explicit import/export interfaces between the component and the main model allow for typecasting the input/output data to/from the component. The authors in (Cotronei and Slawig, 2020) report the full implementation of the single-precision radiation scheme and show that the calculations can be accelerated by about 40%.

## 4.4. Chapter Summary

This chapter describes the implementation procedures required to achieve Primary Goal 1 and 2 of the dissertation. Primary Goal 1 aims at building a new version of the atmospheric model ECHAM6 with the isolated radiation component. This goal is achieved by extracting the radiation component from the model, generating two slices from the original model called the isolated radiation component and the carvedout model, and finally re-integrating these two slices for building a new version of the model. In this new version, the source code of ATM and RAD are completely separated and they share no variables. In addition, an explicit mechanism is implemented to create a correct synchronization between these two parts.

Furthermore, the new version is capable of generating bitwise identical results to the output of the original ECHAM6 and paves the way for Primary Goal 2 and the external goal of the dissertation. The procedure to achieve Primary Goal 1 benefits from the novel program analysis approach introduced in Chapter 6.

Primary Goal 2, however, aims at a new version of ECHAM6 with the concurrent radiation scheme. The procedure to achieve this goal requires the following steps:

- extracting entry points of RAD and their call sequences
- extracting the shared variables between ATM and RAD
- finding the correct synchronization points between ATM and RAD
- implementing synchronization mechanisms between ATM and RAD

All these requirements have already been fulfilled in the new version of ECHAM6 (provided by Primary Goal 1). Thus, Primary Goal 2 can take advantage of this version and easily convert it to another version that uses a concurrent radiation scheme. Furthermore, the concurrency between ATM and RAD is implemented using the MPI framework as it offers a number of advantages (such as providing for a better scalability and load-balancing in the model) over OpenMP.

Finally, the external goal of this dissertation aims at a new version of ECHAM6 with the single-precision arithmetic radiation scheme. To achieve this goal, it is also a prerequisite to separate the source code of ATM and RAD from each other. However, Primary Goal 1 has already taken care of this step as well. The full implementation procedure to achieve the external goal is, nevertheless, beyond the scope of this dissertation, but it has already been pursued by an external project. A report shows that the time-to-solution of simulations with this new version of ECHAM6 that benefits from the single-precision arithmetic radiation scheme can be reduced by 40%.

# 5. Performance Results

*This chapter presents the performance evaluation of the concurrent radiation scheme (implemented in the atmospheric model ECHAM6) in comparison with the classical approach. Before delving into details, we first describe our evaluation methodology and the experiment setups to obtain the performance results required for the discussions in this chapter.*

## 5.1. Methodology

The aim of the concurrent radiation scheme is to minimize the time-to-solutions of the paleoclimate simulations (at the CR resolution) using the atmospheric model ECHAM6. It is intuitive to conclude that a performance analysis should evaluate how successfully the concurrent radiation scheme improves the model to achieve this goal. The evaluation therefore has to investigate the following issues:

1. searching for the best time-to-solution of the model (using old and new schemes) regarding the target simulations
2. comparing the performance of both schemes

Since the performance of the model varies as it scales, a variety of setups (imposed by some strict disciplines (Roeckner et al., 2003)) will be examined in search for the shortest time-to-solutions of the model (in both schemes). In the concurrent radiation scheme, however, a wide range of configurations has to be examined as ATM and RAD can take up non-identical choices of domain decomposition. Nonetheless, it should be noted that the same set of rules also applies to ATM in the concurrent radiation scheme as in the classical scheme, thus placing a constraint on the permitted configurations. The methodology here will thus take advantage of this limitation to reduce the search space in the following manner. In the first step, the model (using the new scheme) is configured to apply identical domain decomposition to ATM and RAD and hence the performance is measured at the prescribed setups. In the next step, a further investigation will be carried out to see if there is any room for improvement at finer or coarser domain decomposition on RAD's side. In the nutshell, the procedure is as follows:

- Step 1: Performance measurement
- Step 2: Performance tuning

Before delving into the discussion, it is noteworthy to emphasize that our methodology considers the performance evaluation of the model at the configurations that are the settings of the PalMod experiments. In Section 7.3, it will be shown that a comprehensive scientific validation already approves of the scientific results of the palaeoclimate simulations (performed by the concurrent radiation scheme) at such settings. In this chapter, the model is setup at the same resolutions used by such experiments but at a variety of domain decomposition to find the best time-to-solution of the model. Thus, the model is expected to be capable of delivering the same accuracy (no matter which domain decomposition is adopted) as the temporal and spatial resolutions of the model remain intact across all the experiments.

## 5.2. Experiments Setups

For the purpose of this study, a new version of ECHAM6 (based on *ECHAM-6.3.05p2*) is deployed with both classical and concurrent radiation schemes which can be configured to calculate the radiative transfer with or without separate MPI processes. Before performing an experiment, the model is set up using a configuration file. Any simulation requires an extensive list of input data to adopt the target settings, but a few relevant ones are as follows:

- Type of radiation scheme (the classical or new scheme)
- Temporal and spatial resolutions
- Allocated resources for ATM and RAD
- Simulation period (start data, stop date)

It is noteworthy that the frequency at which the radiative transfer is calculated (in all the simulations performed for this study) is set to its default value (which is once in every two hours, i.e. $\Delta t_{\mathrm{rad}} = 8 * \Delta t_{\mathrm{atm}}$ with the normal time step of 15 minutes) for all the experiments. Additionally, once the model is configured to use the concurrent radiation scheme, an equal number of MPI processes, and thus identical domain decomposition, is assigned to the main model (ATM) or the radiation component (RAD) .

The performance results presented in this dissertation are obtained from a suit of *Atmospheric Model Intercomparison Project (AMIP)* experiments on the *CR* resolution from the year 1976 to 1981. Most of the presented graphs require specialized measurements and thus numerous experiments were designed to collect the target results. To increase the accuracy, any data point presented in the graphs is the average of a minimum ten samples acquired from repeating the required simulations.

Experiments are performed on the *Mistral* supercomputer at *Deutsches Klimarechenzentrum (DKRZ)* on a machine configuration with Intel Haswell processors (*E5-2680v3 12C 2.5GHz*) and Melanox FDR Infiniband high speed interconnect. All runs allocate a layout of one MPI process per CPU core on the computing nodes equipped with two processors which are exclusively dedicated to the experiments.

Lastly, the output data for generating the plots presented in this dissertation is available under a permanent data archive at (Zenodo Repository, last access: 17 January 2022) and can be inspected without any limitations. Thanks to the arrangements by DKRZ, the source code of the model which was prepared for the experiments and used for generating the plots presented in this dissertation is also available under a permanent repository at (WDC-Climate, 2022) and can be used (after acquiring the required permission) for a further analysis.

## 5.3. The Profiling Approach

The technique used in this chapter for profiling the performance of the atmospheric model ECHAM6 is based on an internal profiling mechanism provided in a number of Fortran modules of the model. Although the performance measurement infrastructure of Score-P (Andreas Knuepfer and Others, 2022) is an interesting tool suite for event tracing and profiling of HPC applications, our experiments revealed that this tool was not capable of profiling the concurrent radiation scheme as the instrumented model would stop working. An official investigation also showed that Score-P was not able to handle MPI intercommunicators (MPICH webpage, last access: 17 January 2022) (which are also used in the concurrent radiation scheme) at the time of this study. As a result, all the code instrumentation and time measurements were performed using the internal library of the model. This effort turned out to be non-trivial as it required intricate code instrumentation and a considerable data analysis. This is because is a large volume of data were generated due to the following reasons:

- The profiling had to be performed by both (the classical and the concurrent) radiation schemes for comparing the performance of both schemes.
- All the MPI processes assigned to ATM and RAD would contribute to the profiling procedure and generating performance data. A post-processing analysis was thus required over a large set of measurements.
- Multiple points in ATM and RAD had to be instrumented, which would contribute to generating the profiling data in every MPI process.

In addition, it was revealed during the experiments that the profiling mechanism would affect the profiling process by placing an extra overhead on the MPI processes (assigned to ATM and RAD). For example, Section 5.5.3 will show how measuring the idle times of MPI processes are affected by the profling mechanisms. Hence, extra efforts were required to remove the inflicted overhead.

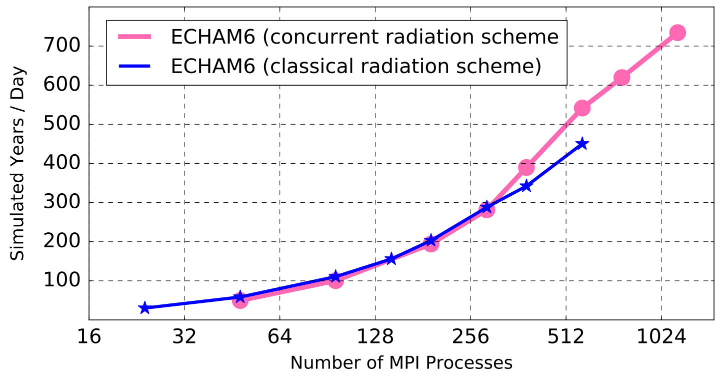


**Figure 5.1.:** The scaling curves of ECHAM6 using the classical and concurrent radiation schemes. In the concurrent radiation scheme, identical domain decomposition setups allow the radiation component and the main model to allocate the same number of MPI processes in all the experiments.

## 5.4. Step 1: Performance measurement

In Step 1, the time-to-solution of the model is measured for both classical and concurrent radiation scheme at the setups described in Section 5.1. Figure 5.1 presents two scaling curves which reflect the performance of the model with the classical radiation scheme (the blue curve) and the concurrent radiation scheme (the red curve). The horizontal axis shows the total number of MPI processes allocated by the model. It is worth emphasizing that ECHAM6 (using the classical radiation scheme) uses the same MPI processes to calculate ATM and RAD. However, when the model is configured to use the concurrent radiation scheme, half of the allocated MPI processes are exclusively dedicated to ATM and the other half to RAD. The vertical axis, on the other hand, reflects the throughput of the model in terms of the number of *simulated years per day (SYPD)*.

As it can be inferred from Figure 5.1, ECHAM6 can achieve only a maximum performance of 450 SYPD at 576 MPI processes using the classical radiation scheme at the CR resolution. However, it yields a significant improvement using the concurrent radiation scheme and reaches a maximum performance of 734 SYPD at 1152 MPI processes. It is noteworthy that, due to the limited number of grid points at the CR resolution, running the classical model at higher domain decomposition is not justified theoretically. Needless to say, it does not attain any significant performance improvement in practice either, as asserted in Figure 3.2 where the scaling curve of ECHAM6 tends to flatten towards the end. This should explain why the blue curve in Figure 5.1 stops at 576 MPI processes, as opposed to the red curve scaling beyond. On this account, the concurrent radiation scheme acquires a new significance as it becomes conducive to a higher scalable model.

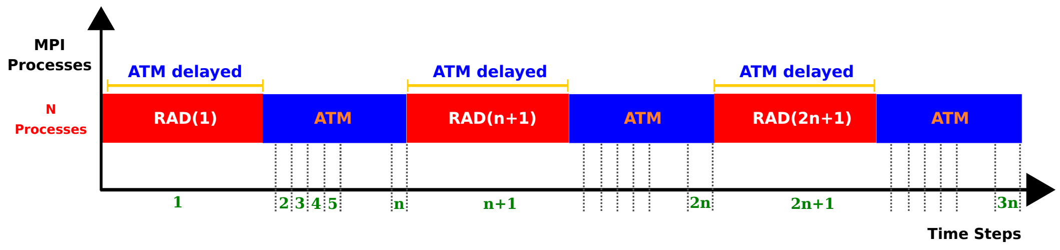


**Figure 5.2.:** The classical organization of the radiation scheme in ECHAM6: the radiation transfer is resolved sequentially with respect to the other atmospheric processes. Hence, ATM is delayed in every radiation time step.

## 5.5. Step 2: Performance tuning

In the previous section, the performance of the model with the concurrent radiation scheme was measured and it was compared with the classical scheme. In this section, we will investigate the possibility of improving the performance of the model further (at the same setups used in Section 5.4) by tuning the concurrency scheme. At this stage, some performance metrics are require to spot any potential performance bottleneck and opportunities for improving the model. As it will be shown, the load-imbalance and the communication between ATM and RAD are two major contributors to degrading the performance of model and they will be studied in some more experiments.

In the classical ECHAM6, as shown in Figure 5.2, ATM and RAD are performed sequentially and, as a result, ATM is delayed by RAD in every radiation time step. This delay contributes directly to increasing the time-to-solution of the model. The length of the delay is equal to the time required for calculating the radiative transfer. Since ECHAM6 takes advantage of data parallelism, multiple MPI processes calculate ATM in parallel. As a result, different MPI processes of ATM might experience different delays, largely due to the load imbalance imposed by the choice of domain decomposition. However, as shown in Figure 5.3, the maximum delay generated by the slowest MPI process will eventually prolong the total runtime of the model.

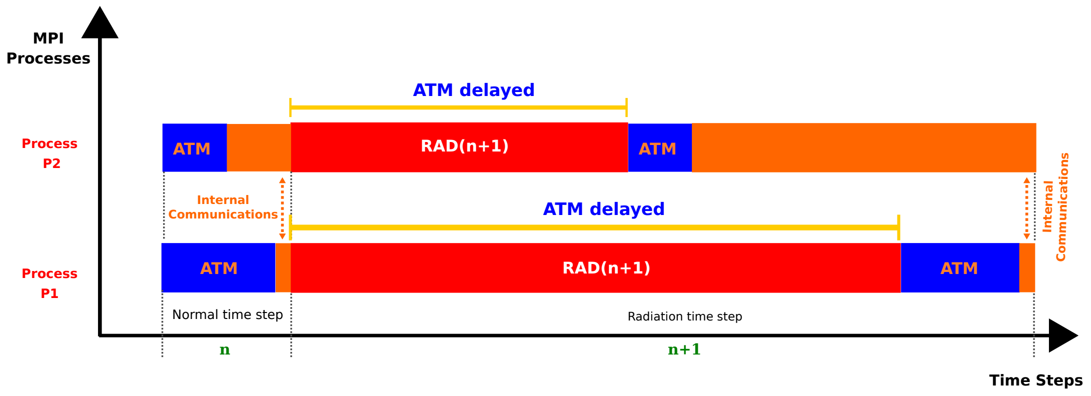


**Figure 5.3.:** Different MPI processes of ATM experience different delays from RAD. The one that experiences the longest delay stalls the other processes.

The concurrent radiation scheme aims at reducing this delay and optimally removing it completely. So the success of the new scheme strongly depends on how effectively it manages to reduce the delay imposed on ATM (by RAD). In this scheme, ATM and RAD are performed on separate sets of MPI processes. A load-imbalance between ATM processes (i.e. the MPI processes responsible for doing ATM in the new version of ECHAM6 configured to use the concurrent radiation scheme) and RAD processes (MPI processes responsible for doing RAD in the new version of ECHAM6 configured to use the concurrent radiation scheme) leads to an idle time imposed on the faster processes. When a radiation time step arrives, (heavy) RAD calculations may still be running, as shown in Figure 5.4, or may have already been finished, as shown in Figure 5.5. In Figure 5.4, ATM is faster than RAD and thus it is delayed partly by the calculations of RAD and partly by the communication between both processes. In contrast, RAD is faster in Figure 5.5, thus ATM is delayed only by the communication. In the next section, we measure the idle time and the communication time experienced by ATM and RAD to see how they affect the performance of the concurrent radiation scheme.

### 5.5.1. Measuring communication overhead

The communication between ATM processes and RAD processes is the only player that always contributes to the delay imposed on ATM in the concurrent radiation scheme. There are two purposes for the data communication:

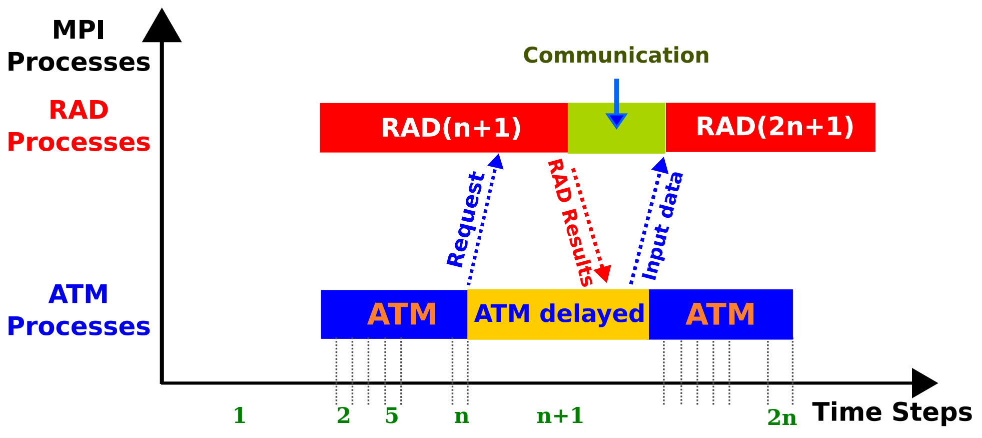


**Figure 5.4.:** The calculations of RAD are heavier than ATM, and, thus, RAD processes are slower than ATM processes. As a result, ATM is delayed by both RAD and the communication between both parties.

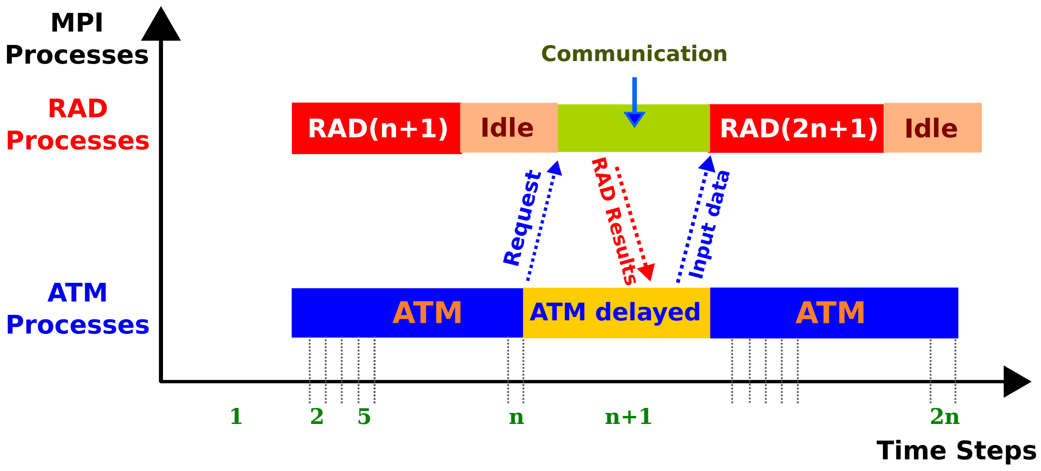


**Figure 5.5.:** The calculations of ATM are heavier than RAD, and, thus, ATM processes are slower than RAD processes. As a result, ATM does not experience any idle time from RAD though yet delayed by the communication between two parties.

- sending input data from ATM to RAD
- sending output data from RAD to ATM

In every radiation time step, the results of RAD are sent to ATM processes and they, in turn, provide new input data for the following radiation calculations to RAD processes immediately. Such interactions and the aggregated volume of exchanged data are reflected in Figure 5.6. The schematics in both Figure 5.4 and Figure 5.5 also show how these interactions make ATM go to a wait state, prolonging the overall simulation time. However, such a data exchange is inevitable due to the close data dependency between ATM and RAD. This section provides an approach for measuring these communication overheads and presents the experiments results. The model will be configured for the same setups used in Section 5.4.

### Code instrumentation

Algorithm 5.1 presents an abstract overview of the code instrumentation in ECHAM6 for measuring the communication overhead between ATM and RAD. The code instrumentation is required at both sides. Algorithm 5.1 measures the communication overhead on ATM and RAD individually as it can be different at each side. This is because of different nature of MPI_SEND (which can be either a blocking or non-blocking operation (ArgonneLab webpage, last access: 17 January 2022)) and MPI_RECV (which is always blocking). This means if ATM, for example, starts sending data to RAD, MPI_SEND might not wait for MPI_RECV at RAD's side to receive the data completely. MPI_SEND can simply leave the data in the buffer and immediately quit. As a result, ATM experiences a shorter communication time and continues. However, RAD may experience a longer communication time because MPI_RECV may have to wait long until the data is transferred (over the network perhaps) and stored in the memory. The same story applies to data exchange from RAD to ATM. This time ATM may experience longer communication than RAD. However, as Figure 5.6 shows, RAD receives a larger volume of data in comparison to ATM and this imposes a bigger communication overhead on RAD than ATM.

There is nevertheless a more essential question: why should we measure the communication cost at RAD's side in the first place? This is a valid question as we should only look for the communication overhead that contributes to the delay experienced by ATM. It may seem that the communication overhead is only what we measure at ATM's side, but this is not necessarily a complete picture. If RAD experiences a bigger communication overhead, it may slow down RAD, thus making ATM finish faster and wait for the following course of data exchange with RAD. As a result, the idle time of ATM in this case does not stem (purely) from the expensive calculations of RAD, but (rather) also from the data communication between them. This becomes especially more interesting if we notice that the heavy calculations of RAD can be handled by higher scalability of the component, while the cost of data communication conversely increases by further scaling (largely due to smaller messages as will be shown shortly). For this reason, the communication overhead on both ATM and RAD will be presented in this section.

The measurement procedure takes place in two occasions: once when ATM sends the input data to RAD and once when ATM receives the results from RAD. The communication overhead (for each side) is the aggregated time required for exchanging the input and output data. In Algorithm 5.1, an MPI barrier makes every ATM process wait for its peer RAD process and vice versa. This guarantees that when ATM and RAD start measuring the communication overhead, both sides are already available for the data exchange and the measurement procedure is not affected by the idle time of the faster process. At the end of an experiment, each ATM process (or RAD process) reports its own measurement. For example, if the model (using identical domain decomposition) allocates 1152 MPI processes, there will be 576 ATM processes (thus a set of 576 measurements for the communication overheads on ATM) and 576 RAD processes (thus another set of 576 measurements for the communication overheads on RAD). The maximum of each set is chosen as the representative of the communication overhead on ATM or RAD processes, respectively.

### Experiments

Figure 5.7 shows the costs of communication between ATM and RAD in the concurrent radiation scheme. The overhead is calculated with respect to the total simulation time. The blue and red curves exhibit the communication overheads measured at ATM' side and RAD's side, respectively. The green curve, however, indicates the effective communication cost on the simulation time.

When ECHAM6 with the concurrent radiation scheme allocates 48, 96 or 192 MPI processes, the communication costs of ATM and RAD are almost the same. Thus, the green curve (the effective communication cost) follows both the red curve and the blue curve. However, at 288 or 384 MPI processes, the communication cost at ATM's side is lower than at RAD's side. The effective communication cost is, nevertheless, the larger communication overhead experienced by RAD. Therefore, the green curve (the effective communication cost) follows the red curve. This because, at these setups, ATM and RAD are equally assigned 144 and 192 MPI processes, respectively, and, thus, RAD becomes relatively more expensive than ATM at these configurations - as shown in Figure 3.4. For this reason, ATM not only sees the communication overhead at its own side, but it is also affected by the larger communication cost at RAD's side. In other words, the larger communication cost at RAD's side causes RAD to wait longer relatively, and, thus catches up with ATM with a longer delay in the next communication point.

When the model allocates 576, 768 and 1152 MPI processes, ATM still experiences a lower communication overhead. This time, however, the green curve switches to the blue curve. This is because ATM and RAD are assigned 288, 384 and 576 MPI processes, respectively, and, thus, ATM becomes more expensive than RAD - as also shown in Figure 3.4. In such cases, ATM will not be delayed by the communication overhead at RAD's side any longer even if RAD experiences a longer communication

**Algorithm 5.1** Instrumenting ECHAM6 (using the concurrent radiation scheme) to measure the communication overhead between ATM and RAD.

**ATM (MPI) Process**

```
Program ECHAM6

  call timer_start (Total_simulationtime)

  call Initialization

  while (time steps < enough)

    call ATM_part_1

    if  (radiation time step) then

        call send_command_to_RAD (RESULTS)

        ! A barrier to make sure ATM and RAD
        ! are ready for communication and
        ! no waiting time affects the measurements.
        call barrier(intercomm)
        call timer_start (CommOverhead_on_ATM)
        call MPI_RECV (resultsdata, RAD, intercomm)
        call timer_stop (CommOverhead_on_ATM)

        call send_command_to_RAD (INPUT)
        call barrier(intercomm)
        call timer_start (CommOverhead_on_ATM)
        call MPI_SEND (inputdata, RAD, intercomm)
        call timer_start (CommOverhead_on_ATM)

    end if

    call ATM_part_2

  end while

 call send_command_to_RAD (FINISH)

  call cleanup

  call timer_stop (Total_simulationtime)

End Program ECHAM6
```

**RAD (MPI) Processes**

```
Program ECHAM6

  call Initialization

  while (TRUE)

    call receive_command_from_ATM

    select case (command)

        case (RESULTS)
          call barrier(intercomm)
          call timer_start (CommOverhead_on_RAD)
          call MPI_SEND (resultsdata, ATM, intercomm)
          call timer_stop (CommOverhead_on_RAD)

        case (INPUT)
          call barrier(intercomm)
          call timer_start (CommOverhead_on_RAD)
          call MPI_RECV (inputdata, ATM, intercomm)
          call timer_stop (CommOverhead_on_RAD)
          call radiation

        case (FINISH)
             exit

    end select
  end while

  call cleanup

End Program ECHAM6
```

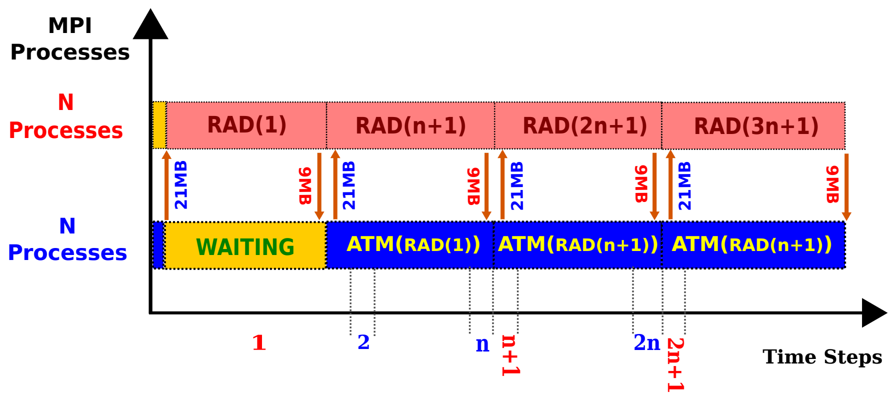


**Figure 5.6.:** The aggregated data volume transferred from the main model to the radiation component and vice versa in the concurrent radiation scheme.

cost. This is because RAD finishes much faster and it will race to idle despite its larger communication overhead, waiting for ATM to catch up. It is noteworthy that the communication cost of RAD rapidly increases towards the end albeit without any negative effect on the time-to-solution of the model.

In the nutshell, the green curve clearly indicates that the effective communication cost on the time-to-solution of the model is less than 0.60 percent of the total simulation time. This is a sheer advantage of the concurrent radiation scheme that imposes a negligible communication overhead on the overall performance of the model. Consequently, we should now focus only on analyzing the impacts of the load imbalance between ATM processes and RAD processes on the model's performance.

### 5.5.2. Measuring the idle times

A major factor that causes ATM to be delayed in the concurrent radiation scheme is the idle times of ATM processes due to the slow RAD processes. In identical domain decomposition, an idle time can be created before any data exchange between each ATM process and only its peer RAD process. In this setup, a peer RAD process is the only MPI process that calculates radiative transfer for the corresponding ATM process and responds only to the commands sent by that process. In finer domain decomposition, however, one ATM process may wait on multiple RAD processes. Conversely, in coarser domain decomposition, multiple ATM processes may be stalled by only one RAD process.

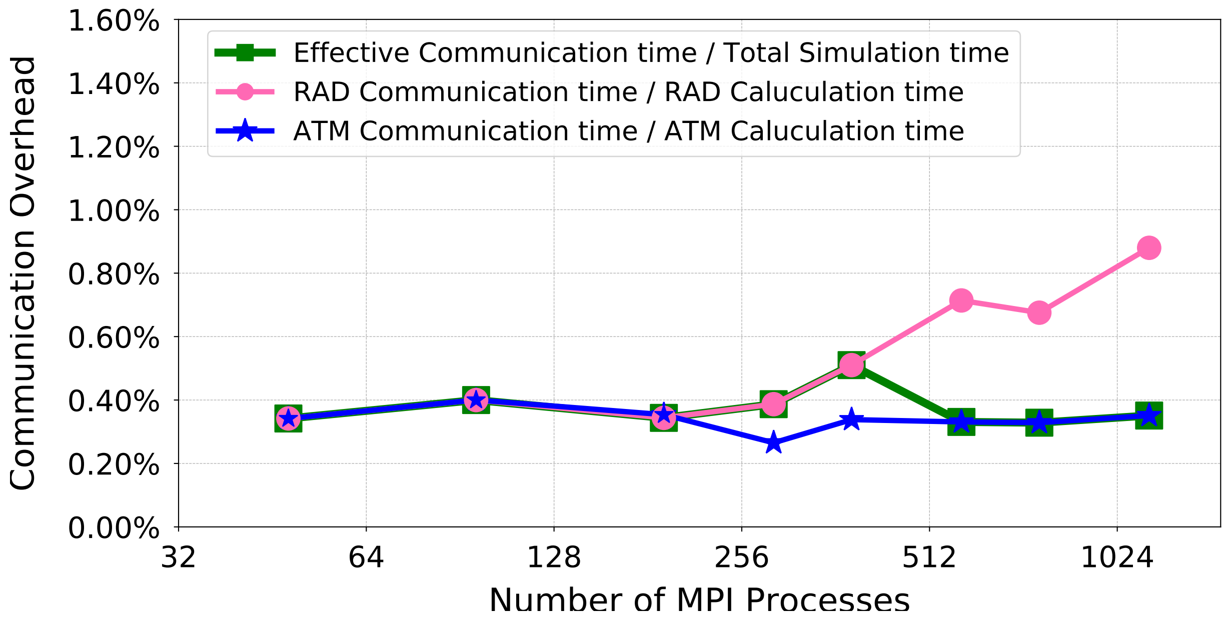


**Figure 5.7.:** Comparing the communication costs of data exchange between the radiation component (RAD) and the main model (ATM) respect to the overall simulation time. The green curve shows the effective communication overhead on the simulation time.

### Code instrumentation

Algorithm 5.2 suggests an approach for measuring the idle time of ATM and RAD processes. Although measuring the idle time of ATM processes is enough for finding the best time-to-solution of the model in the concurrent radiation scheme, we measure the idle time of RAD processes for the further tuning of the model (as discussed in the following sections). Since the model uses a client-server architecture for controlling RAD from ATM, the measurement procedure takes place before the command exchange between each ATM process and its peer RAD process. As soon as one process is ready for data exchange, it starts a timer to measure its idle time until the peer process arrives at the same point and becomes ready for the interaction. An MPI barrier guarantees that both parties are ready for the data exchange before any of them starts the command exchange. Algorithm 5.2 shows an abstract overview of the code instrumentation in ECHAM6 though the real source code is much more complex. At the end of an experiment, each MPI process reports its own idle time. If the model sets up identical domain decomposition at 1152 MPI processes, for example, the first set of 576 processes is assigned to ATM and the second set is assigned to RAD. Thus, two sets of 576 measurements can be collected from the reports. The maximum idle time of each set is chosen as the representative of the idle time of ATM or RAD processes, respectively.

### Experiments

After an accurate code instrumentation, several experiments are carried out to measure the idle time of ATM and RAD. Since the length of waiting time depends on the choice of domain decomposition (adopted by ATM and RAD) and allocated resources, experiments are repeated for each setup chosen in Section 5.4. Figure 5.8 depicts the results of these experiments. The blue and red curves show the respective idle times that ATM and RAD experience. The idle times appear when RAD is resolved faster or slower than ATM (thus a load imbalance between ATM processes and RAD processes appears). It should be noted that an idle time denotes the maximum length of time that an MPI process (assigned to RAD or ATM) has to wait for its peer MPI process until it catches up. In other words, the idle time can be different from one ATM process to the other, but the blue curve shows only the maximum value (which eventually contributes to overall runtime of the model). This is also true about the red curve that shows the maximum idle time of the RAD processes.

At lower numbers of MPI processes, as shown in Figure 5.8, calculating RAD takes longer, hence forcing ATM to wait relatively long for receiving feedback from RAD (in each radiation time step). At higher numbers of MPI processes, however, RAD scales better and finishes faster than ATM. It therefore has to wait for the arrival of the next radiation time step so that the radiation results can be transferred from RAD to ATM. Figure 5.8 reveals a few interesting points about the behaviors

**Algorithm 5.2** Instrumenting (the new version of) ECHAM6 (that uses the concurrent radiation scheme) to measure the idle time of MPI processes assigned to the main model (ATM) and radiation component (RAD). Each MPI process of ATM waits for peer process in RAD. The algorithm synchronizes the peer (MPI) processes to measure the length of time that they wait for each other. At the end of the experiment, each process reports its own idle time and the maximum idle time will contribute to the overall simulation time.

**ATM (MPI) Process**

```
Program ECHAM6

  call timer_start (Total_simulationtime)

  call Initialization

  while (time steps < enough)

    call ATM_part_1

    if  (radiation time step) then

        ! A barrier shows when both ATM and RAD
        ! are ready for data exchange
        call timer_start (ATM_idletime)
        call barrier(intercomm)
        call timer_stop (ATM_idletime)
        call send_command_to_RAD (RESULTS)
        call MPI_RECV (resultsdata, RAD, intercomm)

        ! A barrier shows when both ATM and RAD
        ! are ready for data exchange
        call timer_start (ATM_idletime)
        call barrier(intercomm)
        call timer_stop (ATM_idletime)
        call send_command_to_RAD (INPUT)
        call MPI_SEND (inputdata, RAD, intercomm)

    end if

    call ATM_part_2

  end while

  call send_command_to_RAD (FINISH)

  call cleanup

  call timer_stop (Total_simulationtime)

End Program ECHAM6
```

**RAD (MPI) Process**

```
Program ECHAM6

  call Initialization

  while (TRUE)

    ! A barrier shows when both ATM and RAD
    ! are ready for data exchange
    call timer_start (RAD_idletime)
    call barrier(intercomm)
    call timer_stop (RAD_idletime)

    call receive_command_from_ATM

    select case (command)

        case (RESULTS)
           call MPI_SEND (resultsdata, ATM, intercomm)

        case (INPUT)
           call MPI_RECV (inputdata, ATM, intercomm)
           call radiation

        case (FINISH)
           exit

    end select
  end while

  call cleanup

End Program ECHAM6
```

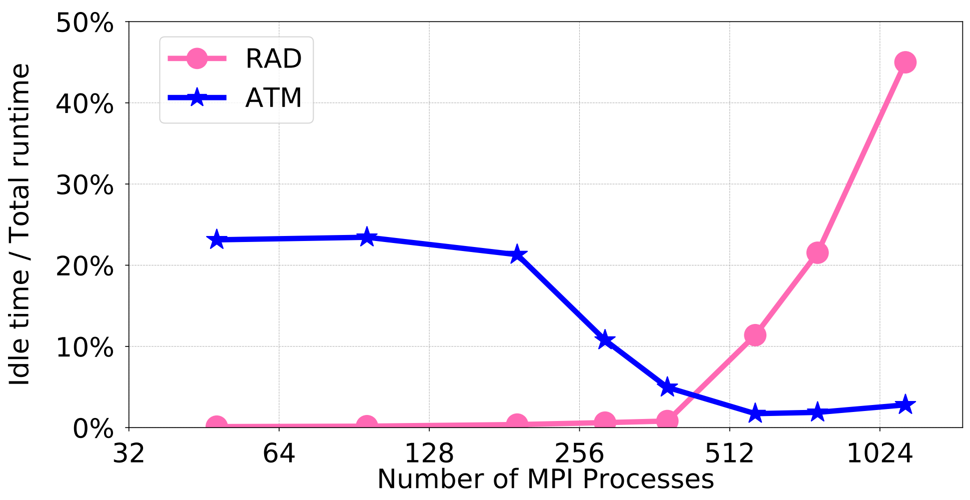


**Figure 5.8.:** The red curve (RAD) shows the increasing idle time of the MPI processes responsible for resolving radiative transfer. It suggests that RAD is a dominant computation in all the configurations before 576 MPI processes and does not wait for ATM. The blue curve (ATM), on the other hand, shows that resolving other atmospheric physics and dynamics is not a dominant computation for all the configurations before 576 MPI processes and ATM therefore experiences a long idle time. However, ATM becomes dominant towards the end, inflicting a long idle time on RAD. Yet the idle time of ATM is not lifted completely. This can be attributed mainly to the unavoidable waiting time of ATM in the first (radiation) time step (as reflected in Figure 3.6) and some infrequent (slightly) longer radiation time steps.

of ATM and RAD. Firstly, the idle time of RAD increases rapidly as the model keeps scaling. This behavior accounts for the higher scalability of the radiation component, which was already reflected in Figure 5.1. Secondly, at some setups such as 576, 768 and 1152 MPI processes, it seems RAD waits long for ATM to catch up. Thus, ATM intuitively is expected to have a zero idle time as RAD is already waiting for it. Surprisingly, however, the blue curve shows that ATM also experiences some delays. Despite the first impression that implies ATM and RAD are waiting for each other at the same time, a very thorough investigation shed light on the secret. It appeared that RAD has to perform some expensive I/O operations in some particular radiation time steps, and, thus, it becomes slower than ATM. That is why the idle time of ATM is not completely zero at such setups.

In addition, extra experiments disclosed some minor negative contributions of MPI barrier (used in the code instrumentation) to the experiments' results. Figure 5.9 and Figure 5.10 exhibit this problem clearly. The blue curve in Figure 5.9 shows

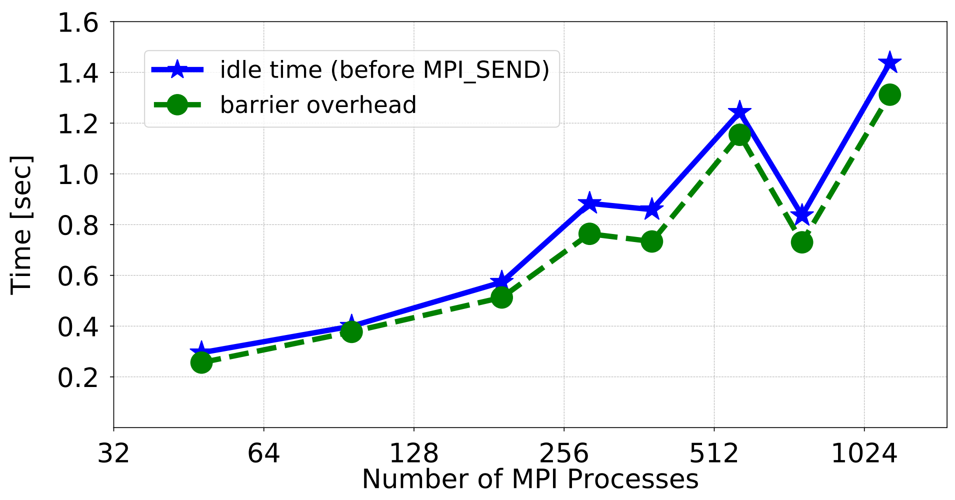


**Figure 5.9.:** The blue curve shows the idle time of ATM before sending input data to RAD. The green curve shows the cost of MPI_barrier used in measuring the idle time. Since the blue and green curves follow each other closely, they suggest that the cost of MPI barrier largely affects the measurement procedure. In other words, ATM experiences almost a zero waiting time before sending input data to RAD.

the idle time that ATM experiences before sending input data to RAD. As the graph suggests, the idle time increases as the model scales. The green curve shows the cost of using MPI barrier for measuring the idle time. Interestingly, the green curve follows the blue curve very closely, implying that what the blue curve shows is largely due to the barrier. In other words, the idle time of ATM is, in fact, almost zero. By the same token, the blue curve in Figure 5.10 shows that the idle time of ATM (before receiving the results from RAD) is decreasing but it does not become zero completely. The green curve, on the other hand, shows that the cost of MPI barrier for measuring the idle time increases as the model scales. Since the cost of barrier contributes directly to the measurement of the idle time, the green curve partly justifies non-zero idle times of ATM at 576, 768 or 1152 MPI processes.

### 5.5.3. Discussions

This section opens with a question as to whether Figure 5.1 presents a correct overview of the performance of the concurrent radiation scheme or it can be improved by a better model's configuration. It was already clarified that there are two main

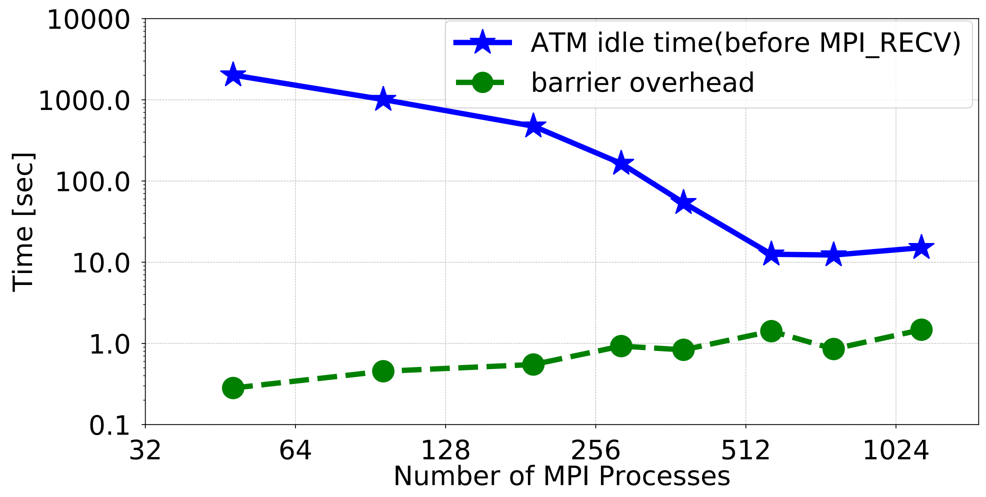


**Figure 5.10.:** The blue curve shows the idle time of ATM before receiving the results from RAD is decreasing but is not removed completely. The green curve shows the cost of MPI_barrier in measuring the idle time, which increases as the model scales. This partly justifies why the idle time of ATM does not become completely zero towards the end.

contributors to degrading the performance of the model using the new scheme: data communication and load imbalance. However, the evidence from Figure 5.7 proved that data exchange between ATM and RAD has little impact on the performance of the model. Figure 5.8. Particularly, the blue curve in Figure 5.8 indicates the idle time imposed on ATM at 48, 96, 192, 288 and 384 MPI processes. Such an idle time stalls ATM repeatedly during the course of simulation. Hence, the time-to-solution of the model can be improved if the idle time is reduced or ideally removed completely. Since RAD experiences a fairly low communication overhead at these setups (as shown by the red curve in Figure 5.7), it can safely be concluded that the idle time is largely due to the expensive calculations of RAD. Since RAD scales well, finer domain decomposition at RAD's side is expected to expedite the overall simulation time.

The graph in Figure 5.11 is almost the same as Figure 5.1, but the added green curve shows the asymptotic performance of the concurrent radiation scheme. The model achieves its asymptotic performance if the cost of RAD is removed completely from the model. This can be calculated from the curves in Figure 3.4 by removing the cost of RAD (shown by the red curve) and considering the cost of ATM (shown by the blue curve) only. It can be inferred from the green curve that if the concurrent radiation scheme uses finer domain decomposition for RAD, there will be some performance improvements albeit negligible.

Additionally, at setups such as 576, 768 and 1152 MPI processes, no performance improvement can be expected. This can be learned from both Figure 5.8 and

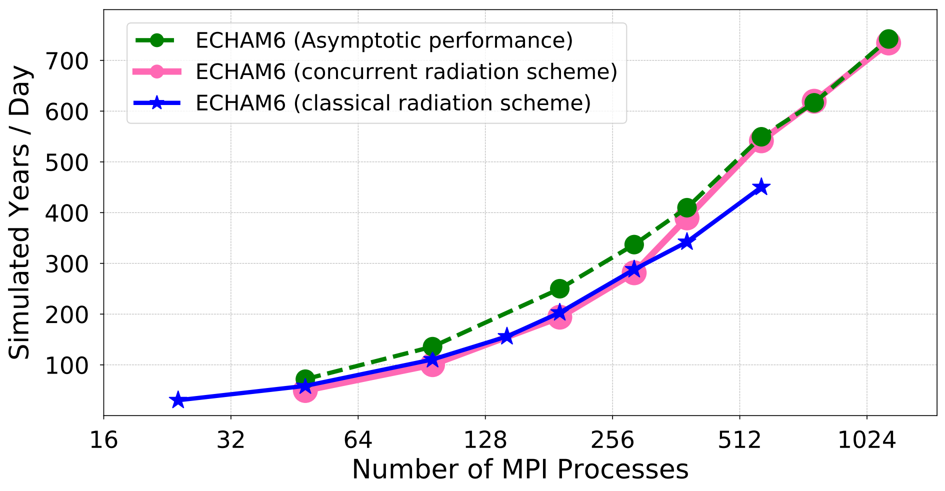


**Figure 5.11.:** This graph resembles Figure 5.1, but the green curve was added to show the asymptotic performance of the concurrent radiation scheme. This curve shows the best performance of the model assuming that the cost of calculating RAD were ideally zero. The asymptotic performance can be calculated from Figure 3.3. The green curve suggests that the performance of the model can be improved at 48, 96, 192, 288 and 384 MPI processes albeit negligible. At 576, 768 and 1152, nevertheless, the concurrent radiation scheme has already achieved its asymptotic performance.

Figure 5.11. The blue curve in Figure 5.8 indicates that ATM waits no longer on RAD (as RAD finally catches up), thus any finer domain decomposition at RAD will not attain any performance improvement. This is also confirmed by the green curve in Figure 5.11 that shows the concurrent radiation scheme has already been able to deliver its asymptotic performance as these setups. On this account, we can safely conclude that the best time-to-solution that the concurrent radiation scheme can achieve is the same as the one delivered at 1152 MPI processes. It is worth emphasizing that such a performance might be achievable at coarser domain decomposition (at RAD) as well. In other words, if ATM keeps allocating 576 MPI processes, it should be possible to run RAD at a lower number of resources such that the best time-to-solution is still attainable.

## 5.5.4. Resource efficiency

This section presents a discussion on the efficiency of the resource utilization of the concurrent radiation scheme. From the discussion in the last section, it was concluded that we have already found the shortest time-to-solution of the concurrent radiation scheme. Among multiple possible setups, it was argued that this performance is achievable (at least) if ATM and RAD adopt identical domain decomposition and each component is assigned 576 MPI processes (i.e. a total of 2x576=1152 resources are assigned to the model). However, Figure 5.8 indicated that RAD experiences a long idle time, suggesting a load imbalance between ATM processes and RAD processes and thus an inefficient resource utilization. This section calculates the resource efficiency of the model for the same setups used in Figure 5.1. A prerequisite to this is, however, the calculation of the speedup that ECHAM6 achieves using the concurrent radiation scheme (regarding the aforementioned configurations). On this account, we first calculate the speedup and then present the parallel efficiency of the model.

#### 5.5.4.1. Methodical Speedup

Figure 5.12 displays the *methodical speedup* of the model across the scaling curves (shown in Figure 5.1). In contrast to the classical definition of speedup where additional resources are used by the same computation, the methodical speedup compares the performance of two different computations on different number of resources. In other words, the methodical speedup compares the simulation time ($T_p$) of the the model using the concurrent radiation scheme to the simulation time ($T_s$) of the original model using the sequential radiation scheme. These two simulations are performed with different computations (which are expected to end up in non-identical results). For each comparison, *2N* MPI processes are allocated by the model in the concurrent radiation scheme (i.e. a set of $N$ MPI processes is assigned to ATM and another set of $N$ MPI processes is assigned to RAD). In the classical radiation scheme, however, only one set of $N$ MPI processes is allocated by the whole model. In other words, the methodical speedup $S$ is defined as below:

Ts = maximum simulated years per day with the classical radiation scheme (using $N$ resources)

Tp= maximum simulated years per day with the concurrent radiation scheme (using *2N* resources)

$$\text{Methodical Speedup} = S = \frac{Ts}{Tp}$$

The red curve in Figure 5.12 displays the actual (methodical) speedup of the model for the same setups chosen for Figure 5.1. The horizantal axis indicates only the

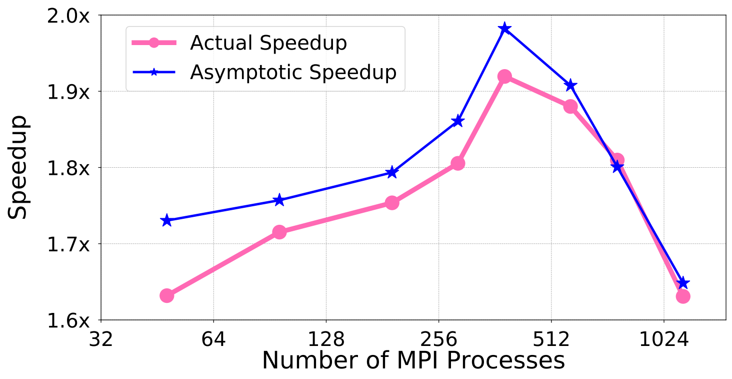


**Figure 5.12.:** The *methodical* speedup of ECHAM6 using the concurrent radiation scheme and comparing it with the asymptotic speedup.

total number of MPI processes that the model allocates once it adopts the concurrent radiation scheme. However, the model allocates half of the MPI processes shown by the horizantal axis when it adopts the the classical radiation scheme. From this figure, it can be inferred that the concurrent radiation scheme achieves an actual speedup ranging from a minimum 1.6x to over 1.9x. The blue curve, however, shows the asymptotic speedup that the model would achieve if there were no cross-dependency, and thus no communication latency, between the radiation component and the main model. Interestingly, the red and blue curve confirm that the concurrent radiation scheme has already achieved its asymptotic performance at 768 and 1152 MPI processes and no further tuning is imaginable, as also suggested by the discussion in Section 5.5.3.

#### 5.5.4.2. Parallel Efficiency

Figure 5.13 shows the efficiency of resource utilization of the model using the concurrent radiation scheme. The parallel efficiency of the concurrent radiation scheme is defined as the ratio of the methodical speedup $S$ to the relative number of the allocated resources $R$, as shown below.

$$R \text{ (ratio of allocated resources)} = \frac{\text{(resources assigned to the concurrent radiation)}}{\text{(resources assigned to the classical radiation)}} = \frac{2N}{N} = 2$$

$$E \text{ (parallel efficiency)} = \frac{S}{R} = \frac{S}{2}$$

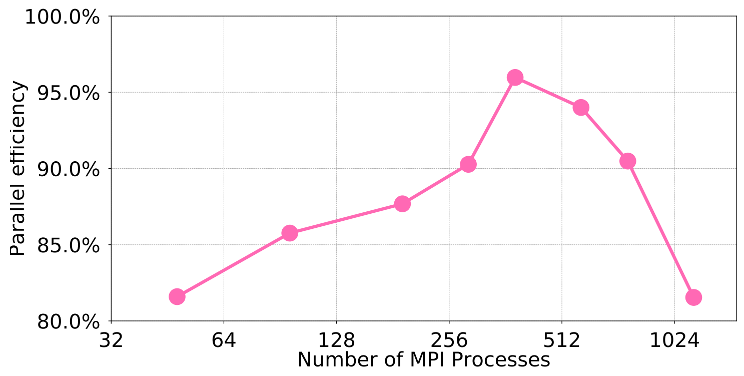


**Figure 5.13.:** The parallel efficiency of ECHAM6 (using the concurrent radiation scheme).

As visible in the figure, the model achieves a minimum parallel efficiency of 80% across the same scaling curve shown in Figure 5.1. The most efficient resource utilization takes place at 384 MPI processes as ATM and RAD experience a minimum idle time at this configuration (as shown in Figure 5.8). While the new scheme has an acceptable resource utilization, an effective load balancing (using finer or coarser domain decomposition in RAD) is expected to push it to the maximum efficiency at almost all the setups across the scaling curve..

## 5.6. Chapter Summary

This chapter presents a detailed performance analysis of the atmospheric model ECHAM6 using the concurrent radiation scheme (which was developed within this dissertation) and compares it with the performance of the model using the classical radiation scheme. The goal is to find the setups that offer the best time-to-solution of the model for paleoclimate simulations within the PalMod project with a minimum resource usage. It was shown when ECHAM6 adopts the concurrent radiation scheme and applies identical domain decomposition to both the main model and the radiation component, it can achieve the shortest time-to-solution at a configuration in which 1152 MPI processes are assigned to the whole model (i.e. the same number of 576 MPI processes is allocated by the main model and the radiation component). However, it was indicated that ECHAM6 cannot achieve such a performance with the classical radiation scheme. Additionally, it was argued that this configuration is not resource efficient as the radiation component experiences a large idle time. Hence, it should be possible to attain the same time-to-solution and a better resource efficiency at a configuration that provides coarser domain decomposition (and thus requiring a lower number of MPI processes) to the radiation component, but assigning the same number of 576 MPI processes to the main model. In the nutshell, it was demonstrated that the concurrent radiation scheme can achieve a maximum speedup of 1.9x over the classical scheme with a minimum of 80% parallel efficiency.

# 6. Component Isolation

*This chapter introduces a novel static program analysis approach to achieve the general goal of this dissertation. The discussion is started by providing solid definitions that are needed in this chapter and then continues by exploring the complexity aspects of a component. We will then proceed to describe the procedures of Component Extraction and Component Isolation, augmented by a solution for extracting the shared variables that couple a component to a Fortran program.*

## 6.1. Definition of a Component

This section provides some preliminaries definitions which are required in the discussions in this chapter. In addition, some symbols will be defined here which will be used throughout the chapter for further discussions.

### Subprogram:

A Fortran subprogram in this dissertation denotes a Fortran procedure (either a Fortran subroutine or function).

### Control Flow Graph (CFG):

A control-flow graph (CFG) of a program is a directed graph that represents how the control of execution of the program changes (Allen, 1970). The nodes of a CFG capture blocks of unconditional statements and its arcs represent the transfer of control from one block to another (through unconditional and conditional branches).

### Component and Entry Point:

**Definition:** Let $P$ be a Fortran program and $G$ be the control flow graph (CFG) of $P$. In addition, let $C$ be the set of the subgraphs of $G$ which are reachable from some known Fortran subprograms $e_1$, ..., $e_k$ in $P$. Hence, $C$ is called ***one Fortran software component*** or in brief ***one component*** of $P$ and $e_1$, ..., $e_k$ are called ***the entry points of the component.*** In addition, $P$ is called the ***containing Fortran program of the component***.

**Carvedout Program:**

**Definition:** Let $P$ be a Fortran program containing the component $C$. Also, let $P_m$ *be* a *modified* version of $P$ by removing all the call-sites to the entry points of $C$ from $P$. Let $G'$ be the control flow graph (CFG) of $P_m$. In this dissertation, the ***carvedout program*** $P_{co}$ denotes a slice of $P_m$ by removing all the program statements (from $P_m$) that are not reachable in $G'$ from the main program unit of $P_m$.

**Syntactic Program Statement and Source Code:**

**Definition:** Let $P$ be a Fortran program containing the component $C$ and let $P_{co}$ be the carvedout program of $P$. Also, let $G$ and $G'$ be the control flow graphs (CFG) of $P$ and $P_{co}$, respectively.

***A syntactic program statement of the component*** denotes a program statement in $P$ that is reachable in $G$ from at least one entry point of $C$. Hence, ***the syntactic source code or content of the component*** shall denote the set of such syntactic program statements.

By the same token, ***a syntactic program statement of the carvedout program*** denotes a program statement in $P$ that is reachable in $G'$ from the main program unit of $P_{co}$. Hence, ***the syntactic source code or content of the carvedout program*** shall denote the set of such syntactic program statements.

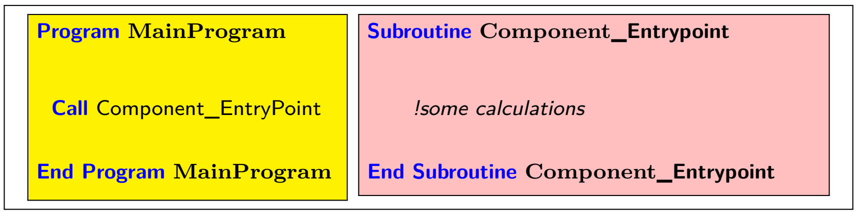
In this chapter, unless explicitly emphasized, a program statement of a component or a carvedout program always denotes a syntactic program statement of the component or the carvedout program, respectively. By the same token, the source code or the content of a component or a carvedout program always denotes the syntactic source code or content of the component or the carvedout program, respectively.

**Color codes:**

In this chapter, we use some color codes in the examples of Fortran programs to distinguish the source code of a component from the carvedout program . Figure 6.1 shows a table defining the meaning of each color code.

| Color | Meaning |
|---|---|
| **Pink** | dedicated source code of a component |
| **Yellow** | dedicated source code of the carvedout program |
| **Teal** | Shared source code between a component and the carvedout program |
| **Grey** | dead source code in a Fortran program |
| **White** | Undecided |

**Figure 6.1.:** Color codes show whether the source code belongs to a component or the carvedout program in a Fortran program.



```
Program MainProgram

  Call Component_EntryPoint

End Program MainProgram
```

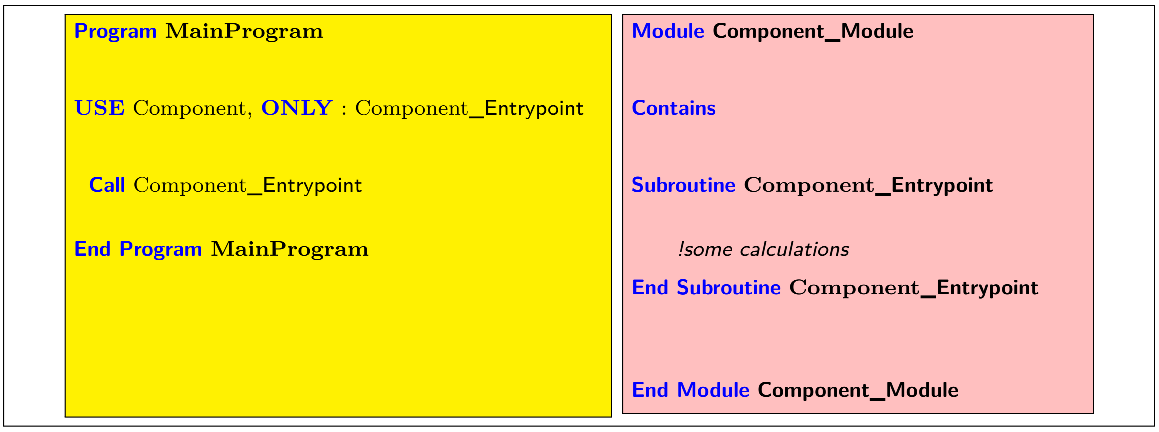
```
Subroutine Component_Entrypoint

      !some calculations

End Subroutine Component_Entrypoint
```

**Figure 6.2.:** A Fortran program with one component. The component has the simplest form. It has a single entry point. The entry point of the component is a Fortran subroutine (outside any Fortran module) and contains the entire source code of the component as it does not make a call to any other subprogram.

## 6.2. Examples of a Component

Figure 6.2 shows a component in the simplest form as the component includes only one single Fortran subroutine. In this example, *"Component_Entrypoint"* is the only entry point of the component and includes the whole source code of the component. However, the source code of a component may be spread well beyond its entry point. Figure 6.3 shows another Fortran program with a component but the source code of the component is contained in a Fortran module. The entry point of the component is a subroutine named "*Component_Entrypoint*" and the source code of the component is spread in a Fortran module called "*Component_Module*".

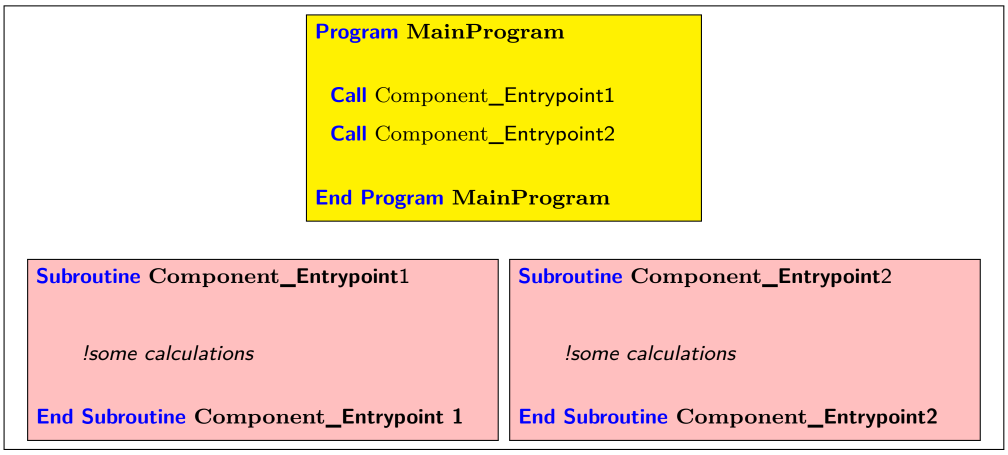
A component can indeed become much more complex and come in a variety of combinations of the two simple forms presented in Figure 6.2 and Figure 6.3. In addition, a component may have one or multiple entry points to provide multiple functionality to the main program. Figure 6.4 shows one component with two entry points that are only Fortran subroutines. By the same token, Figure 6.5 shows another component with two entry points, but the component includes only Fortran modules and the entry points are the Fortran subroutines defined within the modules.



```
Program MainProgram

USE Component, ONLY : Component_Entrypoint

  Call Component_Entrypoint

End Program MainProgram
```

```
Module Component_Module

Contains

Subroutine Component_Entrypoint

      !some calculations
End Subroutine Component_Entrypoint


End Module Component_Module
```

**Figure 6.3.:** A Fortran program with one component. The component has a single entry point. The entry point of the component is a Fortran subroutine in a Fortran module. The source code of the component is spread beyond the entry point inside the Fortran module.

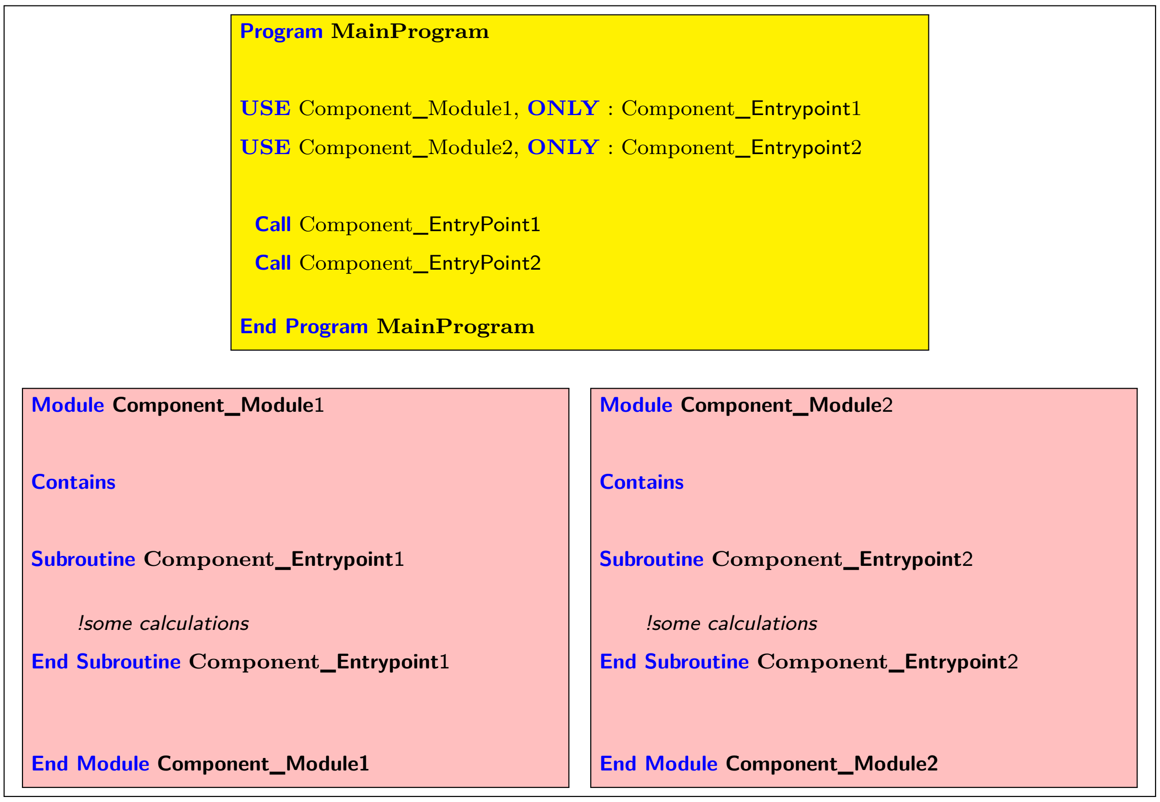


```
Program MainProgram

  Call Component_Entrypoint1
  Call Component_Entrypoint2

End Program MainProgram
```

```
Subroutine Component_Entrypoint1

      !some calculations

End Subroutine Component_Entrypoint 1
```

```
Subroutine Component_Entrypoint2

      !some calculations

End Subroutine Component_Entrypoint2
```

**Figure 6.4.:** One component with two entry points. The component includes only Fortran subroutines.

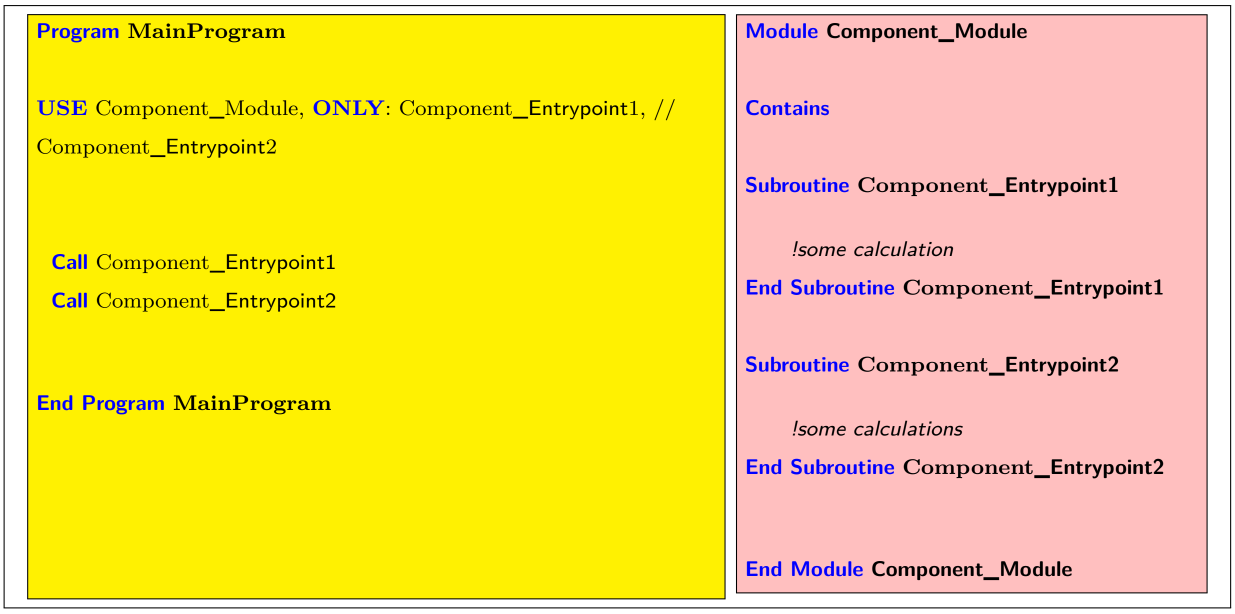


```
Program MainProgram

USE Component_Module1, ONLY : Component_Entrypoint1
USE Component_Module2, ONLY : Component_Entrypoint2

  Call Component_EntryPoint1
  Call Component_EntryPoint2

End Program MainProgram
```

```
Module Component_Module1

Contains

Subroutine Component_Entrypoint1

      !some calculations
End Subroutine Component_Entrypoint1


End Module Component_Module1
```

```
Module Component_Module2

Contains

Subroutine Component_Entrypoint2

      !some calculations
End Subroutine Component_Entrypoint2


End Module Component_Module2
```

**Figure 6.5.:** One component with two entry points. The component includes only Fortran modules. The entry points of the component are the Fortran subroutines defined inside the modules.



```
Program MainProgram

USE Component_Module, ONLY: Component_Entrypoint1, //
Component_Entrypoint2


  Call Component_Entrypoint1
  Call Component_Entrypoint2


End Program MainProgram
```

```
Module Component_Module

Contains

Subroutine Component_Entrypoint1

      !some calculation
End Subroutine Component_Entrypoint1

Subroutine Component_Entrypoint2

      !some calculations
End Subroutine Component_Entrypoint2


End Module Component_Module
```

**Figure 6.6.:** Component with multiple entry points: The entry points are subroutines of one module.

When a component has multiple entry points, they may be completely isolated from each other or share source code in part. The next section analyzes variants of a component in detail.

## 6.3. Variants of a Component

In real world applications, software components may exist in a wide variety of shapes and forms. There are several factors that affect the complexity of a component, which are as follows:

1. The type of a component
2. The depth of a component
3. Program sharing
4. The width of a component

These different aspects of a component are described in the following sections.

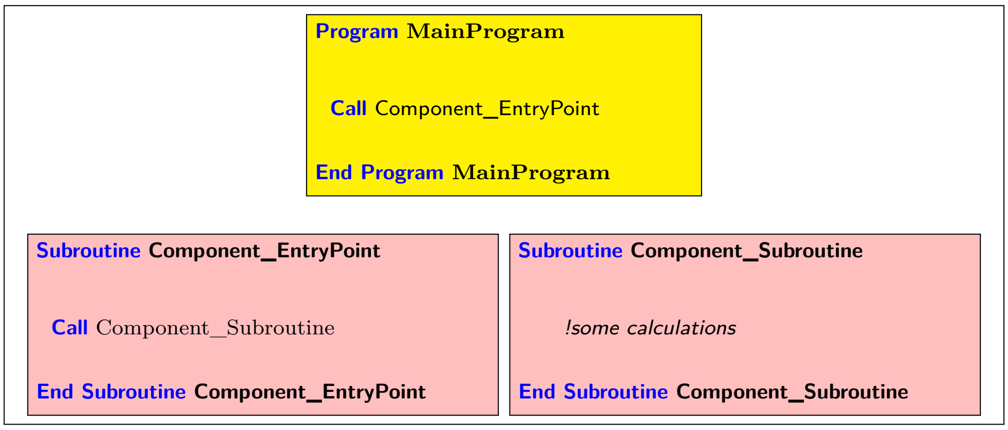
### 6.3.1. The type of a component

One component can have one of the three types:

1. It might include only Fortran subprograms, as shown in Figure 6.2, Figure 6.4 and Figure 6.7.
2. It might include only Fortran modules, as shown in Figure 6.3, Figure 6.5, Figure 6.6, and Figure 6.8.
3. It might include a combination of Fortran subprograms and modules. One example of combined type components is shown in Figure 6.9. However, a combined type component may appear in various combinations of Fortran subprograms and modules.

### 6.3.2. The width of a component

The width of a component is equal to the number of entry points of the component. Each entry point of the component offers a new functionality to the main program. The width of a component is at least 1 when the component has a single entry point. The components in Figure 6.2, Figure 6.3, Figure 6.9, Figure 6.7 and Figure 6.8 have a width of 1 and the components in Figure 6.4, Figure 6.5 and Figure 6.6 have a have a width of 2.

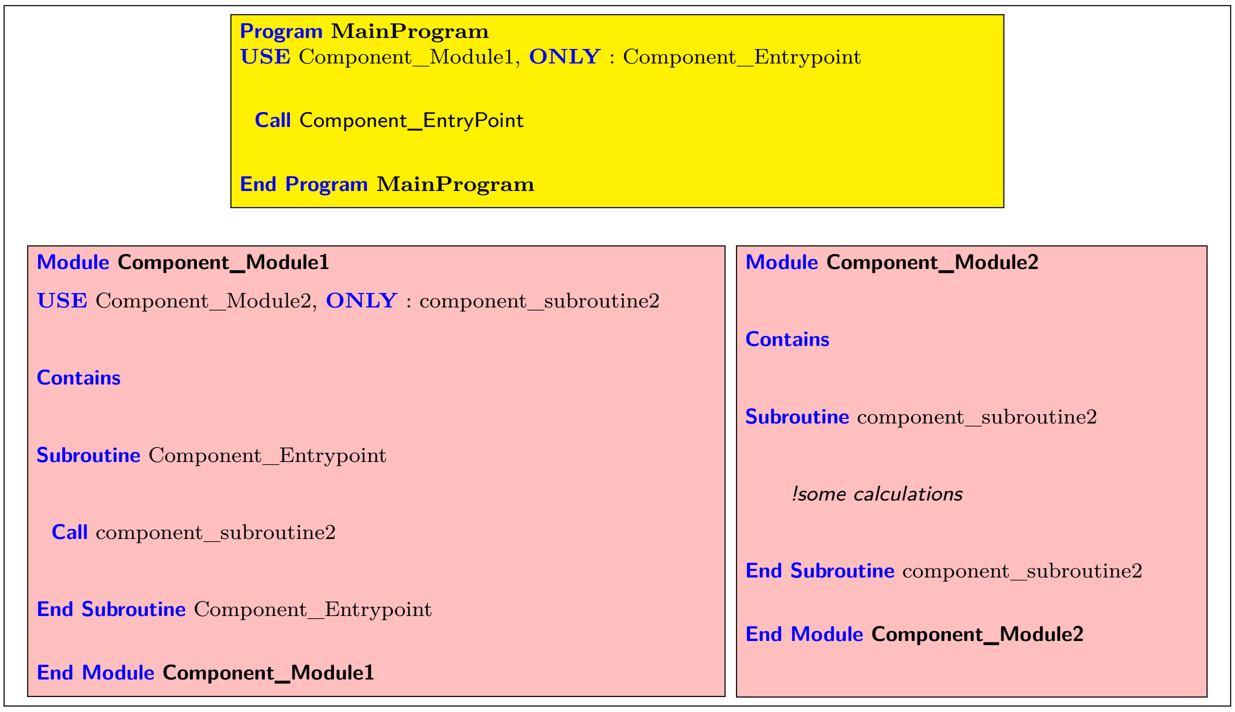


```
Program MainProgram

  Call Component_EntryPoint

End Program MainProgram
```

```
Subroutine Component_EntryPoint

  Call Component_Subroutine

End Subroutine Component_EntryPoint
```

```
Subroutine Component_Subroutine

      !some calculations

End Subroutine Component_Subroutine
```

**Figure 6.7.:** One component that contains only (two) Fortran subprograms.

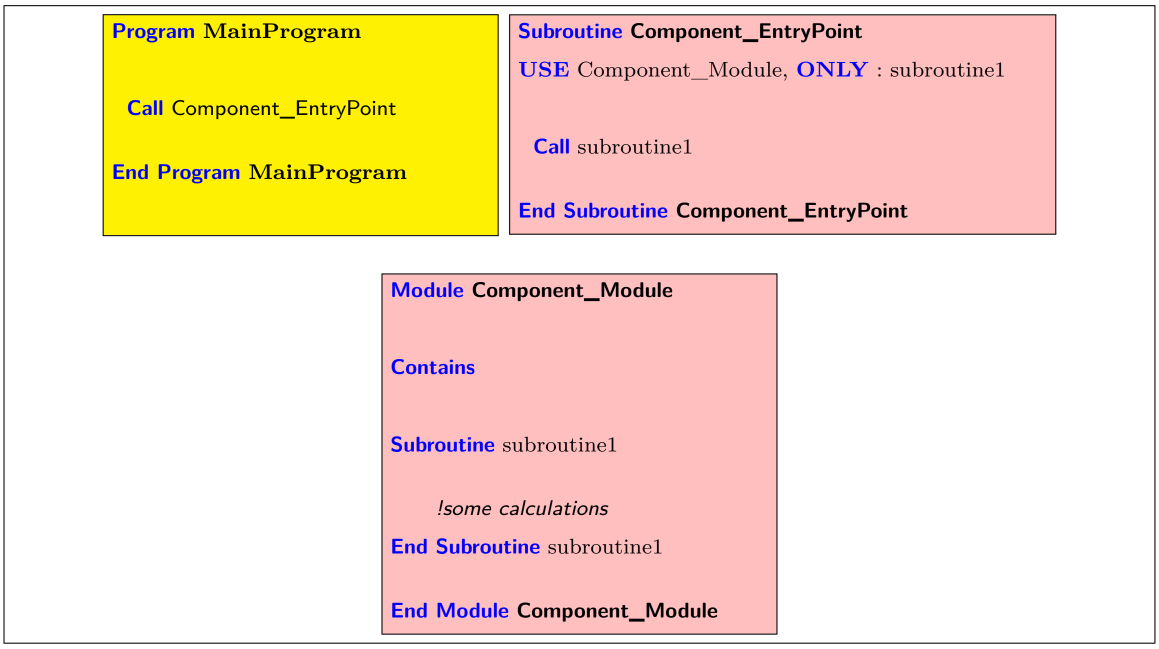


```
Program MainProgram
USE Component_Module1, ONLY : Component_Entrypoint

  Call Component_EntryPoint

End Program MainProgram
```

```
Module Component_Module1
USE Component_Module2, ONLY : component_subroutine2

Contains

Subroutine Component_Entrypoint

  Call component_subroutine2

End Subroutine Component_Entrypoint

End Module Component_Module1
```

```
Module Component_Module2

Contains

Subroutine component_subroutine2

      !some calculations

End Subroutine component_subroutine2

End Module Component_Module2
```

**Figure 6.8.:** One component that contains only (two) Fortran modules.



```
Program MainProgram

  Call Component_EntryPoint

End Program MainProgram
```

```
Subroutine Component_EntryPoint
USE Component_Module, ONLY : subroutine1

  Call subroutine1

End Subroutine Component_EntryPoint
```

```
Module Component_Module

Contains

Subroutine subroutine1

      !some calculations
End Subroutine subroutine1

End Module Component_Module
```

**Figure 6.9.:** One component with a combined type. The component includes one Fortran subroutine and one Fortran module. A combined type component may include any combination of Fortran subprograms and modules.

## 6.3.3. Program Sharing

Program sharing between a component and a containing Fortran program takes place in different forms, which are as follows:

- program statement sharing
- program namespace sharing
- program file sharing

### 6.3.3.1. Program Statement Sharing

Any program statement in a Fortran program that contains a component has one of the following status regarding the component and its corresponding carvedout program:

- non-shared
- shared
- dead

**Shared and non shared program statements:**
A shared program statement between a component (in a Fortran program) and its corresponding carvedout program is defined as below:

**Definition**. Let $P$ be a Fortran program containing the component $C$ and let $P_{co}$ be the carvedout program of $P$. Also, let $G$ and $G'$ be the control flow graphs (CFG) of $P$ and $P_{co}$, respectively.

In this chapter, ***a shared program statement*** denotes a program statement of $P$ that is reachable in $G$ from at least one entry point of $C$ and reachable in $G'$ from the main program unit of $P_{co}$.

A **non-shared program statement of the component** denotes a program statement in $P$ that is reachable in $G$ from at least one entry point of $C$, but not reachable in $G'$ from the main program unit of $P_{co}$.

By the same token, a **non-shared program statement of the carvedout program** denotes a program statement in $P$ that is reachable in $G'$ from the main program unit of $P_{co}$, but not reachable in $G$ from none of the entry points of $C$.

It is noteworthy that the set of shared program statements is denoted as ***the shared source code or content*** and the set of non-shared program statements is denoted as ***the non-shared source code or content***.

**Dead program statement:**

A dead program statement may exists in a component (in a Fortran program) or in its corresponding carvedout program. A dead program statement is defined as below:

**Definition.** Let $P$ be a Fortran program containing the component $C$ and let $P_{co}$ be the carvedout program of $P$. Also, let $G$ and $G'$ be the control flow graphs (CFG) of $P$ and $P_{co}$, respectively.

***a dead program statement w.r.t. the component*** denotes a program statement of $P$ that is not reachable in $G$ from none of the entry points of $C$.
***a dead program statement w.r.t. the carvedout program*** denotes a program statement of $P$ that is not reachable in $G'$ from the main program unit of $P_{co}$.
***a dead program statement w.r.t. the program*** denotes a program statement of $P$ that is not reachable in $G$ from the main program unit of $P$.

It is also noteworthy that ***dead content*** denotes a set of dead program statements. Dead program statements appear in two main forms, which are as follows:

- dead program declarations:
  Examples of such program statements are the definitions of variables, derived data types and etc. that are dead w.r.t. a component or its corresponding carvedout program.
- dead program instructions:
  An example of such program instructions can be found in the subprograms that provide different services to the component and the carvedout program.

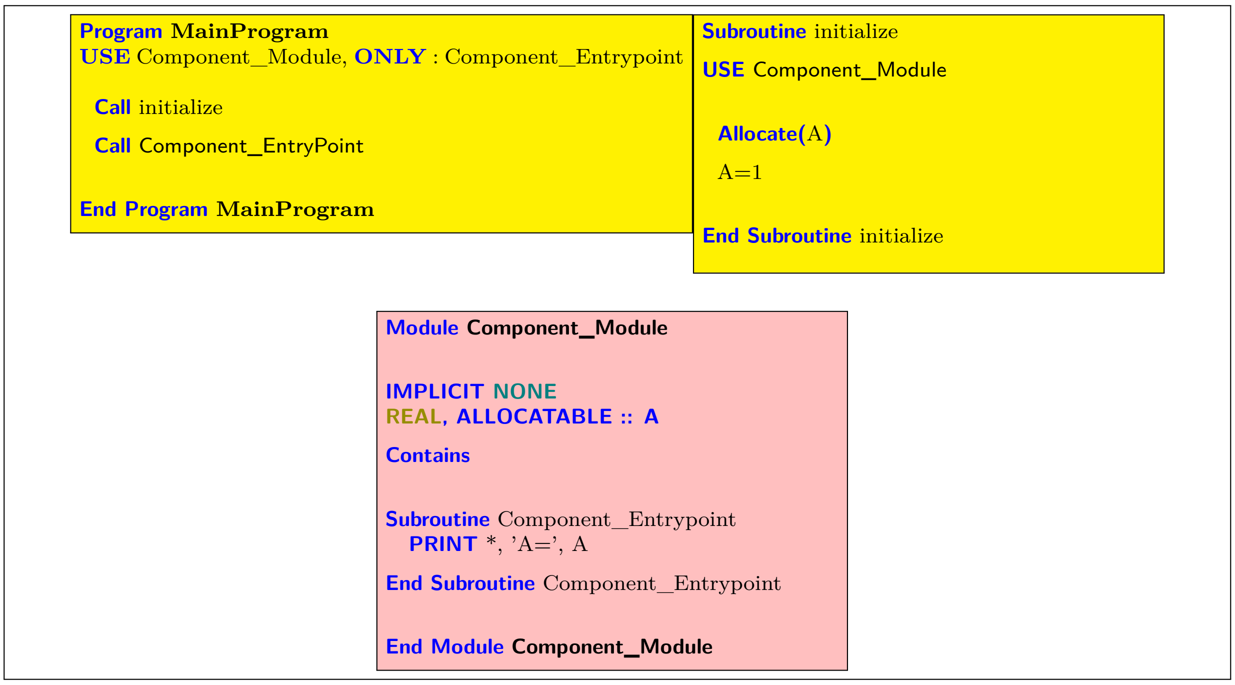
Furthermore, a non-shared program statement that is not a (syntactic) program statement of the component may still have a special importance to the component. These program statements are denoted as the semantic program statement of the component and defined as below:

**Definition.** Let $P$ be a Fortran program containing the component $C$ and let $P_{co}$ be the carvedout program of $P$. Also, let $G$ and $G'$ be the control flow graphs (CFG) of $P$ and $P_{co}$, respectively. In addition, let $s$ and $s'$ be two program statements of $P$, but let $s$ be a syntactic program statement of $C$ and let $s'$ be a non-shared program statement of $P_{co}$.

It is said that $s'$ is ***a semantic program statement*** of $C$ if there is a data flow dependency from $s'$ to $s$.

Although semantic program statements of a component are not syntactically reachable from the entry points of the component in the control flow graph of the program, they are semantically important to the component. A missing semantic program statement of a component will not lead to a compilation failure though it can potentially affect the correct execution of the component. For example, a semantic program statement modifies variables of a component, allocates the dynamic variables of the component or opens a file that the component writes to or reads from.

Figure 6.10 shows a Fortran program which has a component with an entry point called *"Component_Entrypoint"*. There is one dynamic variable called *"A"* defined in the component. The variable is allocated inside a subroutine called *"initialize"*. This subroutine is not, however, reachable from the entry point of the component (in the CFG of the program). Thus, the memory allocation instruction is not considered a syntactic program statement of the component, but rather a semantic program statement of the component.



```
Program MainProgram
USE Component_Module, ONLY : Component_Entrypoint

  Call initialize

  Call Component_EntryPoint

End Program MainProgram
```

```
Subroutine initialize

USE Component_Module

  Allocate(A)

  A=1

End Subroutine initialize
```

```
Module Component_Module

IMPLICIT NONE
REAL, ALLOCATABLE :: A

Contains

Subroutine Component_Entrypoint
   PRINT *, 'A=', A

End Subroutine Component_Entrypoint

End Module Component_Module
```

**Figure 6.10.:** The semantic program statement of the component.

### 6.3.3.2. Program Namespace Sharing

In this dissertation, we recognize three different namespaces in a Fortran program which are defined below:

**Definition.** Let $P$ be a Fortran program and $C$ a component in $P$. **A subprogram namespace** in $P$ denotes the definition of a Fortran subprogram in $P$. Also **a module namespace in** $P$ denotes the definition of a Fortran module in $P$. In addition, **the global namespace of** $P$ denotes the source code of $P$ excluding all the definitions of the Fortran subprograms and modules in $P$. Furthermore, **a namespace of the component** $C$ denotes a namespace of $P$ in which the source code of $C$ resides.

In a Fortran program containing a component, every namespace has one of the following status regarding the component and its corresponding carvedout program:

- shared
- non-shared
- dead

## Shared and non-share namespace:

A shared or non-shared namespace between a component (in a Fortran program) and its corresponding carvedout program is defined as below:

**Definition.** Let $P$ be a Fortran program containing the component $C$ and let $P_{co}$ be the carvedout program of $P$. Also, let $G$ and $G'$ be the control flow graphs (CFG) of $P$ and $P_{co}$.

A ***shared namespace*** denotes the namespace $N$ in $P$ that contains at least one syntactic program statement of $P$ which is reachable in $G$ from the entry points of $C$ and contains at least one syntactic program statement of $P$ which is reachable in $G'$ from the main program unit of $P_{co}$.

A ***shared namespace with shared content*** denotes the shared namespace $N$ in $P$ in which every syntactic program statement is reachable in $G$ from at least one of the entry points of $C$ and also reachable in $G'$ from the main program unit of $P_{co}$. Otherwise, it is denoted as ***a shared namespace with shared and non-shared content***.

Conversely, a ***non-shared namespace*** denotes the namespace $N$ in $P$ in which all the syntactic program statements are only reachable in $G$ from any of the entry points of $C$ or only reachable in $G'$ from the main program unit of $P_{co}$.

Furthermore, a ***dead namespace w.r.t the component*** denotes the namespace $N$ in $P$ in which there is no single syntactic program statement of $P$ that is reachable in $G$ from any of the entry points of $C$.

Moreover, a ***dead namespace w.r.t the carvedout program*** denotes the namespace $N$ in $P$ in which there is no single syntactic program statement of $P$ that is reachable in $G'$ from the main program unit of $P_{co}$.

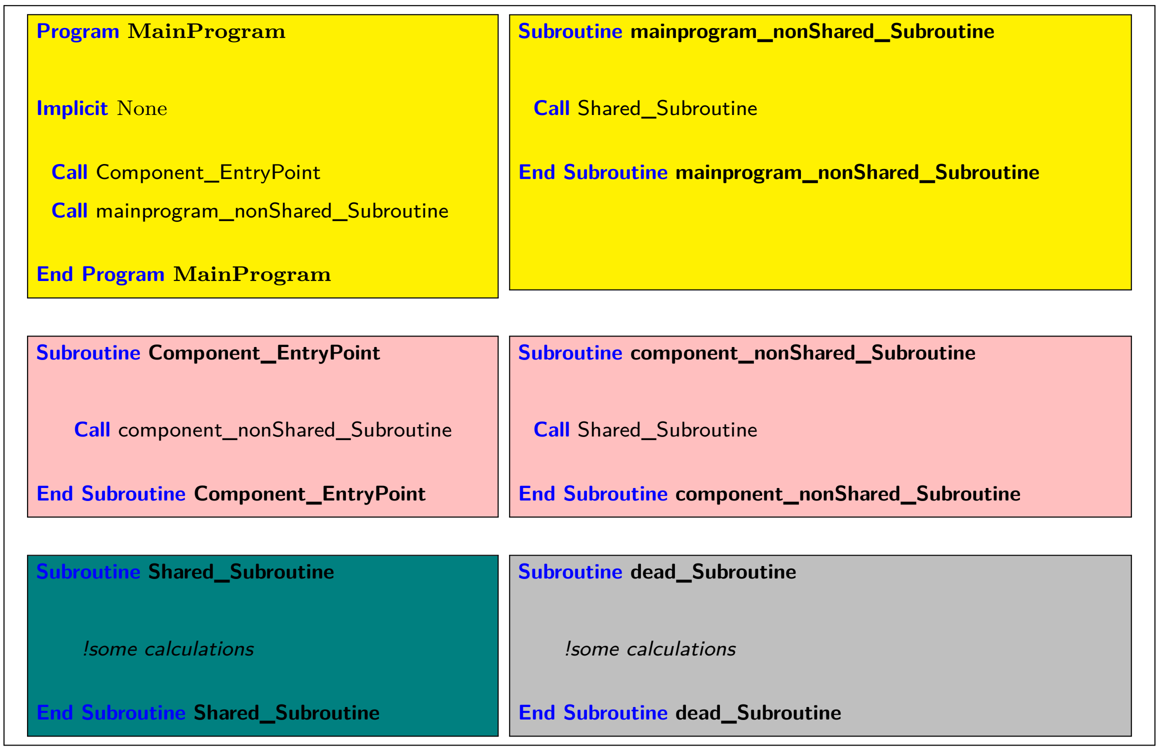
The global namespace of a Fortran program is always considered a shared namespace.

## Examples:

We first present some examples on some namespaces that are Fortran subprograms. Figure 6.11 shows two Fortran subroutines "*mainprogram_nonShared_Subroutine*" and "*component_nonShared_Subroutine*" which are non-shared namespaces. The subroutine "*dead_Subroutine*" is dead w.r.t. the component and its corresponding carvedout program. It is also dead w.r.t. to the program, which can be considered a bad practice in software engineering. The subroutine *"MainProgram" and "main-program_nonShared_Subroutine"* are also dead w.r.t. to the component, but not the carvedout program.

We now present some examples on some namespaces that are Fortran modules. Figure 6.12 shows a Fortran module that is a shared namespace though it is not clear which contents of the module are shared or non-shared. Note that using a Fortran module takes effect with a statement such as "*USE module_name*". In contrast, Figure 6.13 shows that the module *component_nonShared_module* is a non-shared namespace of the component, and the module *mainprogram_nonShared_module* will be a non-shared namespace of the corresponding carvedout program. Figure 6.14 shows that the contents of the shared namespace *Shared_Module* including a Fortran parameter, a derived data type, a variable and a subroutine are shared fully.

Figure 6.15 shows a shared namespace with shared and non-shared contents.
In contrast to some shared definitions, the subroutines *mainprogram_subroutine* and *component_subroutine* are not shared. Similarly, Figure 6.16 shows a shared namespace that contains no shared definitions. Finally, the module *deadcode_nonShared_module* in Figure 6.17 is considered a dead namespace w.r.t. the component, its corresponding carvedout program and the Fortran program. Whereas, *mainprogram_nonShared_Module* is a dead namespace only w.r.t. the component.



```
Program MainProgram

Implicit None

  Call Component_EntryPoint
  Call mainprogram_nonShared_Subroutine

End Program MainProgram
```

```
Subroutine mainprogram_nonShared_Subroutine

  Call Shared_Subroutine

End Subroutine mainprogram_nonShared_Subroutine
```

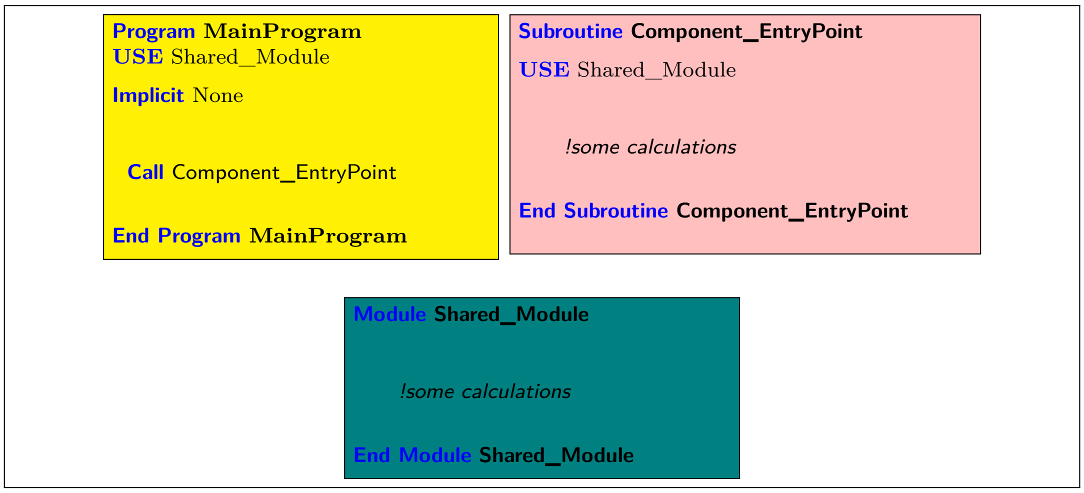
```
Subroutine Component_EntryPoint

    Call component_nonShared_Subroutine

End Subroutine Component_EntryPoint
```

```
Subroutine component_nonShared_Subroutine

  Call Shared_Subroutine

End Subroutine component_nonShared_Subroutine
```

```
Subroutine Shared_Subroutine

    !some calculations

End Subroutine Shared_Subroutine
```

```
Subroutine dead_Subroutine

    !some calculations

End Subroutine dead_Subroutine
```

**Figure 6.11.:** Examples of different subprogram namespaces. The subroutine “*mainprogram_nonShared_Subroutine*” (called only by the program) and ”*component_nonShared_Subroutine*” (called only by the component) are non-shared subprogram namespaces. The subroutine “*Shared_Subroutine*” is considered shared as it is reachable from the entry points of the component (*Component_EntryPoint*) and the entry point of the carvedout program (*MainProgram*). The subroutine *“component_EntryPoint*” is not considered a shared subprogram namespace as it is the entry point of the component. The subroutine *“dead_Subroutine”* is considered dead with respect to the program and the component. Subroutines *“MainProgram”* and *“mainprogram_nonShared_Subroutine”* are also dead with respect to the component.

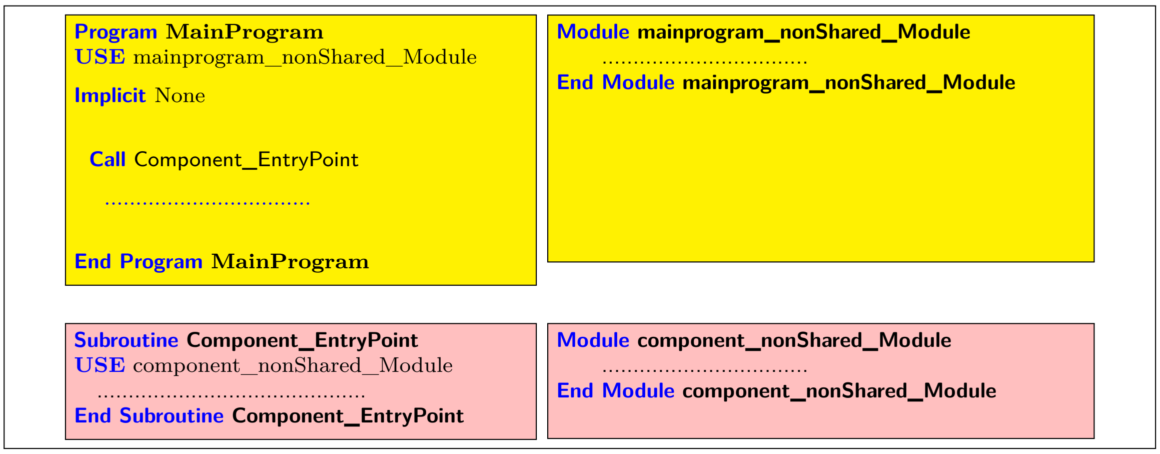


```
Program MainProgram
USE Shared_Module
Implicit None

  Call Component_EntryPoint

End Program MainProgram
```

```
Subroutine Component_EntryPoint
USE Shared_Module

    !some calculations

End Subroutine Component_EntryPoint
```

```
Module Shared_Module

    !some calculations

End Module Shared_Module
```

**Figure 6.12.:** An example of a module which is a shared namespace. "*Shared_Module*" is a shared namespace, but it is not clear which contents of the module are shared or non-shared.



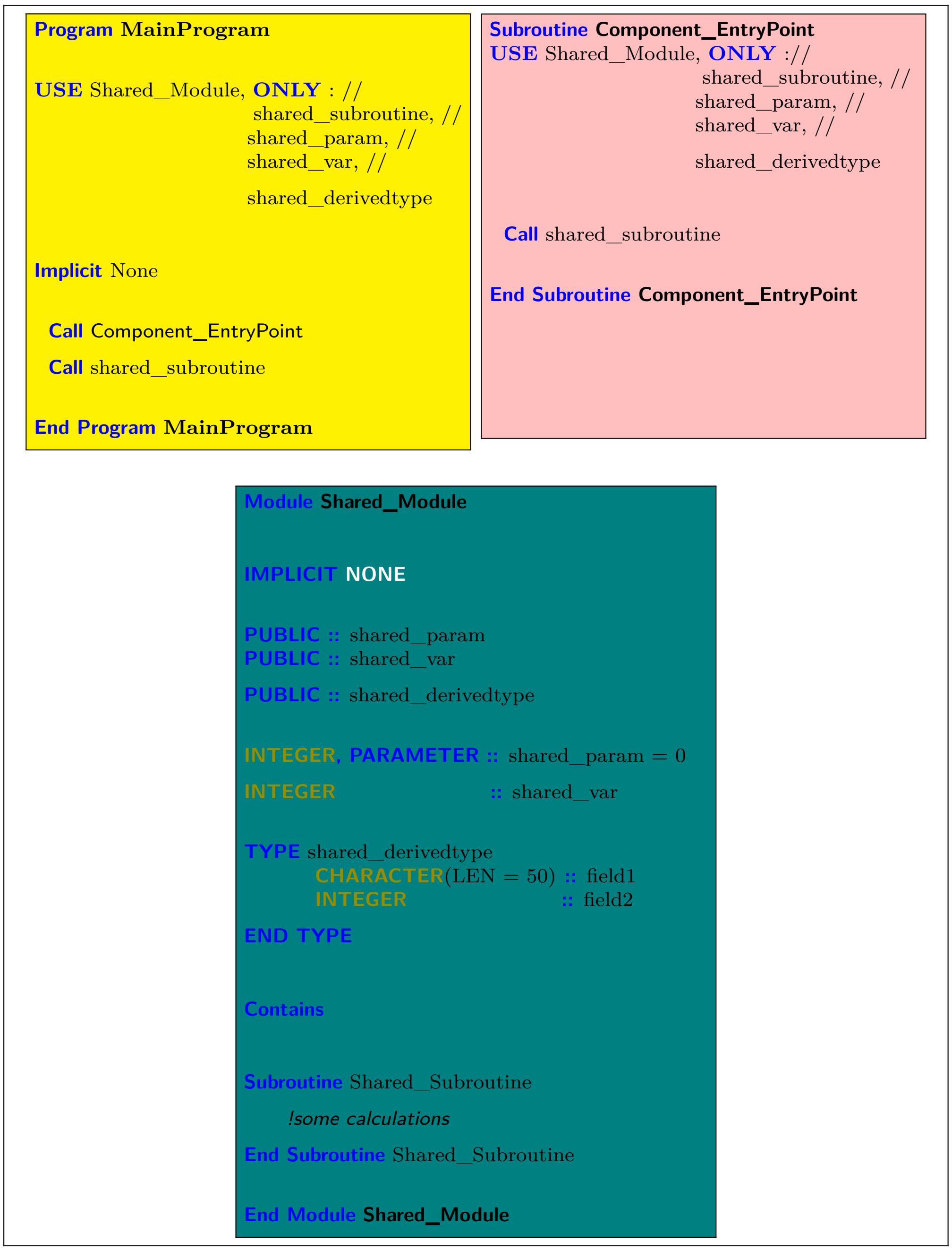
```
Program MainProgram
USE mainprogram_nonShared_Module
Implicit None

  Call Component_EntryPoint

    ................................

End Program MainProgram
```

```
Module mainprogram_nonShared_Module
    ................................
End Module mainprogram_nonShared_Module
```

```
Subroutine Component_EntryPoint
USE component_nonShared_Module
  ..........................................
End Subroutine Component_EntryPoint
```

```
Module component_nonShared_Module
    ................................
End Module component_nonShared_Module
```

**Figure 6.13.:** This is an example of modules that are non-shared namespace. The module "*component_nonShared_module*" is used only by the component and the module "*mainprogram_nonShared_module*" will be used only by the carvedout program.

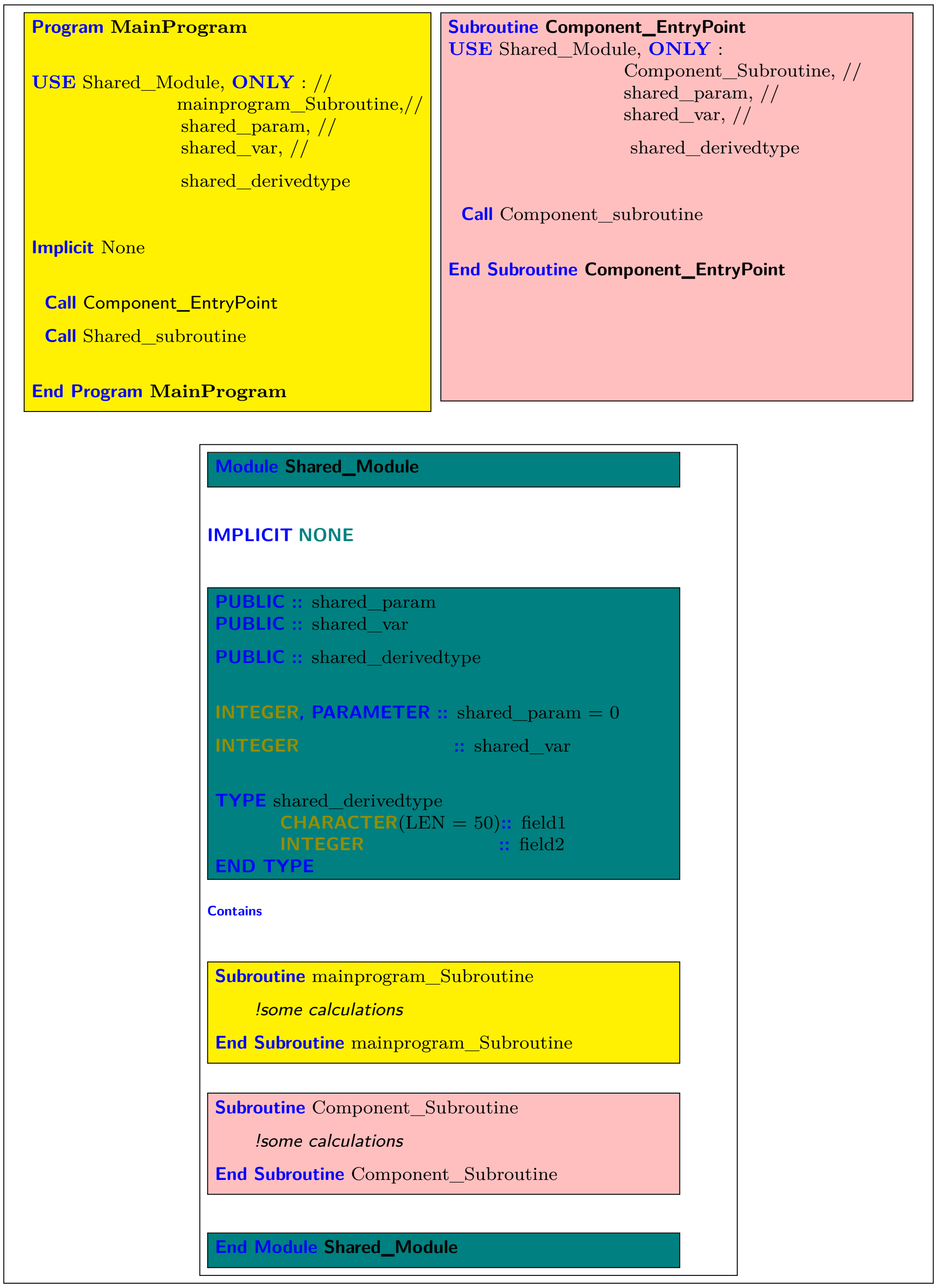


```
Program MainProgram

USE Shared_Module, ONLY : //
                          shared_subroutine, //
                          shared_param, //
                          shared_var, //

                          shared_derivedtype


Implicit None

  Call Component_EntryPoint

  Call shared_subroutine

End Program MainProgram
```

```
Subroutine Component_EntryPoint
USE Shared_Module, ONLY ://
                         shared_subroutine, //
                         shared_param, //
                         shared_var, //

                         shared_derivedtype


  Call shared_subroutine

End Subroutine Component_EntryPoint
```

```
Module Shared_Module


IMPLICIT NONE

PUBLIC :: shared_param
PUBLIC :: shared_var

PUBLIC :: shared_derivedtype

INTEGER, PARAMETER :: shared_param = 0

INTEGER            :: shared_var

TYPE shared_derivedtype
        CHARACTER(LEN = 50) :: field1
        INTEGER             :: field2
END TYPE


Contains


Subroutine Shared_Subroutine

      !some calculations

End Subroutine Shared_Subroutine

End Module Shared_Module
```

**Figure 6.14.:** This is an example of a shared module namespace with shared contents. The module "*Shared_Module*" and its internal definitions are fully shared.

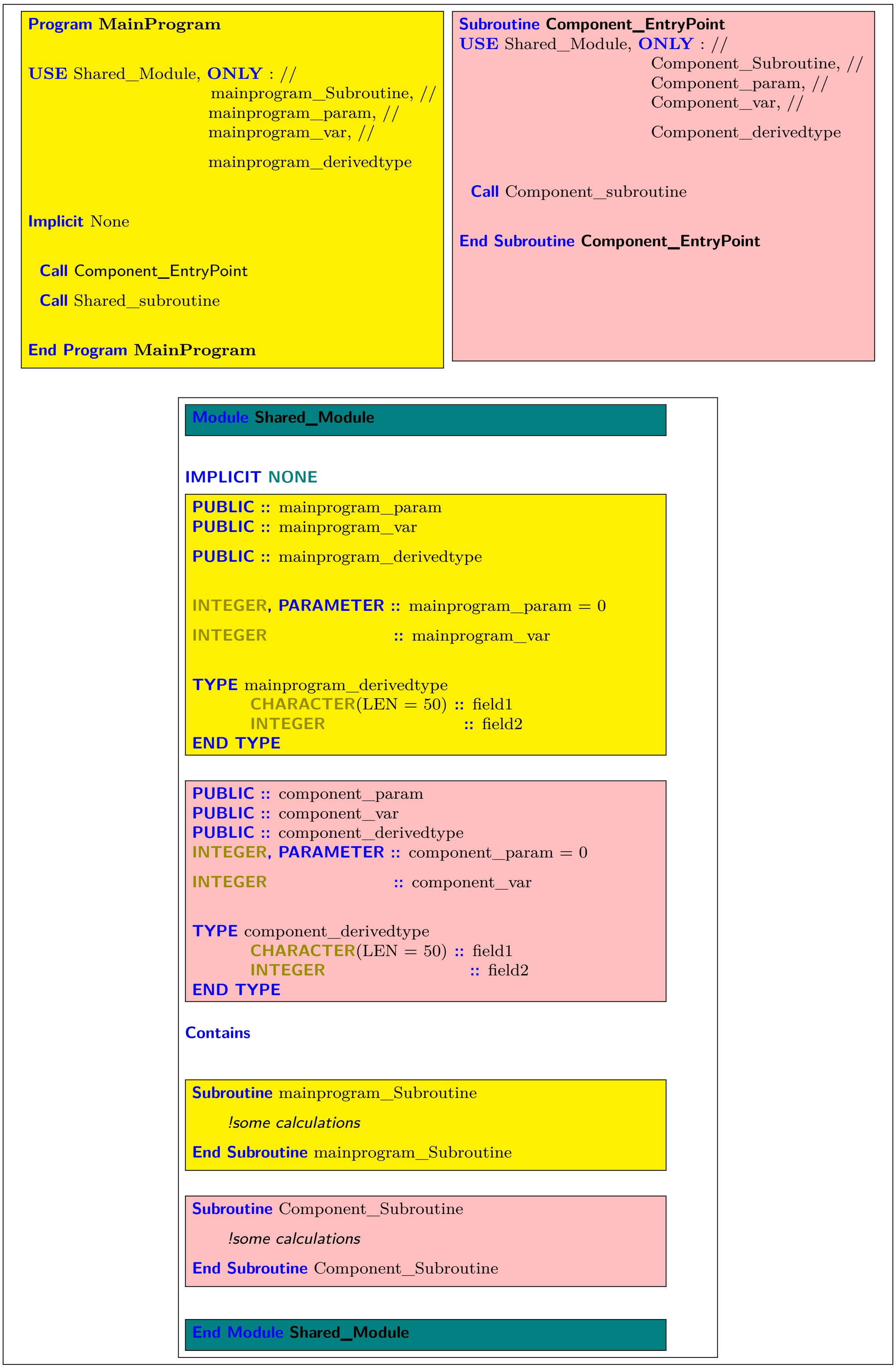


```
Program MainProgram

USE Shared_Module, ONLY : //
                    mainprogram_Subroutine,//
                    shared_param, //
                    shared_var, //

                    shared_derivedtype


Implicit None

  Call Component_EntryPoint

  Call Shared_subroutine

End Program MainProgram
```

```
Subroutine Component_EntryPoint
USE Shared_Module, ONLY :
                    Component_Subroutine, //
                    shared_param, //
                    shared_var, //

                    shared_derivedtype


  Call Component_subroutine

End Subroutine Component_EntryPoint
```

```
Module Shared_Module


IMPLICIT NONE


  PUBLIC :: shared_param
  PUBLIC :: shared_var

  PUBLIC :: shared_derivedtype


  INTEGER, PARAMETER :: shared_param = 0

  INTEGER            :: shared_var


  TYPE shared_derivedtype
        CHARACTER(LEN = 50):: field1
        INTEGER            :: field2
  END TYPE

Contains


  Subroutine mainprogram_Subroutine

        !some calculations

  End Subroutine mainprogram_Subroutine


  Subroutine Component_Subroutine

        !some calculations

  End Subroutine Component_Subroutine


  End Module Shared_Module
```

**Figure 6.15.:** This is an example of a shared namespace with shared and non-shared contents. In "*Shared_Module*", the Fortran parameter, variable and derived data type are shared, but "*component_subroutine*" is used only by the component, but "*mainprogram_subroutine*" will be used only by the corresponding carvedout program.

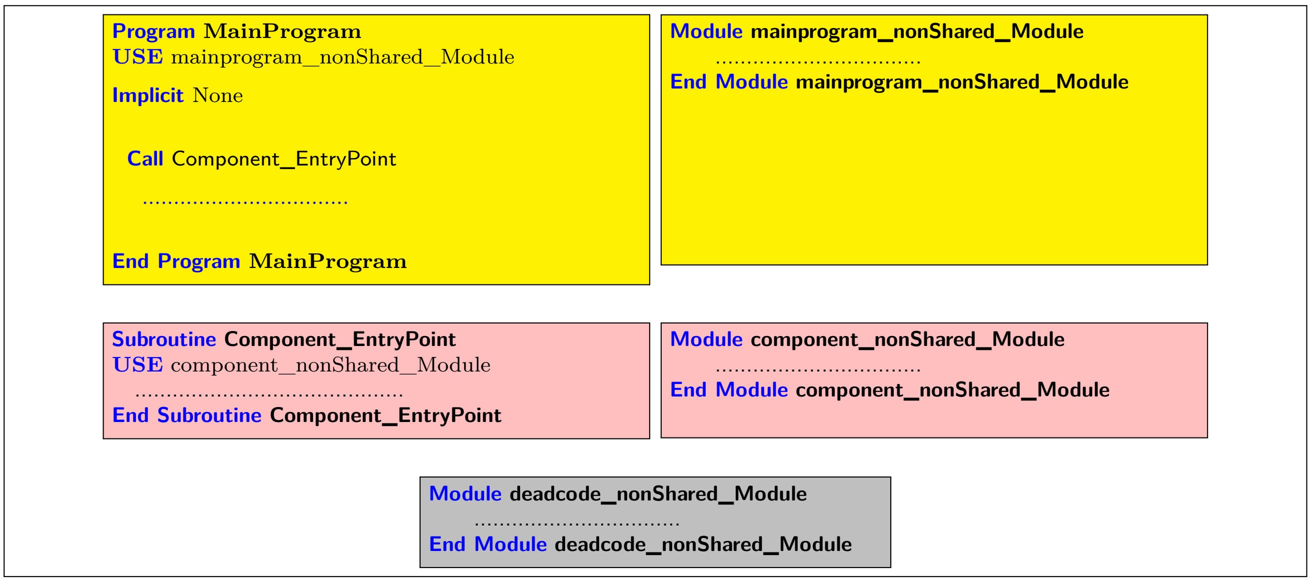


```
Program MainProgram

USE Shared_Module, ONLY : //
                           mainprogram_Subroutine, //
                           mainprogram_param, //
                           mainprogram_var, //
                           mainprogram_derivedtype

Implicit None

  Call Component_EntryPoint
  Call Shared_subroutine

End Program MainProgram
```

```
Subroutine Component_EntryPoint
USE Shared_Module, ONLY : //
                           Component_Subroutine, //
                           Component_param, //
                           Component_var, //
                           Component_derivedtype

  Call Component_subroutine

End Subroutine Component_EntryPoint
```

```
Module Shared_Module

IMPLICIT NONE
 PUBLIC :: mainprogram_param
 PUBLIC :: mainprogram_var
 PUBLIC :: mainprogram_derivedtype

 INTEGER, PARAMETER :: mainprogram_param = 0
 INTEGER            :: mainprogram_var

 TYPE mainprogram_derivedtype
         CHARACTER(LEN = 50) :: field1
         INTEGER             :: field2
 END TYPE

 PUBLIC :: component_param
 PUBLIC :: component_var
 PUBLIC :: component_derivedtype
 INTEGER, PARAMETER :: component_param = 0
 INTEGER            :: component_var

 TYPE component_derivedtype
         CHARACTER(LEN = 50) :: field1
         INTEGER             :: field2
 END TYPE

Contains

 Subroutine mainprogram_Subroutine
     !some calculations
 End Subroutine mainprogram_Subroutine

 Subroutine Component_Subroutine
     !some calculations
 End Subroutine Component_Subroutine

 End Module Shared_Module
```

**Figure 6.16.:** This is an example of a shared module namespace with non-shared contents. In "*Shared_Module*", no Fortran parameters, variables, derived data types and subroutines are shared.



```
Program MainProgram
USE mainprogram__nonShared__Module
Implicit None

 Call Component_EntryPoint
  ................................

End Program MainProgram
```

```
Module mainprogram_nonShared_Module
      ................................
End Module mainprogram_nonShared_Module
```

```
Subroutine Component_EntryPoint
USE component__nonShared__Module
 ..........................................
End Subroutine Component_EntryPoint
```

```
Module component_nonShared_Module
      ................................
End Module component_nonShared_Module
```

```
Module deadcode_nonShared_Module
      ................................
End Module deadcode_nonShared_Module
```

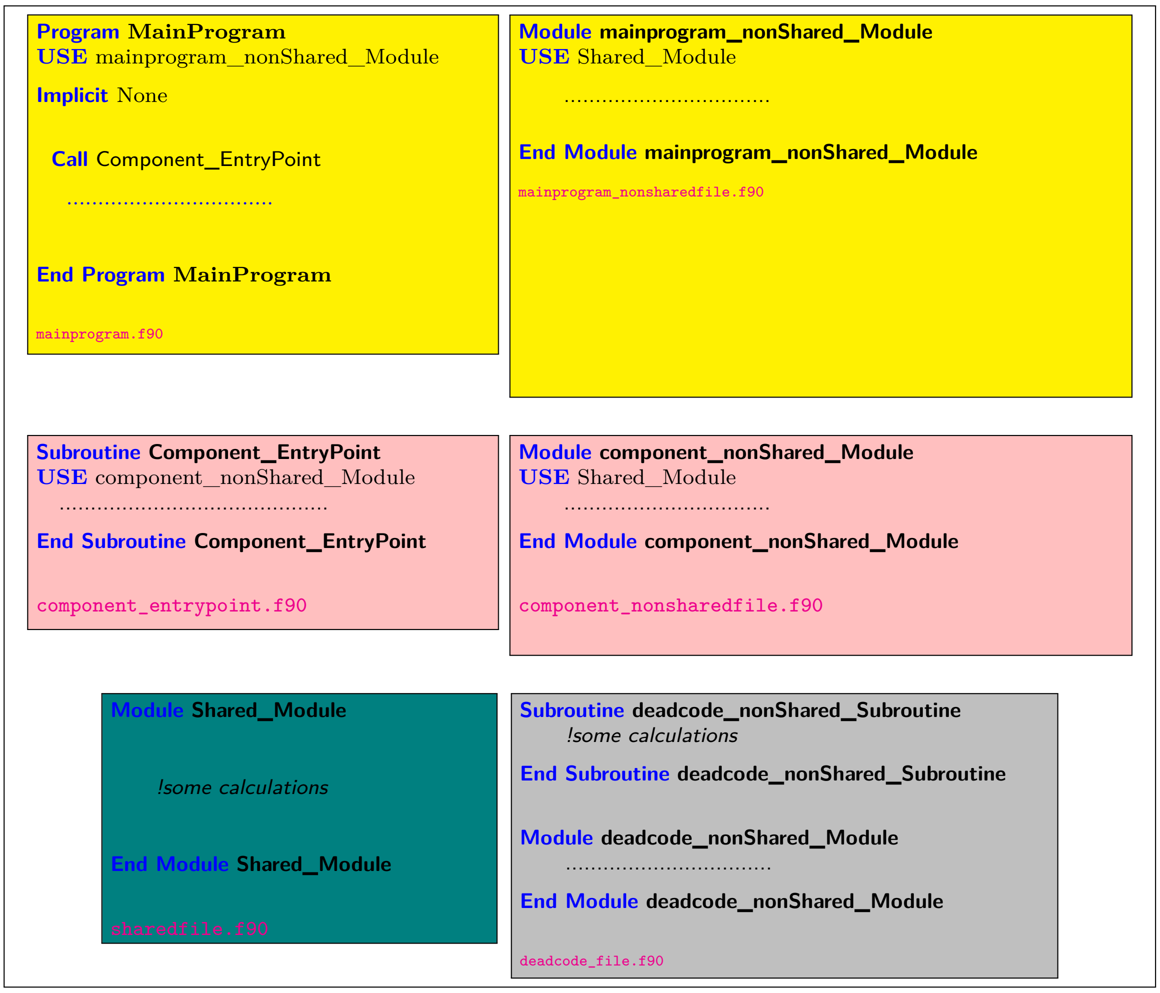
**Figure 6.17.:** This is an example of modules that are dead namespace. "*deadcode_nonShared_module*" is a dead namespace w.r.t. the component ans its the corresponding carvedout program as well as the Fortran program. However, "*mainprogram_nonShared_Module*" is a dead namespace w.r.t. the component, but not to the Fortran program.

#### 6.3.3.3. File sharing

The other factor affecting the complexity of a component is how the source code of a component is spread across Fortran files in a Fortran program. It is noteworthy that, in the previous examples, the name of the Fortran files containing the components were deliberately ignored in order to focus on other aspects. One component may, however, share several Fortran files with the carvedout program. A shared file is defined as below:

**Definition.** Let $P$ be a Fortran program containing the component $C$ and let $P_{co}$ be the carvedout program of $P$. ***A shared Fortran file*** of $P$ denotes a Fortran file of $P$ that contains the source code of $C$ and $P_{co}$ partially or completely. A shared file may contain shared, non-shared or dead namespaces. Otherwise, it is denoted as a ***non-shared Fortran file***.

Figure 6.18 demonstrates a Fortran program with a shared and three non-shared files. The shared file "*sharedfile.f90*" contains one shared namespace and the non-shared files "*mainprogram_nonsharedfile.f90*" and "*component_nonsharedfile.f90*" contain non-shared module namespaces. However, the non-shared file "*deadcode.f90*" contains a dead module namespace (*deadcode_nonShared_Module*) as well as a dead subprogram namespace (*deadcode_nonShared_Subroutine*).



```
Program MainProgram
USE mainprogram_nonShared_Module
Implicit None

  Call Component_EntryPoint
    ................................

End Program MainProgram

mainprogram.f90
```

```
Module mainprogram_nonShared_Module
USE Shared_Module
      ................................

End Module mainprogram_nonShared_Module

mainprogram_nonsharedfile.f90
```

```
Subroutine Component_EntryPoint
USE component_nonShared_Module
   .........................................
End Subroutine Component_EntryPoint

component_entrypoint.f90
```

```
Module component_nonShared_Module
USE Shared_Module
      ................................
End Module component_nonShared_Module

component_nonsharedfile.f90
```

```
Module Shared_Module

      !some calculations

End Module Shared_Module

sharedfile.f90
```

```
Subroutine deadcode_nonShared_Subroutine
      !some calculations
End Subroutine deadcode_nonShared_Subroutine

Module deadcode_nonShared_Module
      ................................
End Module deadcode_nonShared_Module

deadcode_file.f90
```

**Figure 6.18.:** This is an example of shared and non-shared Fortran files in a Fortran program. The Fortran file "*sharedfile.f90*" is a shared file that contains a shared module namespace while the non-shared files "*mainprogram_nonsharedfile.f90*" and "*component_nonsharedfile.f90*" contain non-shared module namespaces. In addition, the non-shared file *"deadcode_file.f90"* contains a dead module namespace (*deadcode_nonShared_Module*) and a dead subprogram namespace (*deadcode_nonShared_Subroutine*).

### 6.3.4. The depth of a component

The depth of a component is equal to the total number of Fortran modules and Fortran subprograms that it contains in the global namespace of a Fortran program. A component with a deep depth invokes numerous Fortran subprograms (that are located in the global namespace of the Fortran program containing the component) or its source code is spread in numerous Fortran modules. The components in Figure 6.2 and Figure 6.3 have a depth of 1 and the components in Figure 6.4, Figure 6.5, Figure 6.6, Figure 6.7, Figure 6.8 and Figure 6.9 have a depth of 2.

### 6.3.5. Epilogue

In this section, we discussed the complexity of a component in a Fortran program and provided some examples for a more clarification. It was shown that the complexity of a component can increase in four dimensions, namely by the types and the number of namespaces that it contains, the number of entry points, and shared contents between the component and other parts of the program. These variants of a component appear in Fortran programs, and, thus, any component extraction solution has to consider such a variety and be able to handle them in practice. Hence, the discussions in this section paved the way for understanding the intricacies of the approach proposed in the next section for achieving the general goal of this dissertation. This approach offers a novel solution for detecting the source code of a component in a Fortran program and separating the shared source code of the component from the other parts of the program. The precise definitions provided in this section (and in this chapter in general) create accurate technical jargon for presenting an accurate description of the solution and removing any ambiguity and misinterpretations.

## 6.4. Definition of Component Extraction and Isolation

In this section, we define two program analysis procedures called *Component Extraction and Component Isolation*, which deal with exploring the entire source of a component and adapting the whole Fortran program for the general goal of this dissertation.

### 6.4.1. Definition of Component Extraction

According to Section 6.1, the entry points of a component are assumed to be known, but the complete source code of a component may not be immediately visible without a thorough program analysis. Thus, the process of Component Extraction in this chapter can be defined as below:

**Definition:** Let $P$ be a Fortran program containing the component $C$ and let $s$ be a dead program statement in $P$ w.r.t. $C$. ***The extracted component*** $C_{ex}$ denotes a slice of $P$ in which every namespace $N$ is reachable in $G$ from at least one entry point of $C$ and every $s$ in $N$ is removed. Hence, ***Component Extraction*** denotes the process of generating $C_{ex}$.

An abstract model of Component Extraction is shown in Figure 6.19. The input data to this process includes the source code of the original Fortran program as well as the information concerning the entry points of the target component and the output is the extracted component.

#### No code dependency

The definition above implies that the extracted component has no source code dependency on the original program and it can, thus, compile independently from the original program successfully. Hence, the extracted component can be re-used in another Fortran program independently. On this account, as shown in Figure 6.20, a compilation test after the process of Component Extraction must always be rendered successful, otherwise the process is considered a failure.

#### Extracted component vs a component

Here, before delving into more details, the difference between a component and an extracted component should be emphasized. As defined in Section 6.1, a component is an *abstract subset* of the program statements in a Fortran program that are only reached from the entry points of the component. Any reference to a component of a Fortran program denotes this abstract model that is pointed to by the entry points of the component, but it does not concretely show the whole source code of the component. On the contrary, an extracted component clearly indicates a concrete subset of the statements of the program that contains only the complete source code of the component.

### 6.4.2. Definition of Component Isolation

Component Isolation is, however, slightly different and can be defined as below:

**Definition:** Let $P$ be a Fortran program containing the component $C$ and let $P_{co}$ be the carvedout program of $P$. Also, let $C_{ex}$ be the extracted component. Furthermore, let $N$ be a shared namespace of $C$ and let $N'$ be an exact copy of $N$ with a different name. ***The isolated component*** $C_{iso}$ denotes a modified copy of $C_{ex}$ in which every $N$ is replaced by $N'$ to create unique namespaces (across $C_{iso}$ and $P_{co}$) and every reference to $N$ is redirected to $N'$ in $C_{iso}$. On this account, ***Component Isolation*** denotes the process of generating $C_{iso}$ and $P_{co}$.

Figure 6.19 shows an abstract model of Component Isolation. The input data to this process includes the source code of the original Fortran program as well as the information concerning the entry points of the target component and its output is the isolated component and carvedout program.

### Unique namespaces

An isolated component and the carvedout program must allow for the re-integration within a new Fortran program. Hence, the union set of the namespaces of the isolated component and the carvedout program is not allowed to have two members with the same name. This feature makes it possible to compile the new Fortran program successfully. Hence, the namespaces with identical names generated during the process of Component Isolation are uniquely renamed.

### No code dependency

Additionally, the isolated component and the carvedout program must have no source code dependency on each other so that each slice can compile successfully independent from the other and the original program. This implies that we always assume that a Fortran program compiles successfully before applying the practice of Component Isolation. As shown in Figure 6.20, a compilation test of each slice must always be successful, otherwise the process is considered a failure.

### Data dependency

It is noteworthy that there are usually data dependencies between a component and the other parts of a Fortran program. However, the namespaces of the isolated component become separated from the carvedout program during the process of Component Isolation. If these two slices are re-integrated, bit-wise identical results to the original program cannot be generated unless a correct memory consistency between both slices is implemented.

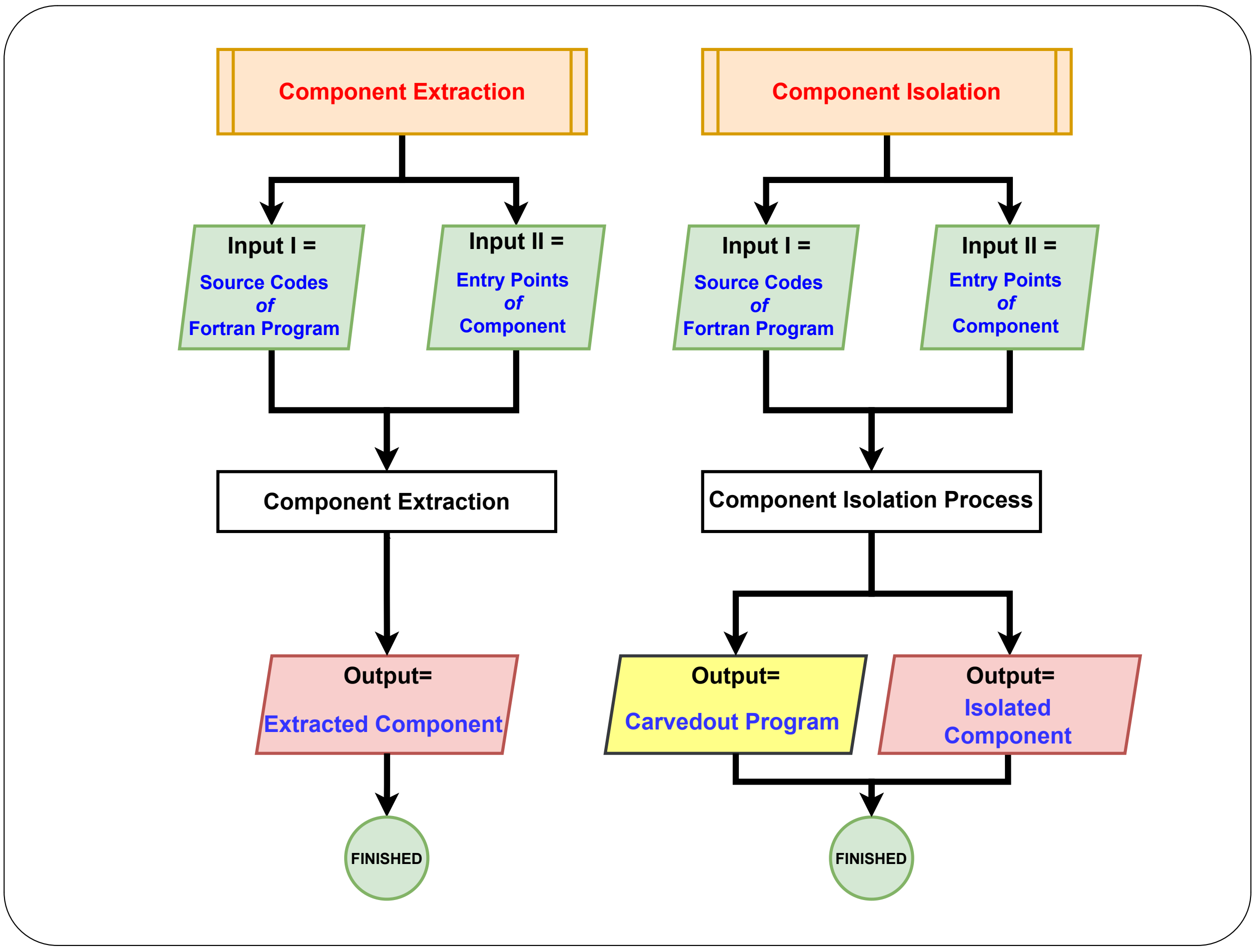


**Figure 6.19.:** An abstract model of the practice of Component Extraction and Component Isolation.

### Separate sets

In Component Isolation, two separate sets of source code will be generated. One set contains the source code of the carvedout program and the other set contains the source code of the isolated component. To improve the visibility of both sets, they should be contained into separate Fortran files.

### Component Isolation vs Component Extraction

Component Isolation and Extraction are very similar. However, Component Isolation aims at generating two slices from a Fortran program (i.e. the isolated component and the carvedout program) while Component Extraction generates only one slice (i.e. the extracted component). Both techniques, nevertheless, generate an extracted component the same way, but, in Component Isolation, it is then converted to an isolated component to guarantee unique namespaces across the two slices.

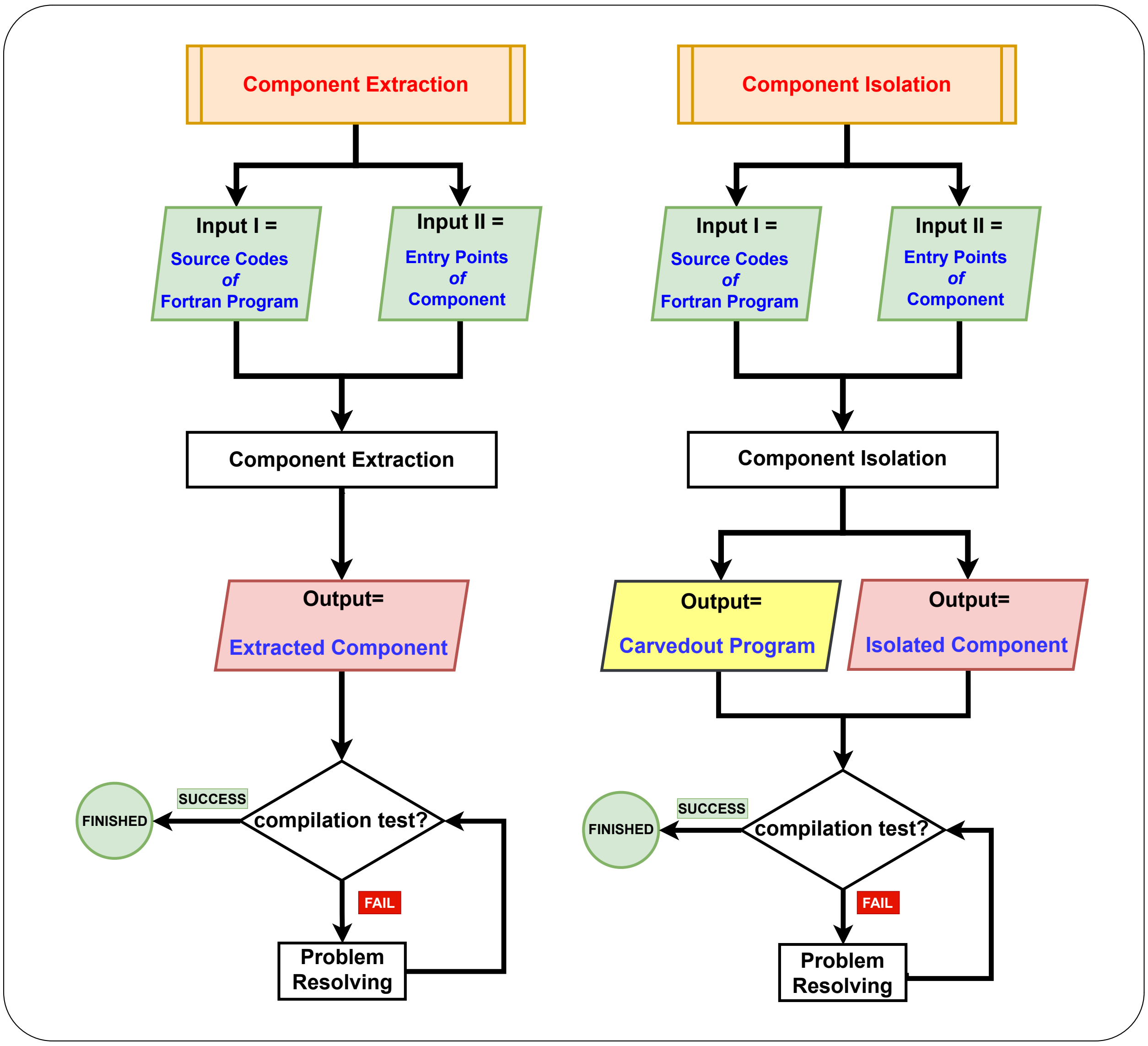

Component Extraction
Input I = Source Codes of Fortran Program
Input II = Entry Points of Component
Component Extraction
Output= Extracted Component
compilation test?
SUCCESS
FINISHED
FAIL
Problem Resolving
Component Isolation
Input I = Source Codes of Fortran Program
Input II = Entry Points of Component
Component Isolation
Output= Carvedout Program
Output= Isolated Component
compilation test?
SUCCESS
FINISHED
FAIL
Problem Resolving

**Figure 6.20.:** A compilation test is added to the abstract models proposed in Figure 6.19 to implement a validation step in the process of Component Extraction and Component Isolation.

## 6.5. Implementation of Component Isolation

This section presents the procedural steps to implement Component Isolation. We try to develop this approach step by step to make it easier for the reader to follow. It also seems to be informative to compare the implementation of Component Isolation to Component Extraction and, thus, we juxtapose both implementations wherever it is helpful in this section.

### 6.5.1. Implementation I: A simple model

A simple model of Component Extraction includes the following steps:

- Step 1: Finding the source code of the component
- Step 2: Adding the source code to the extracted component

Component Isolation is performed in three steps, which are as follows:

- Step 1: Finding the source code of the component
- Step 2: Adding the source code to the isolated component
- Step 3: Removing the source code of the component from the program to generate the carvedout program

So, Component Isolation is exactly similar to Component Extraction in Step 1 in order to find the source code of the component. In Step 2, Component Extraction and isolation generate different slices. Component Extraction builds the extracted component and Component Isolation creates the isolated component. Component Isolation has one more extra step (Step 3) in which it generates the carved out program. Figure 6.21 shows these two simple models. In these models, it is assumed that the target component does not share any source code with the carvedout program.

#### 6.5.1.1. Step 1: Finding the source code of the component

The first step to extract or isolate a component from a Fortran program is the detection of its source code. Hence, a dependency analysis is required in this step to explore the source code of the component. This analysis can be performed at a fine-grained level to pick up program statements of the component one by one. Since the simple model assumes that the component and the other parts of the program do not share any namespace, it seems that a fine-grained dependence analysis deals with numerous irrelevant details - making it very time-consuming. Instead, we choose to collect the source code of the component at a coarser level by picking the namespaces of the component from the set of the total namespaces of the program. If any entry point of a component reaches the program statements of more

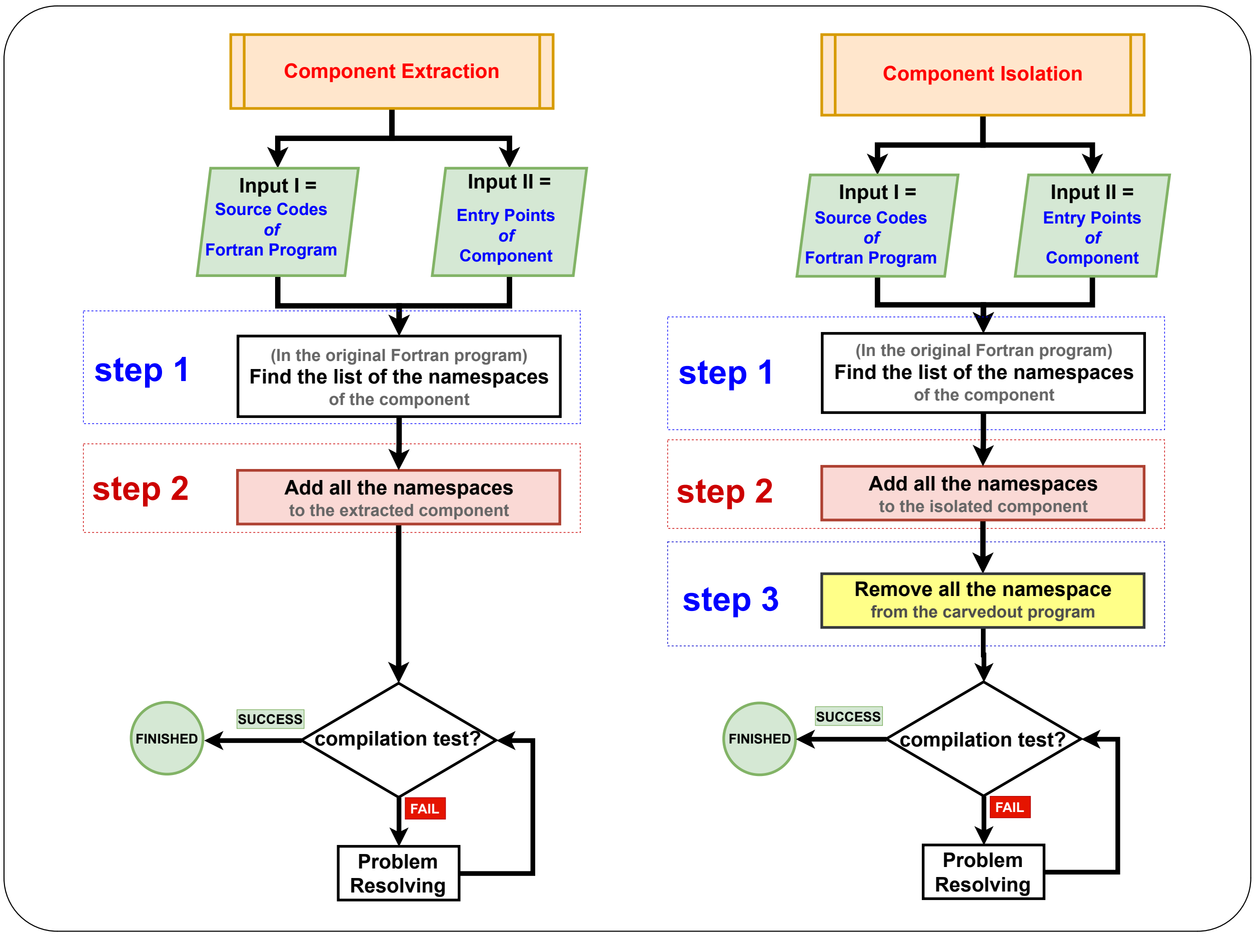

Component Extraction
Input I = Source Codes of Fortran Program
Input II = Entry Points of Component
step 1
(In the original Fortran program) Find the list of the namespaces of the component
step 2
Add all the namespaces to the extracted component
compilation test?
SUCCESS
FINISHED
FAIL
Problem Resolving
Component Isolation
Input I = Source Codes of Fortran Program
Input II = Entry Points of Component
step 1
(In the original Fortran program) Find the list of the namespaces of the component
step 2
Add all the namespaces to the isolated component
step 3
Remove all the namespace from the carvedout program
compilation test?
SUCCESS
FINISHED
FAIL
Problem Resolving

**Figure 6.21.:** A simple model of Component Extraction and Component Isolation

than one namespace, there must be a dependency between these namespaces. By extracting these dependencies, all the namespaces that a component depends on can be detected. Hence, a dependency analysis technique is required in this regard, as discussed below.

**Namespace dependence in Fortran:** This section discusses the nature of dependencies between namespaces of a Fortran program and shows how such features can be used in detecting the namespaces of a target component. A dependence between two namespaces A and B is defined as below:

**Definition.** Assuming there is a Fortran program with two namespaces A and B. If there is one definition in namespace B that the namespace A relies on, it is said that the **namespace A depends on namespace B.**

It was discussed earlier in Section 6.3.3.2 that there are three different types of namespaces in Fortran, namely the global namespace, module namespaces and subprogram namespaces. There is only one global namespace in any Fortran program, which contains all local namespaces such as subprogram and module namespaces. Thus, the dependency technique presented in this section only considers the dependencies between module and subprogram namespaces.

In Fortran, any namespace can potentially use public variables, derived data types, subprograms and etc. defined in another module namespace. Fortran reserves the keyword "*USE*" to show that one namespace requires some public definitions from another module. If the module namespace A depends on the module namespace B, it must contain a program statement such as "*USE B*" to declare a general dependence of A on B. Sometimes however, A has access only to a limited number of definitions in B. For example, if A uses only the variable "*varB*" defined in the namespace B, it can therefore declare a minimal dependency on the namespace B by stating "*USE B, ONLY : varB*".

By the same token, if a namespace depends on a subprogram namespace, it must contain a call to that subprogram. In addition, Fortran reserves the keyword "*EXTERNAL*" to show a potential namespace dependency in a Fortran file on a subprogram namespace defined in the global namespace in another file. If the namespace A defined in a Fortran file such as "*File1.f90*" contains a call to a subprogram B defined in the global namespace within another Fortran file such as "*File2.f90*", there is a dependency between A and B and Fortran expects a program statement such as "*EXTERNAL :: B*" in the the Fortran file containing the namespace of A, i.e. "*File1.f90*".

**Extracting namespaces dependencies in Fortran programs:** As discussed above, there might be dependencies between modules and subprograms namespaces in a Fortran program. To find namespace dependencies, the following types of search will be required:

- Finding a dependence on a module namespace:
  To find the dependency of the namespace A on the module namespace B, a search for "USE B" in the namespace A is required. Exploring all the"USE" statements in the namespace A should lead to the detection of all the dependencies of A on other Fortran modules.
- Finding a dependence on a subprogram namespace:
  If a (module or subprogram) namespace has dependencies on subprograms defined in a module namespace, the dependencies on the module (but not on the individual subprograms) are only considered. However, dependencies on subprograms defined in the global namespace are captured. To find a dependency of the (module or subprogram) namespace A on the subprogram namespace B (defined in the global namespace), a search for the program statement "*CALL B*" within the namespace A is required. This method, however, suggests that the search has to be performed for every subprogram defined in the global namespace. An optimal solution, however, limits the search for dependencies on all the subprograms defined in the global namespace within the same Fortran file in which the namespace A is defined as well as the subprograms indicated by "*EXTERNAL*" statements (within the same file).

#### 6.5.1.2. Step 2: Adding a namespace to the extracted and isolated component

Once the namespaces of a component are explored, it is, however, a matter of software organization where the source code of an extracted or isolated component should end up. Step 2 suggests the following tasks:

- placing the source code of each namespace of the component in a new Fortran file so that the extracted or isolated component can be re-used in other software applications.
- placing Fortran files of the extracted component or the isolated component in a separate folders from the original program to maximize their visibility.

#### 6.5.1.3. Step 3: Removing all the namespaces from the carvedout program

Once a namespace is added to the isolated component, it will be removed from the carvedout program. This is based on the assumption of the simple model presented in this section that a component does not share any namespace with the carvedout program. Thus, Step 3 is added to Component Isolation to perform this task. This step makes sure that the carvedout program does not retain any dependency on the component.

### 6.5.2. Implementation II: an incremental procedure

The algorithm proposed in Figure 6.21 still needs some improvements to make sure that it is an effective solution in practice. The main problem is raised by the compilation test, which takes place at the end of the process. If the compilation test fails however, it will be very difficult to trace back the problem leading to the failure. For this reason, Figure 6.22 suggests adding one more step to perform a compilation test in each iteration and apply the practice of Component (Extraction or) Isolation incrementally. In this approach, the namespace of a component is added one by one to the extracted or isolated component. If the compilation test succeeds, the process goes to the next iteration. If the test fails, the troubleshooting is performed immediately (not at the end of the practice) to localize the problem. Such a test usually requires the re-configuration of the build system ( in our case the GNU Build System (GNU Webpage, last access: 25 Mar 2022) for compiling the modified program. This additional step is still worthy of the overhead that it imposes on the practice of Component Extraction and Component Isolation as it minimizes the search space for the required trouble shooting.

Note that an extracted or isolated component and a carvedout program are by definition the slices of a Fortran program generated at the end of the practice of Component Extraction and Isolation. However, these terminologies are still exploited to refer to the output of the incremental procedure in each iteration. Consequently, some dependencies may be lacking or redundantly exist in these slices within the intermediary iterations.

### 6.5.3. Implementation III: handling shared namespaces

So far , it was assumed that components in a Fortran program contain only non-shared namespaces. However, in real-world software applications, components contain both non-shared and shared namespaces. As a result, Component Extraction and Isolation should be adapted as explained below:

1. If there is a shared namespace, both the isolated component and carvedout program depend on this namespace and require individual copies of the namespace. Hence, the following modifications are necessary in the practice of Component Isolation (but not in Component Extraction):
    a) Shared namespaces are not removed from the carvedout program and thus Step 4 is modified accordingly.
    b) In addition, shared namespaces are renamed before being added to the isolated component.

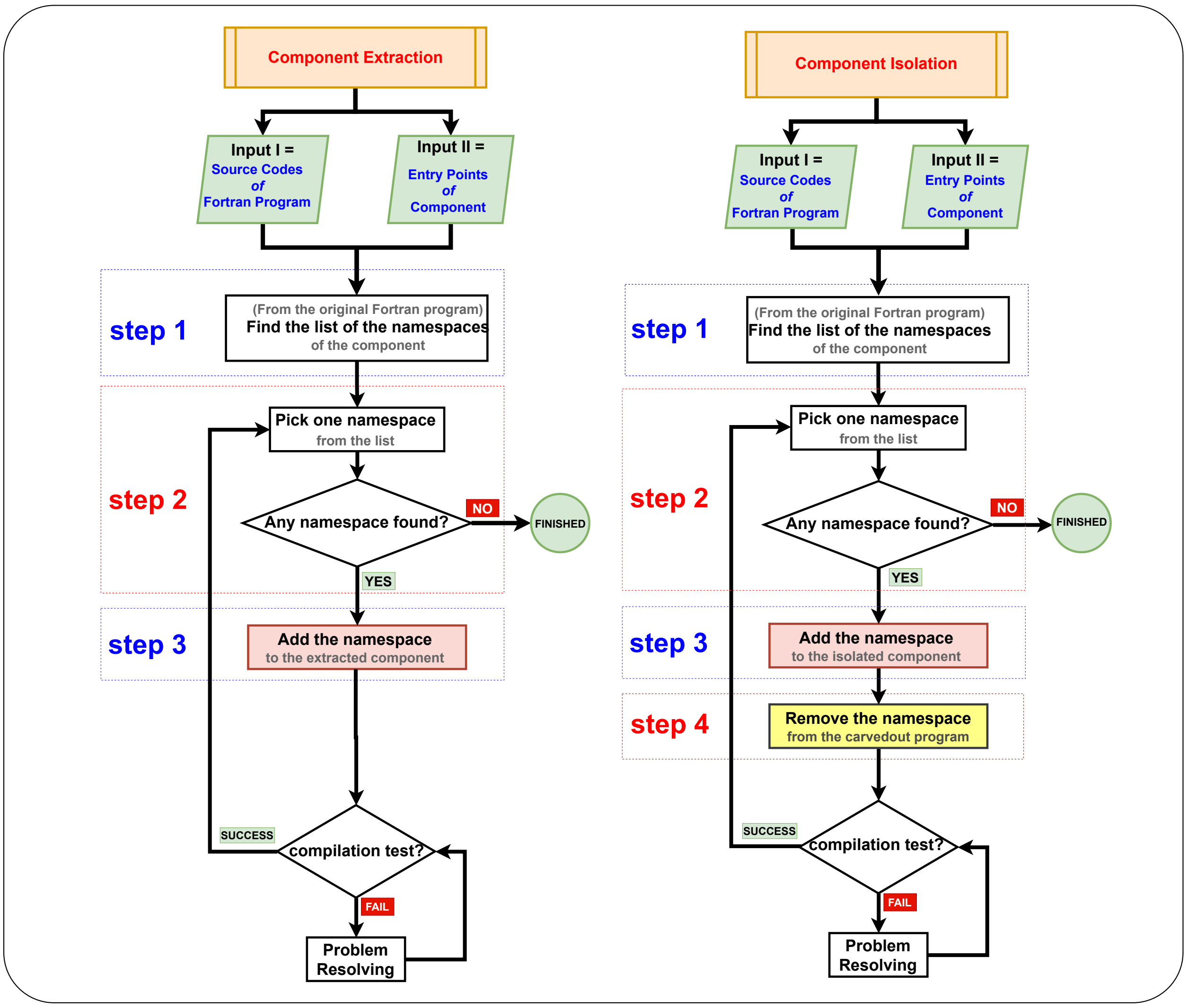

Component Extraction
Input I = Source Codes of Fortran Program
Input II = Entry Points of Component
step 1
(From the original Fortran program) Find the list of the namespaces of the component
step 2
Pick one namespace from the list
Any namespace found?
NO
FINISHED
YES
step 3
Add the namespace to the extracted component
SUCCESS
compilation test?
FAIL
Problem Resolving
Component Isolation
Input I = Source Codes of Fortran Program
Input II = Entry Points of Component
step 1
(From the original Fortran program) Find the list of the namespaces of the component
step 2
Pick one namespace from the list
Any namespace found?
NO
FINISHED
YES
step 3
Add the namespace to the isolated component
step 4
Remove the namespace from the carvedout program
SUCCESS
compilation test?
FAIL
Problem Resolving

**Figure 6.22.:** Extracting and isolating a component incrementally

c) Finally, any reference to a shared namespace in the isolated component must be modified accordingly to refer to the re-named copy (in the isolated component) rather than the original (in the carvedout program). For example, if the subprogram namespace *"subroutineFOO" is renamed to "comp_subroutineFOO"* when added to the isolated component, any dependency on this subprogram (in the isolated component) such as "*CALL subroutineFOO" or* "*EXTERNAL :: subroutineFOO"* will be converted to "*CALL comp_subroutineFOO"* or "*EXTERNAL :: comp_subroutineFOO"*. By the same token, if a module namespace "*moduleFOO*" is renamed to "*comp_moduleFOO"* (when added to the isolated component), any dependency on this namespace such as "*USE moduleFOO"* (in the isolated component) will be converted to "*USE comp_ModuleFOO"*.

2. As discussed in Section 6.3.3.2, a shared namespace may contain dead contents either w.r.t. the component or w.r.t. the carvedout program. In consequence, dead contents w.r.t. the component must be removed from the copy of the shared-namespace added to the extracted or isolated component and the dead contents w.r.t. to the carvedout program must be removed from the copy of the namespace in carvedout program. Thus, the procedure of Component Extraction and Isolation must be modified to include one step to handle the dead contents. This is a topic of a broader discussion which will be provided in the next section.

## 6.5.4. Implementation IV: removing dead contents

This section explores the impacts of dead contents on the practices of Component Extraction and Isolation and proposes an additional step in order to detect such contents and eliminate them from an extracted or isolated component. It was explained in Section 6.3 that dead contents appear either as dead program definitions or dead program instructions, which will be discussed here.

### 6.5.4.1. dead program definitions

Dead program definitions in a Fortran program are dead program statements defining variables, derived data types, namespaces and etc. A namespace usually contains a set of program definitions, but chances are only a subset of them is actually used either by a component or by the other parts of the containing Fortran program. Dead program definitions may create wrong dependencies on other namespaces. Figure 6.24 shows an example of such wrong dependencies. The entry point of the component *"component_entrypoint"* invokes two subroutines, namely *"cmp_sub_shared"* and *"cmp_sub_nonshared"*. As a result, the component has two

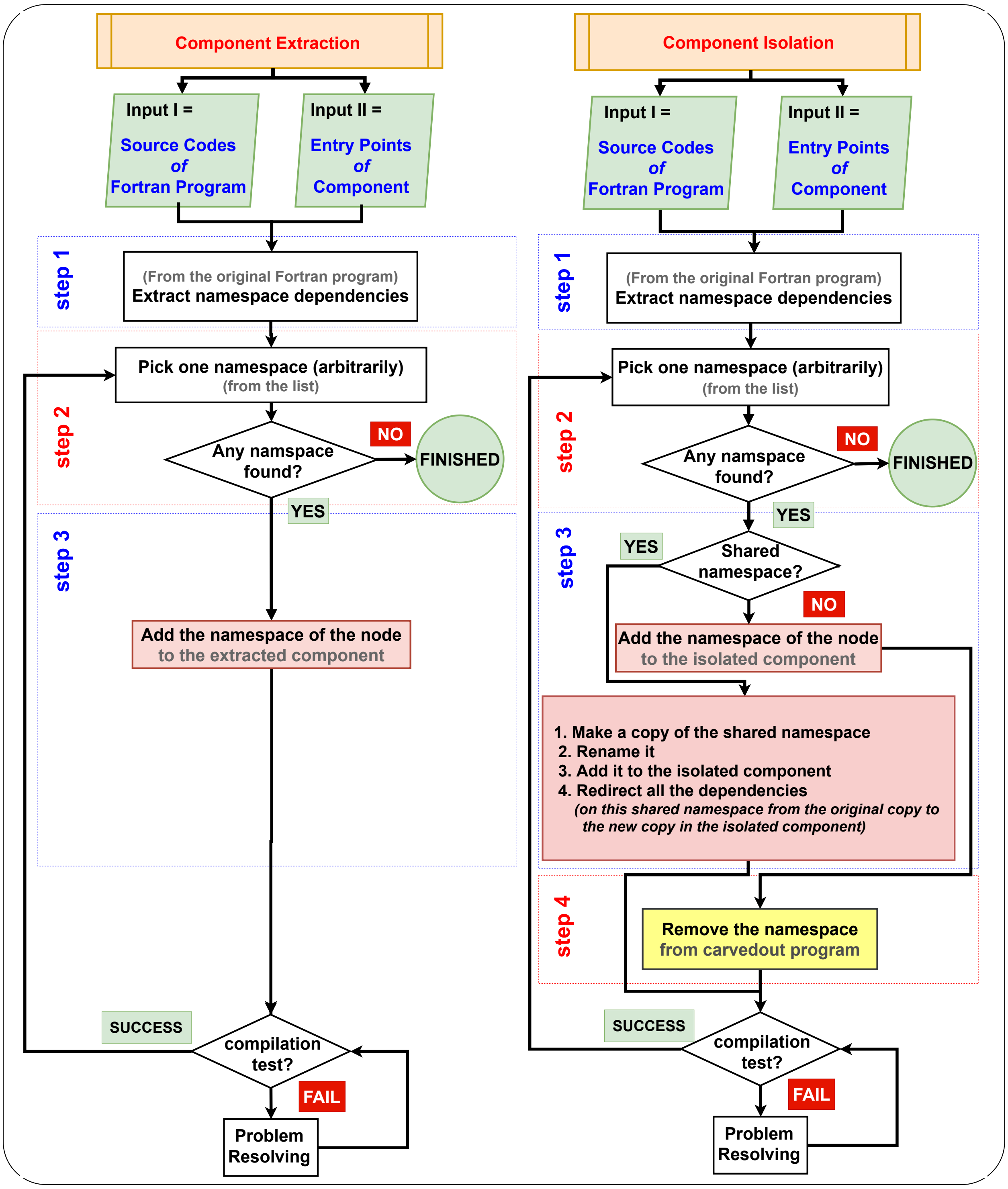

Component Extraction
Input I =
Source Codes of Fortran Program
Input II =
Entry Points of Component
step 1
(From the original Fortran program)
Extract namespace dependencies
step 2
Pick one namespace (arbitrarily)
(from the list)
Any namspace found?
NO
FINISHED
YES
step 3
Add the namespace of the node to the extracted component
SUCCESS
compilation test?
FAIL
Problem Resolving
Component Isolation
Input I =
Source Codes of Fortran Program
Input II =
Entry Points of Component
step 1
(From the original Fortran program)
Extract namespace dependencies
step 2
Pick one namespace (arbitrarily)
(from the list)
Any namspace found?
NO
FINISHED
YES
step 3
YES
Shared namespace?
NO
Add the namespace of the node to the isolated component
1. Make a copy of the shared namespace
2. Rename it
3. Add it to the isolated component
4. Redirect all the dependencies
(on this shared namespace from the original copy to the new copy in the isolated component)
step 4
Remove the namespace from carvedout program
SUCCESS
compilation test?
FAIL
Problem Resolving

**Figure 6.23.:** Handling shared namespaces

correct dependencies on *“shared_module”* and *“nonshared_module2”*. However, there are dead definitions in these two namespaces that lead to some wrong dependencies of the extracted component or isolated component on the dead namespaces w.r.t. the component. In particular, the subroutines *“prg_sub_shared”* and *“dead_sub”* in *“shared_module”* are not invoked by the component and they are, therefore, dead w.r.t. the component. So they create wrong dependencies on *“nonshared_module1”* and *“dead_module1”*. By the same token, the subroutine *“dead_sub”* in the namespace *“nonshared_module2”* is not invoked by the component, but it creates a wrong dependency on the namespace *“dead_module2”*.

#### 6.5.4.2. dead program instructions

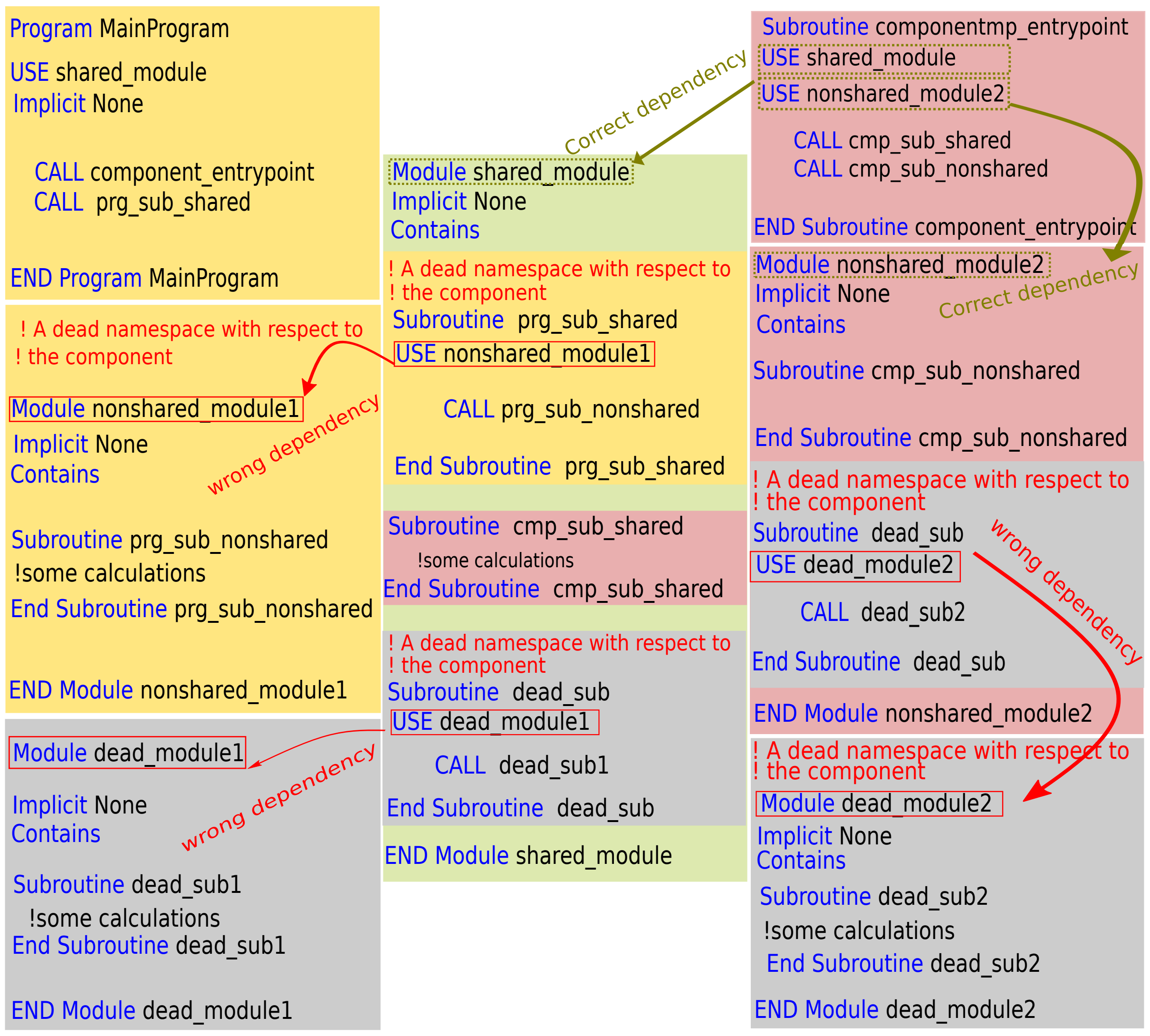
Dead program instructions w.r.t. to a component (of a containing Fortran program) or w.r.t. the carvedout program can be found in shared subprogram namespaces. If a subprogram is shared, it might potentially contain a path that is not taken when the subprogram is invoked either by the component or by the carvedout program. A subprogram that provides different services to a component and the carvedout program is a good example. Note that if a subprogram namespace is dead w.r.t. to a component, it means that the definition of the subprogram is not reached (in the control flow graph of the program) from the entry points of the component. However, if some program instructions of a subprogram namespace are dead w.r.t. the entry points of the component, it implies that there is a path in the subprogram that is never taken when the subprogram is invoked by the component.

Figure 6.25 shows a component in a Fortran program. It invokes a shared subroutine called *“shared_sub*” and sets the input parameter *“pathtype”* to 1 to run the subroutine *“component_sub*”. This choice creates a dependency on the module namespace *“nonshared_module1”*. The main program also invokes the subroutine *“shared_sub”*. However, it sets *“pathtype”* to 2, leading to the invocation of the subroutine *“mainprogram_sub*”. This subroutine is a dead namespace w.r.t. the component, but it creates a dependency on the module namespace *“nonshared_module2*” during the process of Component Extraction and Isolation. Thus, a dead code removal technique is needed to spot such dead paths and eliminate them.

#### 6.5.4.3. Negative consequences of the wrong dependencies

The wrong dependencies due to dead contents may lead to the following problems:

- a longer processing time to extract or isolate a component from a Fortran program.



```
Program MainProgram
USE shared_module
 Implicit None

   CALL component_entrypoint
   CALL  prg_sub_shared

END Program MainProgram
```

```
 ! A dead namespace with respect to
! the component

Module nonshared_module1
 Implicit None
Contains

Subroutine prg_sub_nonshared
 !some calculations
End Subroutine prg_sub_nonshared

END Module nonshared_module1
```

```
Module dead_module1

Implicit None
Contains

Subroutine dead_sub1
   !some calculations
End Subroutine dead_sub1

END Module dead_module1
```

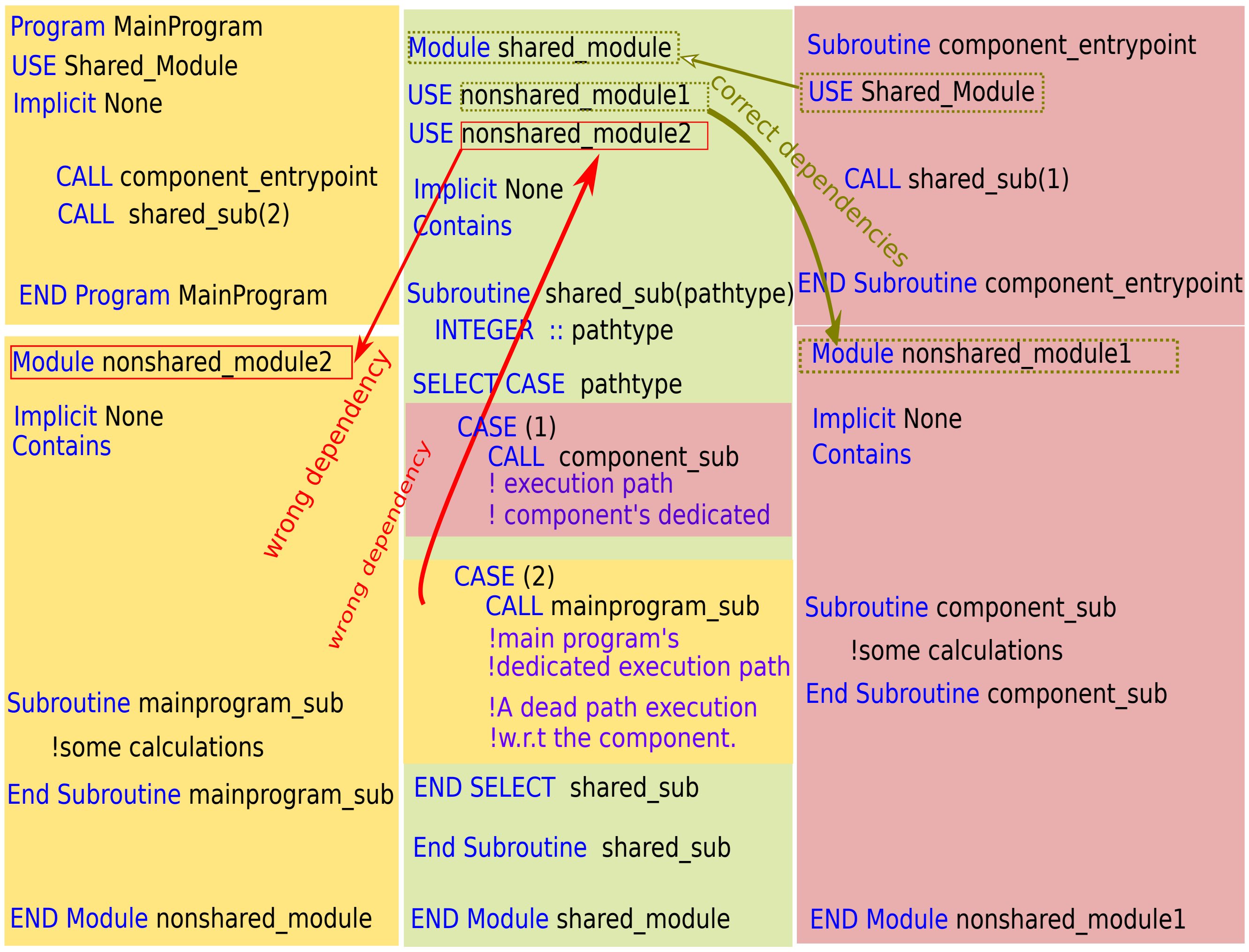
```
Module shared_module
Implicit None
Contains
! A dead namespace with respect to
! the component
 Subroutine  prg_sub_shared
 USE nonshared_module1

       CALL prg_sub_nonshared

 End Subroutine  prg_sub_shared

Subroutine  cmp_sub_shared
    !some calculations
End Subroutine  cmp_sub_shared

! A dead namespace with respect to
! the component
Subroutine  dead_sub
 USE dead_module1

       CALL  dead_sub1

End Subroutine  dead_sub

END Module shared_module
```

```
 Subroutine componentmp_entrypoint
 USE shared_module
 USE nonshared_module2

    CALL cmp_sub_shared
    CALL cmp_sub_nonshared

END Subroutine component_entrypoint
```

```
Module nonshared_module2
Implicit None
Contains

Subroutine cmp_sub_nonshared

End Subroutine cmp_sub_nonshared
! A dead namespace with respect to
! the component
Subroutine  dead_sub
USE dead_module2

      CALL  dead_sub2

End Subroutine  dead_sub

END Module nonshared_module2
```

```
! A dead namespace with respect to
! the component
 Module dead_module2
 Implicit None
 Contains
  Subroutine dead_sub2
  !some calculations
   End Subroutine dead_sub2

END Module dead_module2
```

**Figure 6.24.:** This is an example of dead program definitions that create wrong dependencies during Component Extraction and Isolation.



```
Program MainProgram
USE Shared_Module
Implicit None

      CALL component_entrypoint
      CALL  shared_sub(2)

 END Program MainProgram
```

```
Module nonshared_module2

Implicit None
Contains

Subroutine mainprogram_sub
      !some calculations
End Subroutine mainprogram_sub

END Module nonshared_module
```

```
Module shared_module
USE nonshared_module1
USE nonshared_module2

Implicit None
Contains

Subroutine shared_sub(pathtype)
   INTEGER  :: pathtype

SELECT CASE  pathtype
       CASE (1)
           CALL  component_sub
           ! execution path
           ! component's dedicated

       CASE (2)
           CALL mainprogram_sub
           !main program's
           !dedicated execution path
           !A dead path execution
           !w.r.t the component.

END SELECT  shared_sub

End Subroutine  shared_sub

END Module shared_module
```

```
Subroutine component_entrypoint
USE Shared_Module

      CALL shared_sub(1)

END Subroutine component_entrypoint
```

```
Module nonshared_module1

Implicit None
Contains

Subroutine component_sub
      !some calculations
End Subroutine component_sub

END Module nonshared_module1
```

**Figure 6.25.:** This is an example of dead program instructions.

- increasing the size of the extracted or isolated component, which might be harmful for some applications. For example, to evaluate the impact of lower precision arithmetic on scientific calculations, all the computations within a component must be modified. The presence of dead contents creates a extra burden on the practice.
- leading to wrong data dependencies in the data flow analysis between a component and the carvedout program.

#### 6.5.4.4. Handling dead contents

Now that the negative impacts of dead contents are clarified, a small adaptation is necessary. Figure 6.26 shows how Step 3 and 4 in Component Extraction and Isolation have been modified to handle the detection and removal of dead contents.

**A method for removing dead content from an isolated component:**
It was explained that dead contents in Fortran programs appear in two main forms and, thus, different removal measures should be taken as described below:

- Removing dead program instructions:
  In this dissertation, we assume that if a component has a dependency on a subprogram namespace, all the paths of the subprogram can be taken by the entry points of the component.
- Removing dead program definitions:
  Before adding a namespace to an extracted or isolated component, a static source code analysis is required to detect dead program definitions. Thus, a backtracking method can be employed to remove one program definition per time from a module namespace. This dissertation exploits the static code analysis of the target compiler to find any reference to a removed definition. If the compilation fails, the definition must be added back to the namespace. If the compilation succeeds, the definition will be removed forever. This process continues until all the definitions of the namespace are visited. An implementation of this technique has been proposed in Figure 6.27.

#### 6.5.4.5. A performance optimization: Reducing the burden in dead contents removal

Searching for dead content in every namespace leads to a longer processing time in Component Extraction and Isolation, thus an optimal solution as proposed below is opportune.

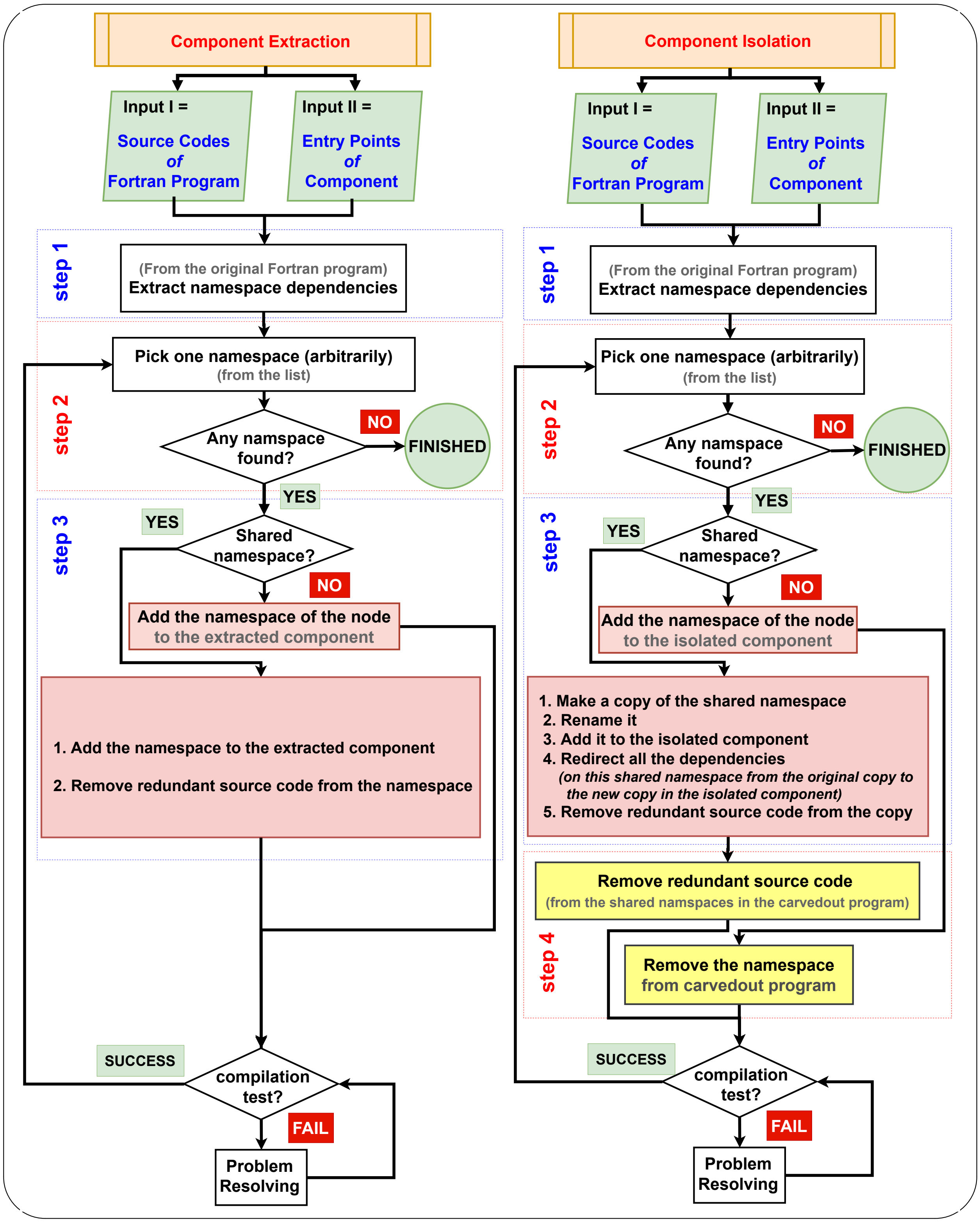

Component Extraction
Input I =
Source Codes of Fortran Program
Input II =
Entry Points of Component
step 1
(From the original Fortran program)
Extract namespace dependencies
step 2
Pick one namespace (arbitrarily)
(from the list)
Any namspace found?
NO
FINISHED
YES
step 3
Shared namespace?
YES
NO
Add the namespace of the node to the extracted component
1. Add the namespace to the extracted component
2. Remove redundant source code from the namespace
SUCCESS
compilation test?
FAIL
Problem Resolving
Component Isolation
Input I =
Source Codes of Fortran Program
Input II =
Entry Points of Component
step 1
(From the original Fortran program)
Extract namespace dependencies
step 2
Pick one namespace (arbitrarily)
(from the list)
Any namspace found?
NO
FINISHED
YES
step 3
Shared namespace?
YES
NO
Add the namespace of the node to the isolated component
1. Make a copy of the shared namespace
2. Rename it
3. Add it to the isolated component
4. Redirect all the dependencies
(on this shared namespace from the original copy to the new copy in the isolated component)
5. Remove redundant source code from the copy
step 4
Remove redundant source code
(from the shared namspaces in the carvedout program)
Remove the namespace from carvedout program
SUCCESS
compilation test?
FAIL
Problem Resolving

**Figure 6.26.:** Handling dead contents removal in the procedure of Component Isolation

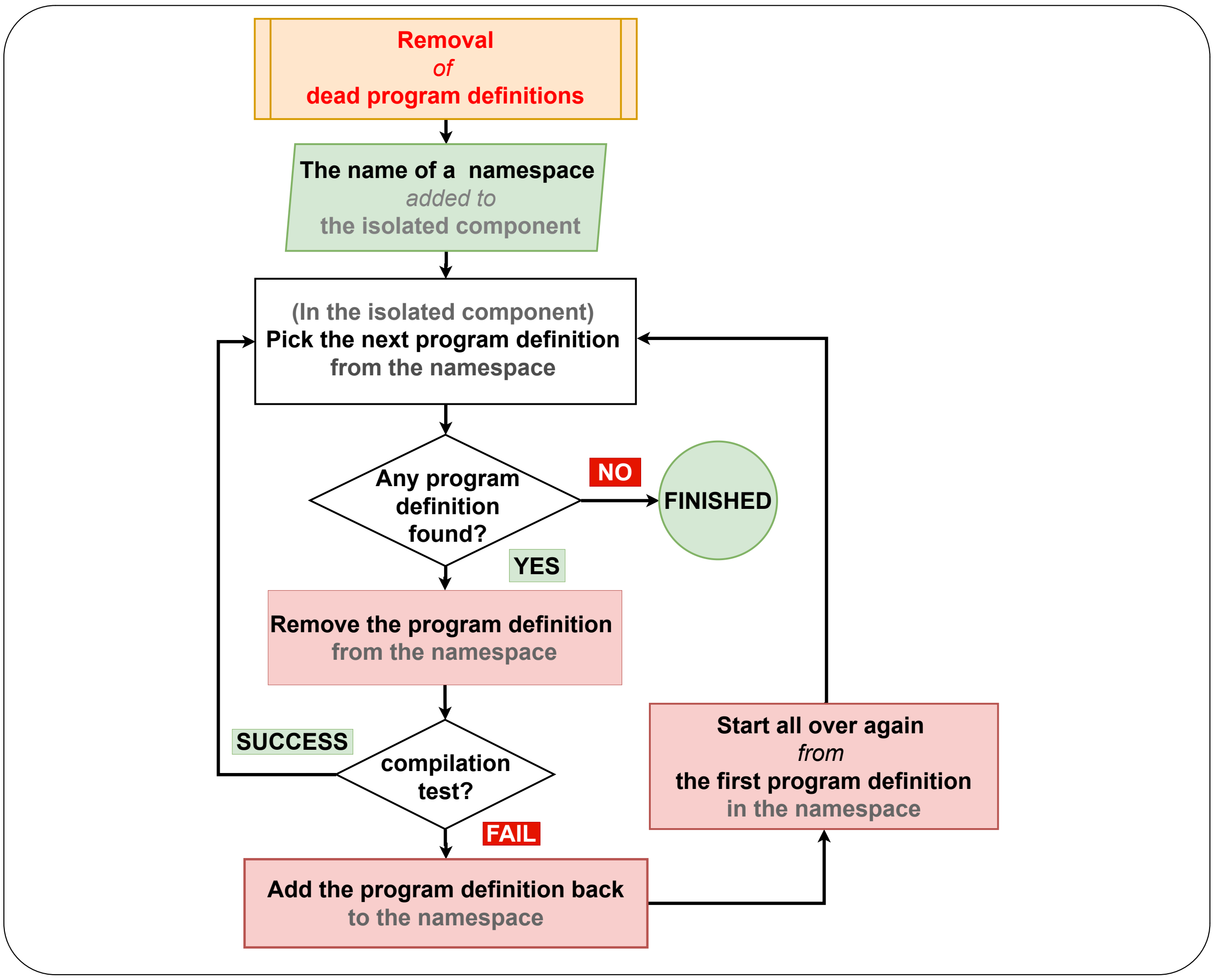


**Figure 6.27.:** Removing dead program definitions from a namespace

**Ignoring dead contents w.r.t. a Fortran program:**
Component Extraction and Isolation ignores dead contents w.r.t. a Fortran program to expedite the process. This is because the existence of such contents in a program is considered a wrong practice in software development, and, thus, a Fortran program is expected to be free of dead contents in advance. In addition, this topic has already been addressed in the literature, and, thus, is out of the scope of this dissertation. Above all, such dead contents do not have any negative impact on the semantic of an extracted or isolated component.

**Ignoring non-shared namespaces:**
So far, we assumed that the procedure of dead contents removal must be applied to every namespace that is added to an extracted or isolated component. However, the non-shared namespaces should be ignored in this process and only shared namespaces must be examined for dead contents. The reason is the dead contents w.r.t. the component or the carvedout program in a non-shared namespace are considered dead w.r.t. the program as well, and, thus, they should be ignored for the same reason explained before. As a result, non-shared namespaces should be exempted from the procedure of dead contents removal (in Step 3 and Step 4) during the process of Component Extraction and Isolation.

**Ignoring independent namespaces:**
If a namespace does not have any dependency on the other namespaces of a Fortran program, it will not lead to further processing even though it contains dead contents. So independent namespaces should be exempted from the procedure of dead contents removal, purely in pursuit of expediting the process of Component Extraction and Isolation.

#### 6.5.4.6. Handling dependency orders

In general, there is a dependency order, and, thus, a priority in the processing of the namespaces of a Fortran program during the procedures of Component Extraction and Isolation. Processing a namespace in a wrong order can potentially lead to the following problems:

- False addition of information:
  Assuming that a component has a correct dependency on the namespace A, but a wrong dependency on B (through some dead contents in A w.r.t.the component). If B is processed first, it leads to a false addition of information. Thus, it is advisable to process A before B to respect the dependency order. Since the wrong dependency is removed during the process of dead contents removal, processing the namespaces B will not be needed any longer. Hence, a correct dependency order can reduce the processing time of Compo-

nent Extraction and Isolation as well as the size of the extracted or isolated component.

- False omission of information:
  This problem happens when useful information from a namespace is removed due to a wrong dependency order. For example, consider a component has correct dependencies on the namespaces A, B and C. Assuming that A and B have a dependency on C. The processing order A, B and C or B, A and C are correct and no useful information in C is omitted. However, the order A, C, and B can lead to an information loss and an extra processing time. Consider C contains variables c1 and c2 while A uses c1 and B uses c2. If A is processed first, and, then, C before B, the procedure of dead contents removal omits the variable c2 from the namespace C as A uses only c1, and, there is not any other namespaces (such as B) to request for c2. However, when B is finally processed, c2 must be added back to C. On this account, a wrong order of dependency handling can potentially lead to a temporary loss of information and imposes an extra processing on the procedure of Component Extraction and Isolation.

#### 6.5.4.7. Considering dependency orders

As discussed, a wrong dependency order may increase the processing time of Component Extraction and Isolation, thus these procedures must be improved in this regard. As shown in Figure 6.28, Step 2 is adapted to take care of dependency orders between the namespaces of a component by searching for a top namespace in each iteration. Note that there must be at least one top namespace in each iteration, but chances are multiple top namespaces exist simultaneously in one iteration. However, it should not matter which top namespace is processed first.

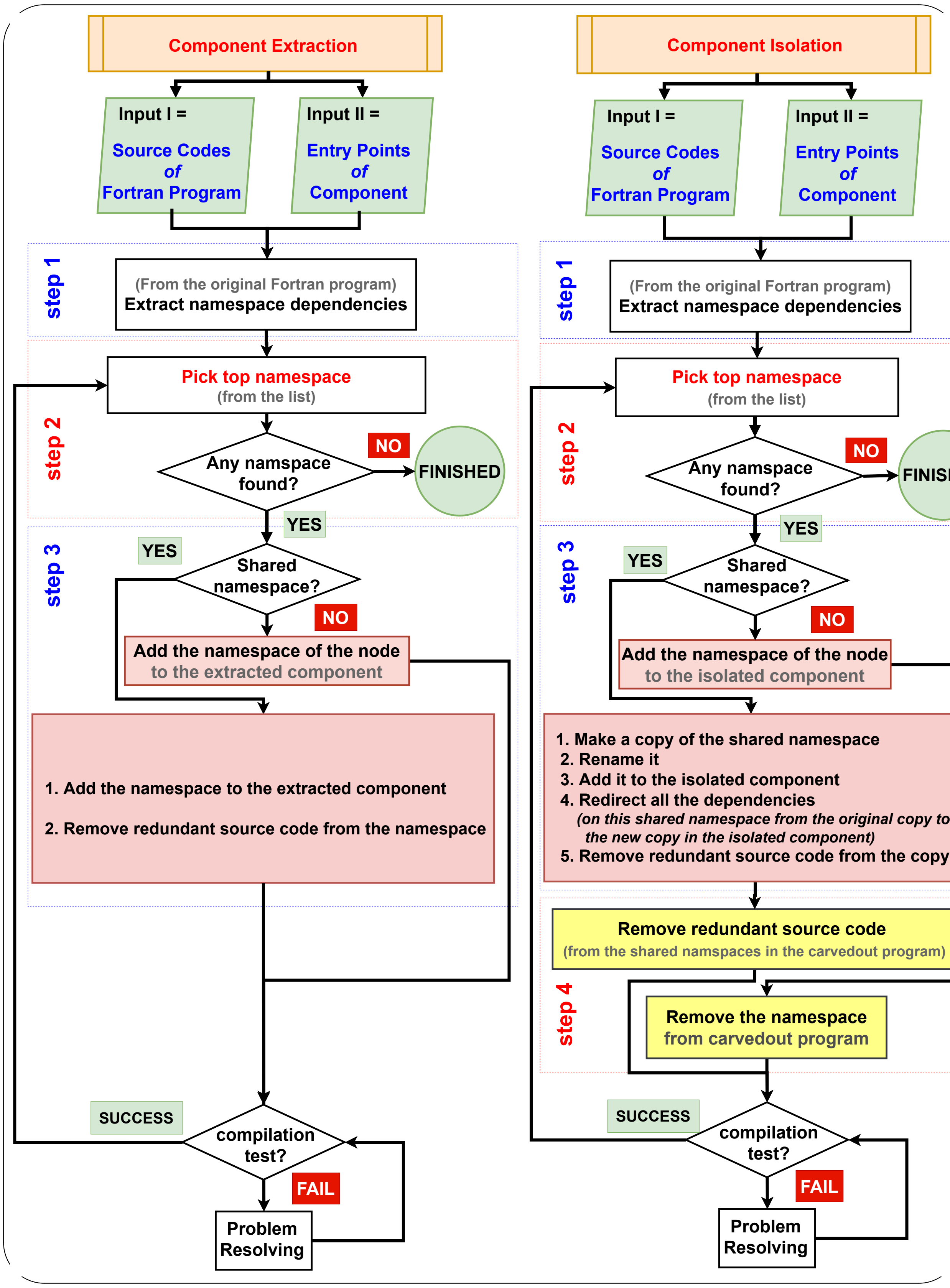

Component Extraction
Input I = Source Codes of Fortran Program
Input II = Entry Points of Component
step 1
(From the original Fortran program) Extract namespace dependencies
step 2
Pick top namespace (from the list)
Any namspace found?
NO
FINISHED
YES
step 3
Shared namespace?
YES
NO
Add the namespace of the node to the extracted component
1. Add the namespace to the extracted component
2. Remove redundant source code from the namespace
SUCCESS
compilation test?
FAIL
Problem Resolving
Component Isolation
Input I = Source Codes of Fortran Program
Input II = Entry Points of Component
step 1
(From the original Fortran program) Extract namespace dependencies
step 2
Pick top namespace (from the list)
Any namspace found?
NO
FINISHED
YES
step 3
Shared namespace?
YES
NO
Add the namespace of the node to the isolated component
1. Make a copy of the shared namespace
2. Rename it
3. Add it to the isolated component
4. Redirect all the dependencies (on this shared namespace from the original copy to the new copy in the isolated component)
5. Remove redundant source code from the copy
step 4
Remove redundant source code (from the shared namspaces in the carvedout program)
Remove the namespace from carvedout program
SUCCESS
compilation test?
FAIL
Problem Resolving

**Figure 6.28.:** Modifying the procedures of Component Extraction and Isolation to account for the dependency orders between the namespaces of a component. In the new scheme, a top namespace is always picked in Step 2 to be be processed first.

### 6.5.5. Implementation V: NDG graph

In the previous section, we discussed the approach of Component Extraction and Isolation that proposes to leverage the dependencies between the namespaces of a Fortran program to extract a component and isolate it from the carvedout program. This section introduces a graph processing technique for storing and processing the dependency information of the namespaces in a Fortran program. We call this graph a *Namespace Dependency Graph* (*NDG*). In particular, this technique affects Step 1, 2 and 3 of Component Extraction and Isolation.

#### 6.5.5.1. Using an NDG graph in Step 1

Step 1 of Component Extraction and Isolation collects all the dependencies between namespaces of a Fortran program. In this section, we will show how an NDG can be exploited to implement this step.

**An NDG graph for storing namespace dependencies**

A namespace dependence graph (NDG) is a directed acyclic graph whose vertices are the namespaces of a Fortran program and its edges show the dependency relations between the namespaces of the program. A simple NDG graph with two vertices A and B and an edge incident on A and B represents two namespaces A and B in a Fortran program and the fact that the namespace A depends on some definitions in the namespace B. This dependency implies that the source code in the namespace A cannot compile successfully if the definitions in the namespace B are not available. A vertex in an NDG graph has one of the following two types:

- A subprogram namespace in the global namespace
- A module namespace

**Creating the complete NDG of a Fortran program**

An NDG graph can be used to store and present the complete dependency relations between the namespace of a Fortran program. An NDG is built by creating an empty directed graph and adding one source node for the main program unit and one source node for each entry point of the component. In the following steps, the dependencies of the main program unit and the entry points of the component on the other namespaces of the original program are extracted (as explained in Section 6.5.1.1) and one node is added to the graph for each namespace.

#### 6.5.5.2. Using the NDG graph in Step 2

Step 2 in Component Extraction and Isolation is responsible for processing the collected information concerning the dependencies between the namespaces of a Fortran

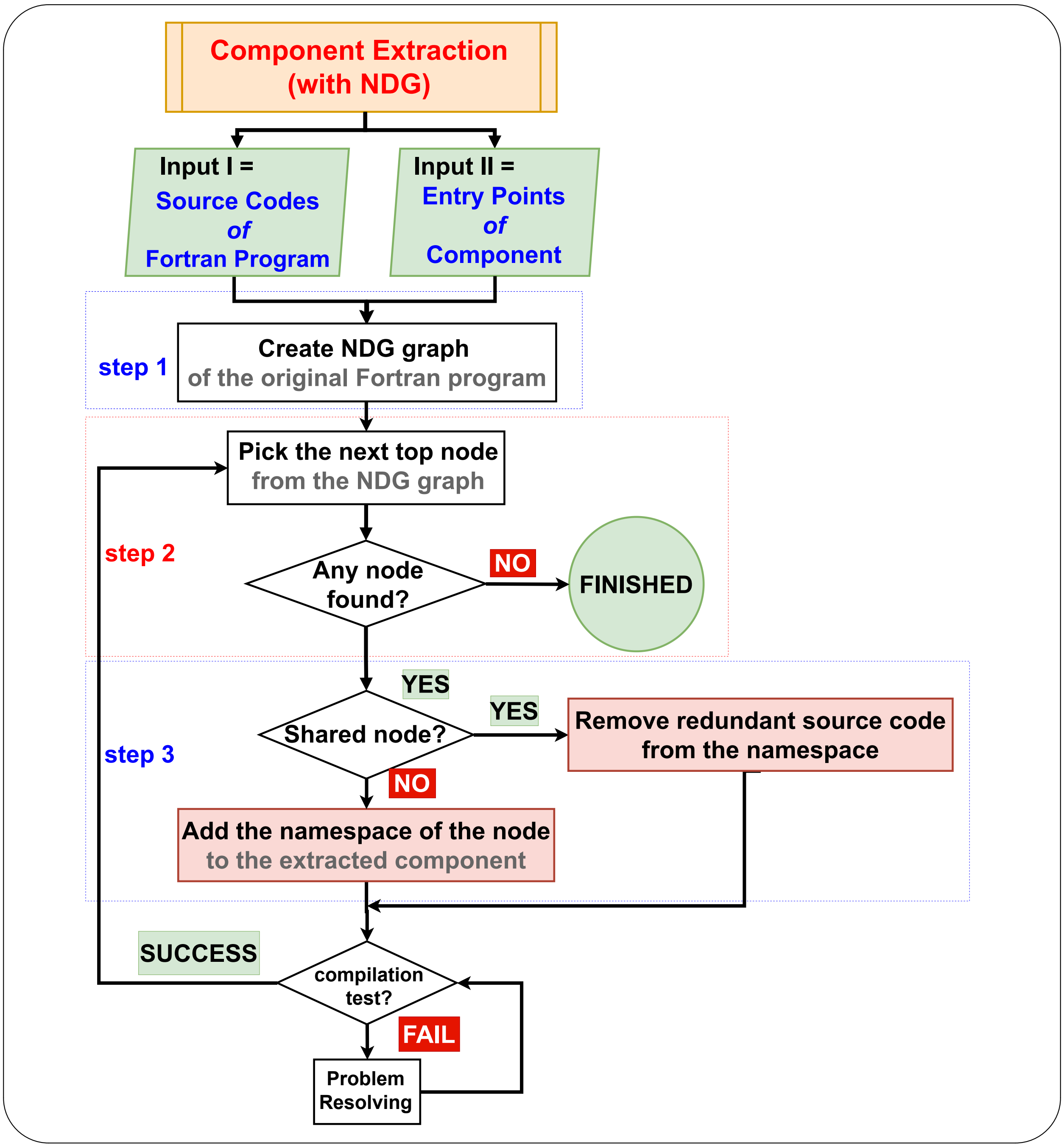

Component Extraction (with NDG)
Input I = Source Codes of Fortran Program
Input II = Entry Points of Component
step 1
Create NDG graph of the original Fortran program
step 2
Pick the next top node from the NDG graph
Any node found?
NO
FINISHED
YES
step 3
Shared node?
YES
Remove redundant source code from the namespace
NO
Add the namespace of the node to the extracted component
SUCCESS
compilation test?
FAIL
Problem Resolving

**Figure 6.29.:** Leveraging an NDG graph in Component Extraction.

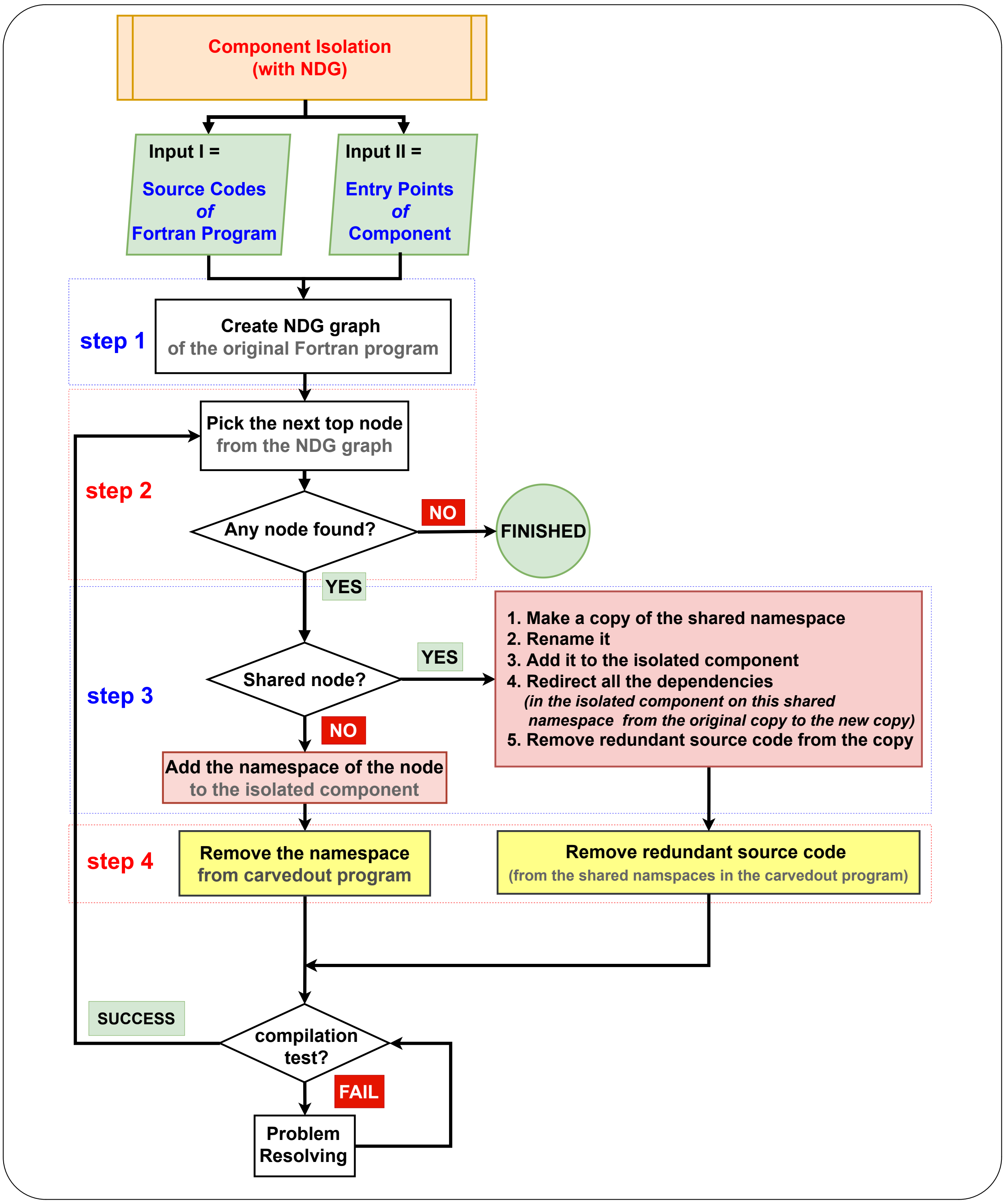

Component Isolation (with NDG)
Input I = Source Codes of Fortran Program
Input II = Entry Points of Component
step 1
Create NDG graph of the original Fortran program
step 2
Pick the next top node from the NDG graph
Any node found?
NO
FINISHED
YES
step 3
Shared node?
YES
1. Make a copy of the shared namespace
2. Rename it
3. Add it to the isolated component
4. Redirect all the dependencies (in the isolated component on this shared namespace from the original copy to the new copy)
5. Remove redundant source code from the copy
NO
Add the namespace of the node to the isolated component
step 4
Remove the namespace from carvedout program
Remove redundant source code (from the shared namspaces in the carvedout program)
SUCCESS
compilation test?
FAIL
Problem Resolving

**Figure 6.30.:** Leveraging an NDG graph in Component Isolation

program and detecting the namespaces on which the entry points of a component depend. As explained before, an NDG graph can be used to store these dependency relations and extracting the required information. In addition, we can take advantage of such a graph to identify a top namespace in each iteration. Such a namespace is a node (denoted as the top node) in the graph that can be reached from at least one entry point of the component, and, no path from the entry points to the top node reaches an unvisited node. An unvisited node is the one that its corresponding namespace has not been added to the extracted or isolated component yet. In each iteration, at least one node becomes a top node and the topological sort algorithm (Wikipedia, last access: 4 May 2022) would serve the purpose in order to find the top namespace. Figure 6.29 and Figure 6.30 explicitly reflect the usage of the NDG graph in Step 2 of Component Extraction and Isolation.

#### 6.5.5.3. Using the NDG graph in Step 3

In Step 3, we need to find shared namespaces. The NDG graph can be used to detect shared namespaces. A shared namespace resides in a node that is reached from at least one entry point of the component and the entry point of the carvedout program. Note that no path between the main program unit and a shared node can visit the entry points of a component since all the call-sites to the entry points are removed from the program before the dependency analysis.

## 6.6. Examples of Component Isolation

This section provides four Fortran programs that each contains one component. These examples should be able to articulate the concept of Component Extraction and Isolation. In all the examples, we explicitly discuss the procedure of Component Isolation, but we show only the results for the Component Extraction as well for comparison. In each example, the source code of the original program is given and the required information on the entry point of the components is provided in a table below the source code of each Fortran program. The components differ in the depth, type and how they share contents with the program. In each example, an NDG graph is built and the extracted and isolated components as well as the carvedout program are generated. We use the color codes (defined in section Section 6.1) to differentiate the isolated (or extracted) component (always shown in pink) from a carvedout program (always shown in yellow). In all examples, the extracted and isolated components are initially empty spaces. In contrast, the complete source code of the original program is assigned to the carvedout program in the beginning though the extra parts are removed incrementally. In addition, we always ignore the call-sites to the entry points of the component while extracting the dependencies of the carvedout program. Furthermore, when all the nodes of the NDG are visited and the processing of the last namespace is complete, the existing call-sites to the entry points of the component are removed from the carvedout program.

### 6.6.1. Example 1

Figure 6.31 shows a Fortran program and a component. We apply Component Isolation to this program to isolate the component from the program. The input to the procedure is the source code of the Fortran program and the name of the entry point of the component, which is the subprogram namespace "*subroutine1*". The output of this process will be an isolated component and a carvedout program.

**Step 1: Creating NDG graph** In this step, a dependency analysis is performed to extract the dependency information of the component and the carvedout program and to store the information in an NDG graph. The analysis is performed on the source code of the original Fortran program that is available in the carvedout program.

We start building the NDG graph by assigning one vertex to the namespace of the entry point of the component (i.e. *subroutine1*). Since this namespace does not have any further dependency, the dependency analysis of the component is over. Hence, we switch to extracting the dependency of the carvedout program by assigning one vertex (*i.e. MainProgram*) to its entry point (i.e. the main program unit). The dependency of the main program unit on the entry point of the component is, however, ignored, and, thus, no edge is placed between the two vertices. Since there is no further dependencies in the main program unit, the analysis is over and the NDG graph is complete as shown Figure 6.32.

It is noteworthy that the internal program statements of "*MainProgram*" and *"subroutine1"* do not appear in the NDG graph as they are irrelevant information. The tables in Figure 6.32 show that the extracted dependency information in more detail. As it is shown, the component contains a non-shared module namespace (*subroutine1*) inside the global namespace of one non-shared file (*originalprogram_file1.f90*). By the same token, the carvedout program has only one subprogram namespace (*MainProgram*) in the global namespace of a non-shared file (*originalmainprogram_file1.f90*).

#### 6.6.1.1. Iteration 1

In the first iteration, the following steps are performed:

**Step 2: Picking the next top node** In this step, the graph is traversed, starting from the vertex of the entry point of the component, to pick the top node (which is the vertex *"subroutine1").*

**Step 3: Adding the namespace to the isolated component** In this step, the definition of the namespace associated to the top vertex (i.e. *subroutine1*) is added to the isolated component. The definition is, however, placed into a new file "*isolatedcomponent_file.f90*".

**Step 4: Remove the namespace from the carvedout program** In this step, the definition of "*subroutine1*" is removed from the carvedout program completely since it is not a shared namespace (as the graph does not contain any path from the vertex "*MainProgram*" to *"module1"*.

#### 6.6.1.2. Iteration 2

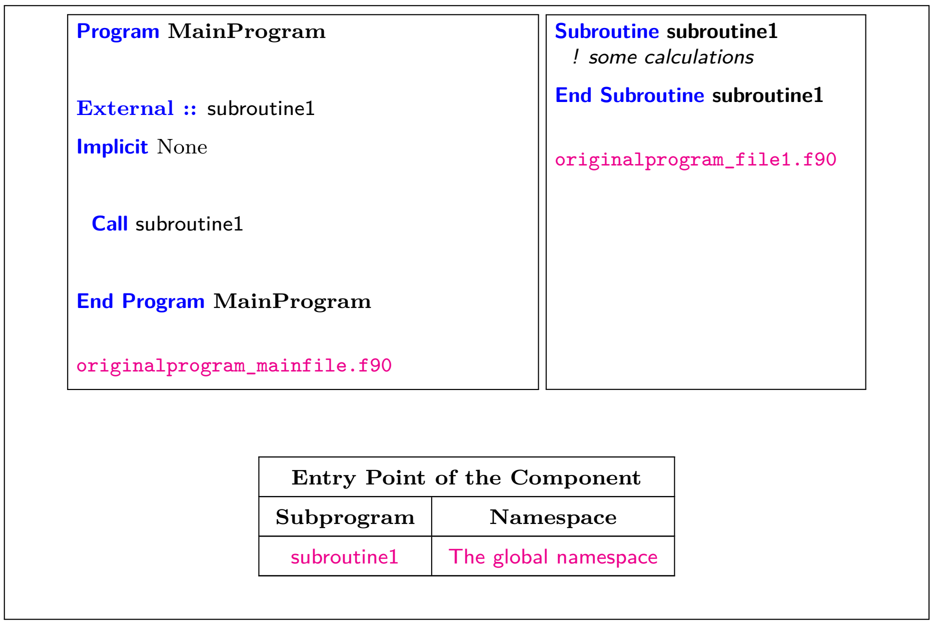
In the second iteration, only Step 2 takes place.

**Step 2: Picking the next top node** The practice stops in the second iteration since the graph does not show any path from the vertex "*subroutine1*".

#### 6.6.1.3. Results

Once the procedure stops, the isolated component and the carvedout program are available. As shown in Figure 6.34, the carvedout program (shown in yellow) contains "*MainProgram*", whose definition has been moved into a new file called *"carvedoutprogram_file.f90"*. In addition, the dependency of the main program unit on the entry point of the component has been removed from the file to prevent the compilation failure of the carvedout program. Hence, the call-site (*call subroutine1*) and the dependency link (*External:: subroutine1*) are struck through (temporarily) in *"carvedoutprogram_file.f90"*. Thus, the carvedout program is now expected to compile successfully independent of the isolated component. Note that these statements can be activated again once the isolated component is re-integrated back to the carvedout program to make the new program capable of generating bit-wise identical results to the output of the original program.

Additionally, the isolated component (shown in pink) contains only " *subroutine1*" (thus the component has a depth of 1), whose definition has been placed into a new file "*isolatedcomponent_file.f90*" to increase the visibility of the component. Thus, the isolated component is now expected to compile successfully independent of the carvedout program.

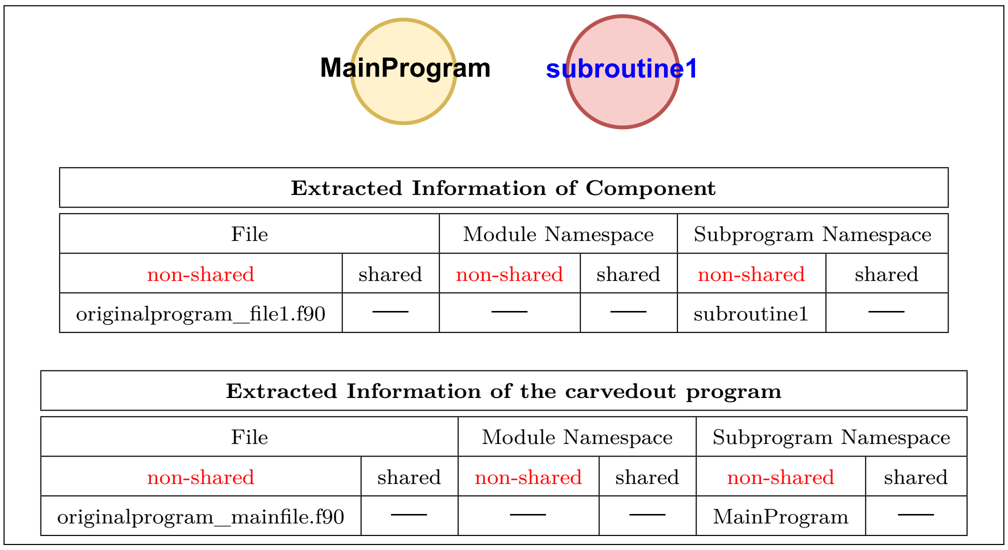


```
Program MainProgram


External :: subroutine1
Implicit None


  Call subroutine1


End Program MainProgram

originalprogram_mainfile.f90
```

```
Subroutine subroutine1
  ! some calculations
End Subroutine subroutine1

originalprogram_file1.f90
```

| Entry Point of the Component | |
|---|---|
| **Subprogram** | **Namespace** |
| subroutine1 | The global namespace |

**Figure 6.31.:** (Example 1: Original Program) A simple Fortran program with one component and two namespaces in the global namespace. The information about the entry point of the component is shown in the table.



| Extracted Information of Component | | | | | |
|---|---|---|---|---|---|
| File | | Module Namespace | | Subprogram Namespace | |
| non-shared | shared | non-shared | shared | non-shared | shared |
| originalprogram_file1.f90 | — | — | — | subroutine1 | — |

| Extracted Information of the carvedout program | | | | | |
|---|---|---|---|---|---|
| File | | Module Namespace | | Subprogram Namespace | |
| non-shared | shared | non-shared | shared | non-shared | shared |
| originalprogram_mainfile.f90 | — | — | — | MainProgram | — |

**Figure 6.32.:** (Example 1: dependency information) Applying Component Isolation to the Fortran program in Figure 6.31 and extracting the information about the namespaces of the program and component.

```
Subroutine subroutine1

  PRINT *,"HELLO WORLD"
End Subroutine subroutine1

extractedcomponent_file.f90
```

**Figure 6.33.:** (Example 1: extracted component) The extracted component generated from the Fortran program in Figure 6.31 by the procedure of Component Extraction .

```
Program MainProgram


~~External :: subroutine1~~
Implicit None


~~Call subroutine1~~


End Program MainProgram

carvedoutprogram_file.f90
```

```
Subroutine subroutine1

  PRINT *,"HELLO WORLD"
End Subroutine subroutine1

isolatedcomponent_file.f90
```

**Figure 6.34.:** (Example 1: isolated component and carvedout program) The isolated component (in pink) and carvedout program (in yellow) generated from the Fortran program in Figure 6.31 by Component Isolation.

### 6.6.2. Example 2

Figure 6.35 shows a Fortran program and a component whose entry point is a subprogram named “*subroutine1*”, but contained in a module namespace. We apply Component Isolation to this program. Like the previous example, the isolated component is initially an empty space, but the complete source code of the original program is assigned to the carvedout program.

**Step 1: Creating NDG graph**
We start building the NDG graph by assigning one vertex to the namespace of the entry point of the component. Since the definition of *“subroutine1”* is inside a module namespace rather than the global namespace, the vertex is assigned to *“module1”*. As “*module1*” does not have further dependencies, thus the analysis of the component is complete. Hence, we switch to extracting the dependency of the carvedout program by assigning one vertex (*i.e. MainProgram*) to its entry point. The dependency of the main program unit on the entry point of the component is, however, ignored, and, thus, no edge is placed between the vertices *“MainProgram*” and *“module1”*. Since there is no further dependencies in the main program unit, the analysis is over and the NDG graph is complete as shown Figure 6.35.

#### 6.6.2.1. Iteration 1

In the first iteration, the following steps are performed:

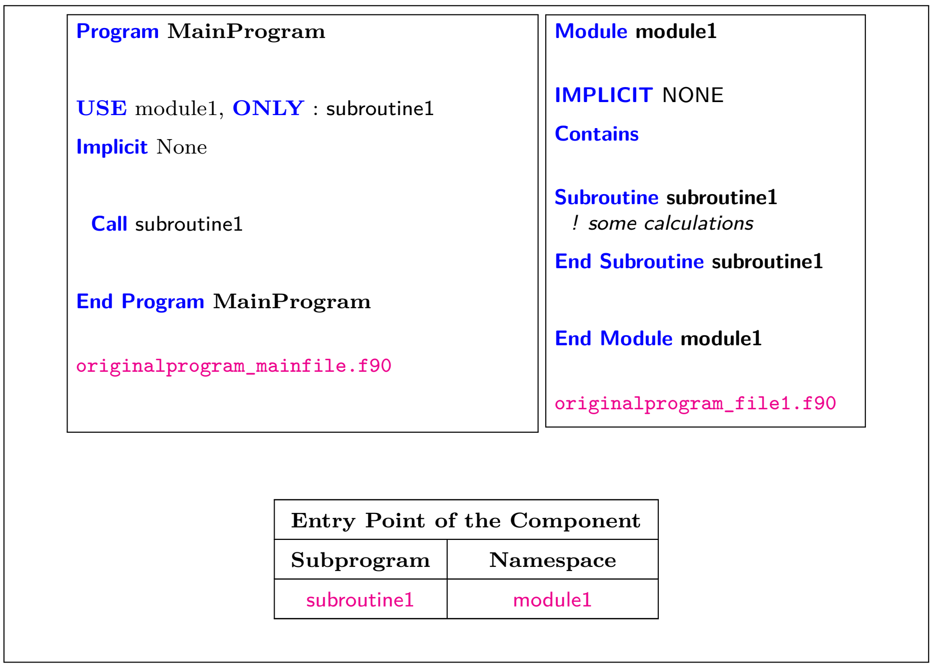
**Step 2: Picking the next top node** In this step, the graph is traversed, starting from the vertex of the entry point of the component, to pick the top vertex (which is the vertex *“module1”*).

**Step 3: Add the namespace to the isolated component** In this step, the definition of the namespace associated to the top vertex (i.e. *module1*) is added to the isolated component. The definition is, however, placed into a new file “*isolatedcomponent_file.f90*”.

**Step 4: Remove the namespace from the carvedout program** In this step, the definition of “*module1*” is removed from the carvedout program completely since it is not a shared namespace (as the graph does not contain any path from the vertex “*MainProgram*” to *“module1”*).

#### 6.6.2.2. Iteration 2

In the second iteration, the practice stops in Step 2 as the graph does not contain any path from the vertex “*module2*”.



```
Program MainProgram

USE module1, ONLY : subroutine1
Implicit None

  Call subroutine1

End Program MainProgram

originalprogram_mainfile.f90
```

```
Module module1

IMPLICIT NONE
Contains

Subroutine subroutine1
  ! some calculations
End Subroutine subroutine1

End Module module1

originalprogram_file1.f90
```

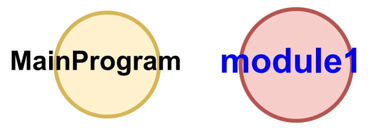
| Entry Point of the Component | |
|---|---|
| **Subprogram** | **Namespace** |
| subroutine1 | module1 |

**Figure 6.35.:** (Example 2: Original Program) A simple Fortran program with one component and two namespaces in the global namespace. The information about the entry point of the component is shown in the table.

### 6.6.2.3. Results

Once the procedure stops, the isolated component and the carvedout program are available. As shown in Figure 6.38, the carvedout program (shown in yellow) contains "*MainProgram*", whose definition has been moved into a new file called *"carvedoutprogram_file.f90"*. In addition, the dependency of the carvedout program on the entry point of the component has been removed from the file to prevent the compilation failure of the carvedout program. Hence, the call-site (*call subroutine1*) and the dependency link (*USE module1, ONLY: subroutine1*) are struck through (temporarily) in *"carvedoutprogram_file.f90"*. Thus, the carvedout program is now expected to compile successfully independent of the isolated component. Note that these statements can be activated again once the isolated component is re-integrated back to the carvedout program to make the new program capable of generating bit-wise identical results to the output of the original program.

Additionally, the isolated component (shown in pink) contains only "*module1*" (thus the component has a depth of 1), whose definition has been placed into a new file "*isolatedcomponent_file.f90*" to increase the visibility of the component. Thus, the isolated component is now expected to compile successfully independent of the carvedout program.



| Extracted Information of Component | | | | | |
|---|---|---|---|---|---|
| File | | Module Namespace | | Subprogram Namespace | |
| non-shared | shared | non-shared | shared | non-shared | shared |
| X | — | X | — | — | — |

| Extracted Information of the carvedout program | | | | | |
|---|---|---|---|---|---|
| File | | Module Namespace | | Subprogram Namespace | |
| non-shared | shared | non-shared | shared | non-shared | shared |
| X | — | — | — | X | — |

**Figure 6.36.:** (Example 2: dependency information) Applying Component Isolation to the Fortran program in Figure 6.35 and extracting the information about the namespaces of the program and component.

### 6.6.3. Example 3

Figure 6.39 shows a Fortran program and a component whose entry point is a subprogram named "*subroutine1*", but contained in a module namespace. In contrast to Example 1 and 2, the component has a depth 2. We apply Component Isolation to this program and explain how it differs from the previous examples. Like the previous examples, the isolated component is initially an empty space, but the complete source code of the original program is assigned to the carvedout program.

**Step 1: Creating NDG graph** The NDG graph of the Fortran program in this example will be generated as in the Example 2. However, "*module1*" also has a dependency on "*module2*" as there is a call to "*subroutine2*" from "*subroutine1*". So another vertex (*module2*) is added to the graph to show the dependency. Since "*module2*" does not have further dependencies, the analysis of the component is over and the NDG graph is complete as shown in Figure 6.39. It is noteworthy that "*subroutine2*" does not appear in the graph as it is not in the global namespace, but inside another namespace (*module2*). As shown by the tables in this figure, the component contains two non-shared module namespaces in two non-shared files (*originalprogram_file1.f90* and *"originalprogram_file2.f90"*). The carvedout program is the same as Example 2.

```
Module module1

IMPLICIT None
Contains

Subroutine subroutine1

  ! some calculations
End Subroutine subroutine1


End Module module1

extractedcomponent_file.f90
```

**Figure 6.37.:** (Example 2: extracted component) The extracted component generated from the Fortran program in Figure 6.35 by the procedure of Component Extraction.

```
Program MainProgram


~~USE module1, ONLY : subroutine1~~
Implicit None


  ~~Call subroutine1~~


End Program MainProgram

carvedoutprogram_file.f90
```

```
Module module1

IMPLICIT None
Contains

Subroutine subroutine1

  ! some calculations
End Subroutine subroutine1


End Module module1

isolatedcomponent_file.f90
```

**Figure 6.38.:** (Example 2: isolated component and carvedout program) The isolated component (in pink) and carvedout program (in yellow) generated from the Fortran program in Figure 6.35 by Component Isolation.

#### 6.6.3.1. Iteration 1

In the first iteration, similar to Example 2, the namespace *“module1”* is added to the isolated component.

#### 6.6.3.2. Iteration 2

In the second iteration, Step 2, 3 and 4 take place.

**Step 2: Picking the next top node** In contrast to previous examples, the procedure does not stop in iteration 2 and the vertex *“module2”* is picked as the top vertex.

**Step 3: Add the namespace to the isolated component** In this step, the definition of the namespace associated to the top vertex (i.e. *module*2) is added to the isolated component. The definition is, however, placed into a new file “*isolatedcomponent_file2.f90*”.

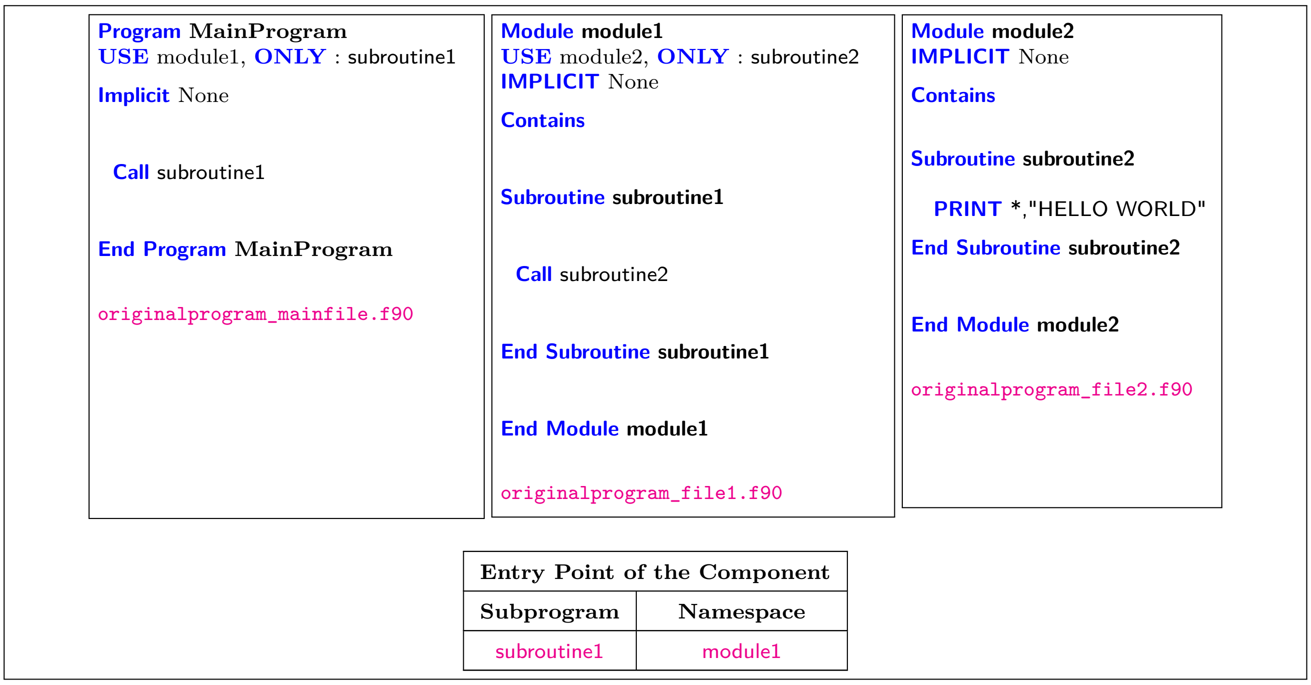
**Step 4: Remove the namespace from the carvedout program** In addition, the definition of “*module2*” is removed from the carvedout program completely since it is not a shared namespace (as the graph does not contain any path from the vertex “*MainProgram*” to *“module2”*.

#### 6.6.3.3. Iteration 3

In the third iteration, the practice stops in Step 2 as the graph does not contain any path from the vertex “*module2*”.

#### 6.6.3.4. Results

Once the procedure stops, the isolated component and the carvedout program are available. As shown in Figure 6.42, the carvedout program contains “*MainProgram*”, whose definition has been moved into a new file called *“carvedoutprogram_file.f90”*. In addition, the dependency of the carvedout program on the entry point of the component has been removed from the file to prevent the compilation failure of the carvedout program. Hence, the call-site (*call subroutine1*) and the dependency link (*USE module1, ONLY: subroutine1*) are struck through (temporarily) in *“carvedoutprogram_file.f90”*. Thus, the carvedout program is now expected to compile successfully independent of the isolated component. Note that these statements can be activated again once the isolated component is re-integrated back to the carvedout program to



```
Program MainProgram
USE module1, ONLY : subroutine1
Implicit None


  Call subroutine1


End Program MainProgram
```

originalprogram_mainfile.f90

```
Module module1
USE module2, ONLY : subroutine2
IMPLICIT None
Contains


Subroutine subroutine1


  Call subroutine2


End Subroutine subroutine1


End Module module1
```

originalprogram_file1.f90

```
Module module2
IMPLICIT None
Contains


Subroutine subroutine2

  PRINT *,"HELLO WORLD"
End Subroutine subroutine2


End Module module2
```

originalprogram_file2.f90

| Entry Point of the Component | |
|---|---|
| **Subprogram** | **Namespace** |
| subroutine1 | module1 |

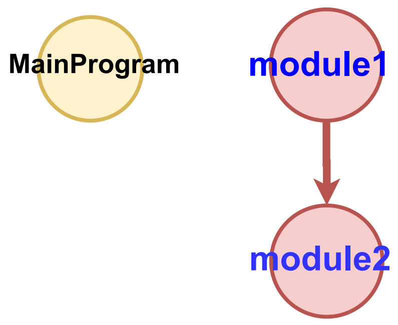
**Figure 6.39.:** (Example 3: Original Program) A Fortran program with one component and three namespaces in the global namespace. The information about the entry point of the component is shown in the table.

make the new program capable of generating bit-wise identical results to the output of the original program.

Additionally, the isolated component contains only "*module1*" and "*module2*" (thus the component has a depth of 2), whose definitions have been placed into a new file "*isolatedcomponent_file1.f90*" and "*isolatedcomponent_file2.f90*" in a new folder to increase the visibility of the component. Thus, the isolated component is now expected to compile successfully independent of the carvedout program.

### 6.6.4. Example 4

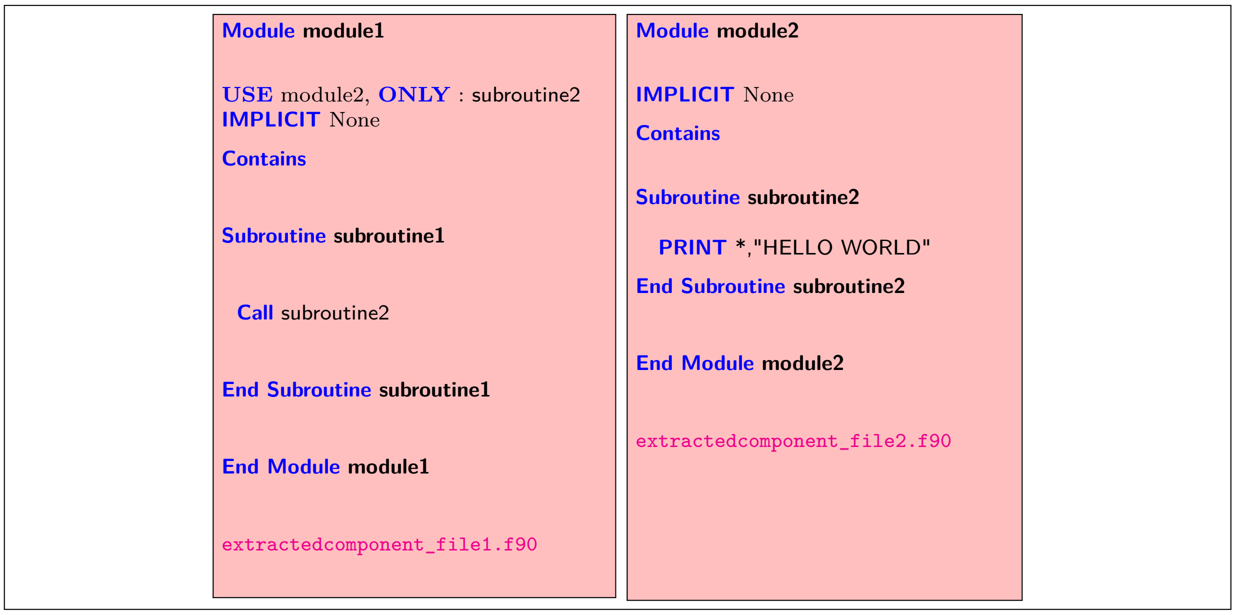
In this example, the Fortran program of Example 3 has been modified so that there is a shared namespace between the component and carvedout program, as shown in Figure 6.43. We apply Component Isolation to this program and explain how it differs from the previous example. Like the previous examples, the isolated component is initially an empty space, but the complete source code of the original program is assigned to the carvedout program.



| **Extracted Information of Component** | | | | | |
|---|---|---|---|---|---|
| File | | Module Namespace | | Subprogram Namespace | |
| non-shared | shared | non-shared | shared | non-shared | shared |
| **X** | — | **X** | — | — | — |

| **Extracted Information of the carvedout program** | | | | | |
|---|---|---|---|---|---|
| File | | Module Namespace | | Subprogram Namespace | |
| non-shared | shared | non-shared | shared | non-shared | shared |
| **X** | — | — | — | **X** | — |

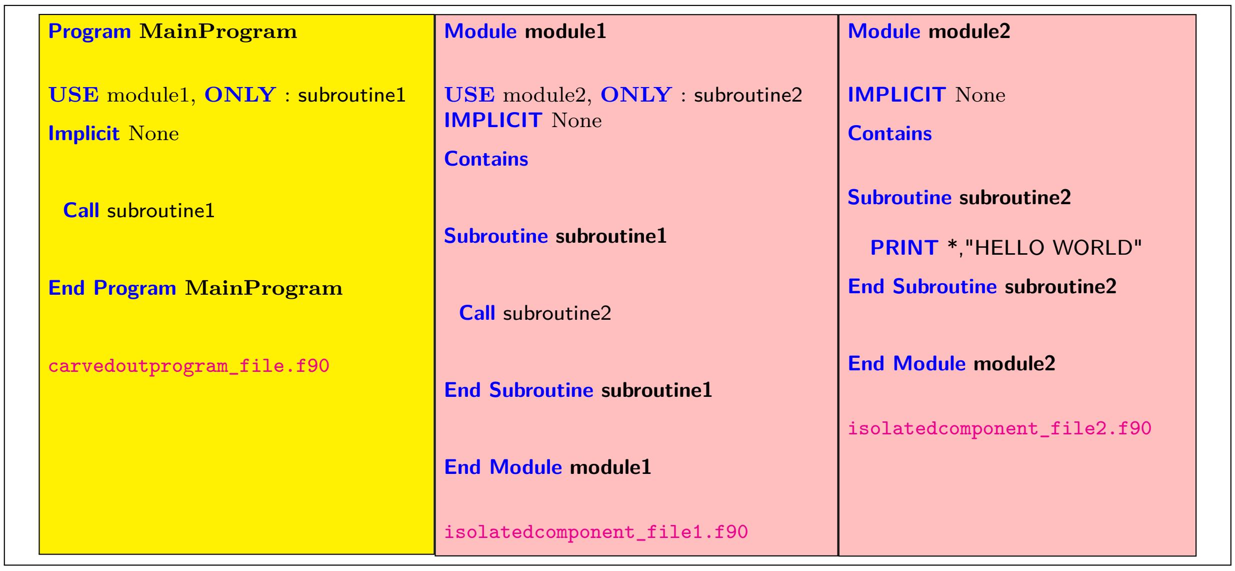
**Figure 6.40.:** (Example 3: dependency information) Applying Component Isolation to the Fortran program in Figure 6.39 and extracting the information about the namespaces of the program and component.



```
Module module1

USE module2, ONLY : subroutine2
IMPLICIT None
Contains


Subroutine subroutine1


  Call subroutine2


End Subroutine subroutine1


End Module module1


extractedcomponent_file1.f90
```

```
Module module2

IMPLICIT None
Contains

Subroutine subroutine2

  PRINT *,"HELLO WORLD"
End Subroutine subroutine2


End Module module2


extractedcomponent_file2.f90
```

**Figure 6.41.:** (Example 3: extracted component) The extracted component generated from the Fortran program in Figure 6.39 by the procedure of Component Extraction.



```
Program MainProgram

USE module1, ONLY : subroutine1
Implicit None


  Call subroutine1


End Program MainProgram


carvedoutprogram_file.f90
```

```
Module module1

USE module2, ONLY : subroutine2
IMPLICIT None
Contains


Subroutine subroutine1


  Call subroutine2


End Subroutine subroutine1


End Module module1


isolatedcomponent_file1.f90
```

```
Module module2

IMPLICIT None
Contains

Subroutine subroutine2

  PRINT *,"HELLO WORLD"
End Subroutine subroutine2


End Module module2


isolatedcomponent_file2.f90
```

**Figure 6.42.:** (Example 3: isolated component and carvedout program) The isolated component (in pink) and carvedout program (in yellow) generated from the Fortran program in Figure 6.39 by Component Isolation.

**Step 1: Creating NDG graph** We start building the NDG graph by assigning one vertex to the namespace of the entry point of the component. Since the definition of *"subroutine1"* is inside a module namespace rather than the global namespace, the vertex is assigned to *"module1"*. However, this module namespace also has a dependency on "*module2*" due to a call to "*subroutine2*" (from "*subroutine1*"). Thus, another vertex (*i.e. module2* ) is added to the graph. As "*module2*" has no further dependencies, the analysis of the component is complete. Hence, we switch to extracting the dependency of the carvedout program by assigning one vertex (*i.e. MainProgram*) to its entry point. The dependency of the main program unit on the entry point of the component is, however, ignored, and, thus, no edge is placed between the vertices *"MainProgram"* and (*module1*). Due to the dependency link *"USE module2, ONLY: subroutine2"* in the main program unit, an edge is established from the vertex *"MainProgram"* to other vertex *"module2"*. Since there is no further dependencies in the main program unit, the analysis is over and the NDG graph is complete as shown Figure 6.44.

#### 6.6.4.1. Iteration 1

In the first iteration, similar to Example 2, the namespace *"module1"* is added to the isolated component.

#### 6.6.4.2. Iteration 2

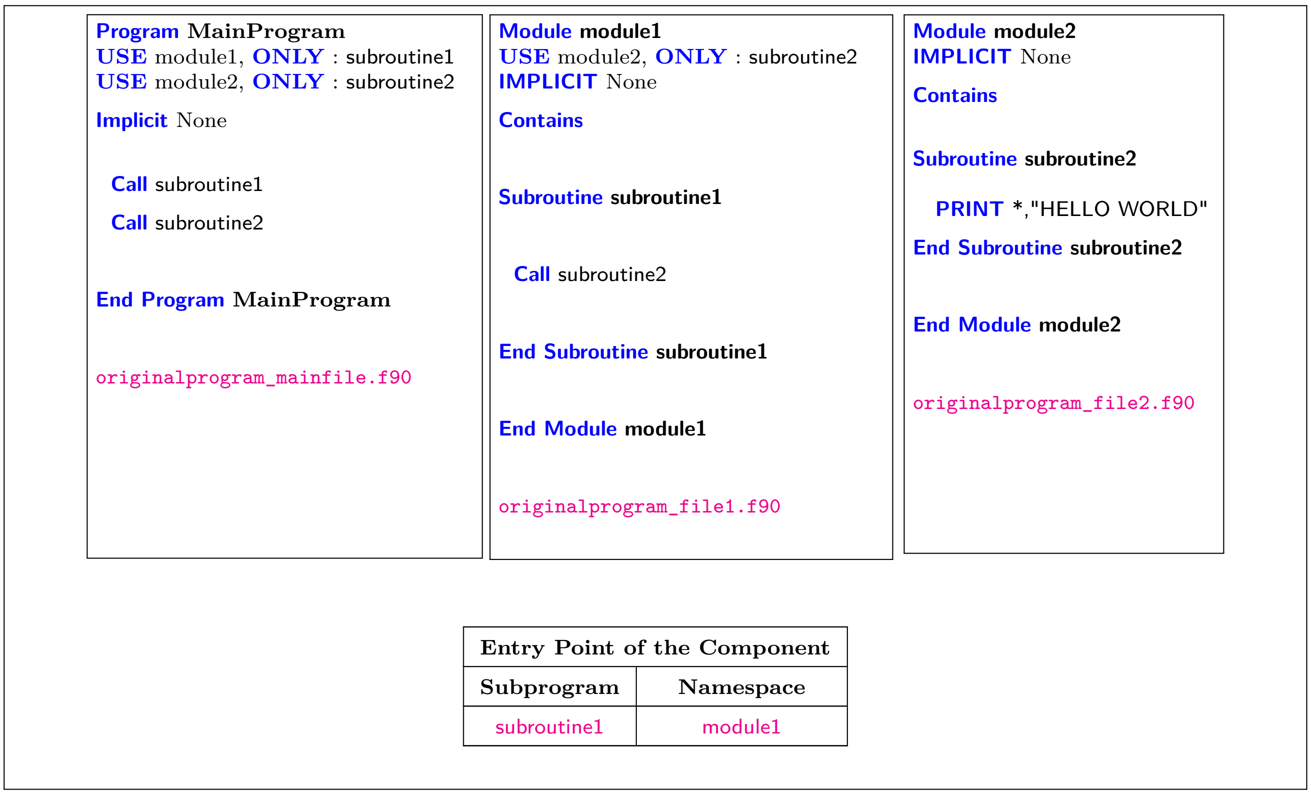
The second iteration is also similar to Example 3, but with a difference that, the definition of "*module2*" is not removed from the carvedout program since it is a shared namespace between the component and the carvedout program. This is because the vertex "*module2*" is reached from both vertices "*MainProgram*" and "*module*1". However, a copy of the definition of "*module2*" is added to the isolated component and it is renamed to "*module2_copy*". In addition, any reference to *"module2"* (in the isolated component) should be redirected to *"module2*_copy".

#### 6.6.4.3. Iteration 3

Similar to Example 3, the practice stops in the third iteration.

#### 6.6.4.4. Results

Once the procedure stops, the isolated component and the carvedout program are available. As shown in Figure 6.46, the carvedout program contains "*MainProgram*" and "*module2*", whose definitions have been moved into two new files called *"carvedoutprogram_file1.f90"* and *"carvedoutprogram_file2.f90"*, respectively. In addition, the dependency of the main program unit on the entry point of the component has been



```
Program MainProgram
USE module1, ONLY : subroutine1
USE module2, ONLY : subroutine2

Implicit None


  Call subroutine1

  Call subroutine2


End Program MainProgram


originalprogram_mainfile.f90
```

```
Module module1
USE module2, ONLY : subroutine2
IMPLICIT None

Contains


Subroutine subroutine1


  Call subroutine2


End Subroutine subroutine1


End Module module1


originalprogram_file1.f90
```

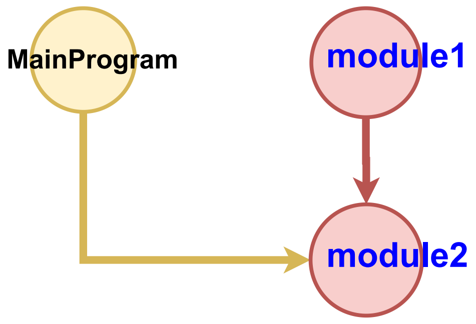
```
Module module2
IMPLICIT None

Contains


Subroutine subroutine2

  PRINT *,"HELLO WORLD"

End Subroutine subroutine2


End Module module2


originalprogram_file2.f90
```

| Entry Point of the Component | |
|---|---|
| Subprogram | Namespace |
| subroutine1 | module1 |

**Figure 6.43.:** (Example 4: Original Program) A Fortran program with one component and three namespaces in the global namespace. The information about the entry point of the component is shown in the table.

removed from the file to prevent the compilation failure of the carvedout program. Hence, the call-site (*call subroutine1*) and the dependency link (*USE module1, ONLY: subroutine1*) are struck through (temporarily) in *"carvedoutprogram_file1.f90"*. Thus, the carvedout program is now expected to compile successfully independent of the isolated component. Note that these statements can be activated again once the isolated component is re-integrated back to the carvedout program to make the new program capable of generating bit-wise identical results to the output of the original program.

Additionally, the isolated component contains only "*module1*" and "*module*2_copy" (thus the component has a depth of 2), whose definitions have been placed into new files "*isolatedcomponent_file1.f90*" and "*isolatedcomponent_file2.f90*" in a new folder to increase the visibility of the component. This should prevent any conflict with the original copy left in *"carvedoutprogram_file2.f90"* once the isolated component is re-integrated back to the carvedout program. Accordingly, the statement *"USE module2, ONLY: subroutine2"* has been converted to *"USE module2_copy, ONLY: subroutine2"* in *"isolatedcomponent_file2.f90"*. Thus, the isolated component is now expected to compile successfully independent of the carvedout program.



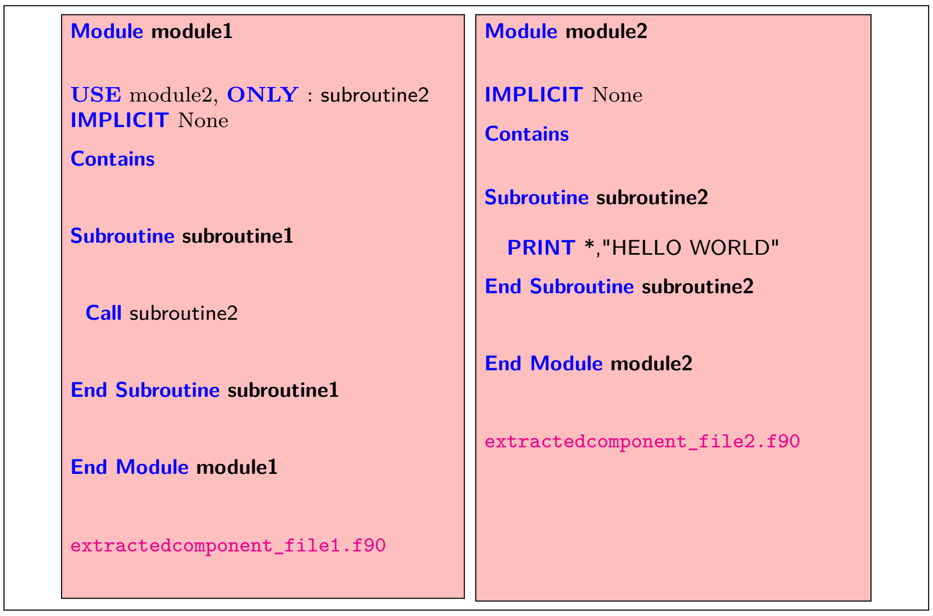
**Extracted Information of Component**

| File | | Module Namespace | | Subprogram Namespace | |
|---|---|---|---|---|---|
| non-shared | shared | non-shared | shared | non-shared | shared |
| X | X | X | X | — | — |

**Extracted Information of the carvedout program**

| File | | Module Namespace | | Subprogram Namespace | |
|---|---|---|---|---|---|
| non-shared | shared | non-shared | shared | non-shared | shared |
| X | X | — | X | X | — |

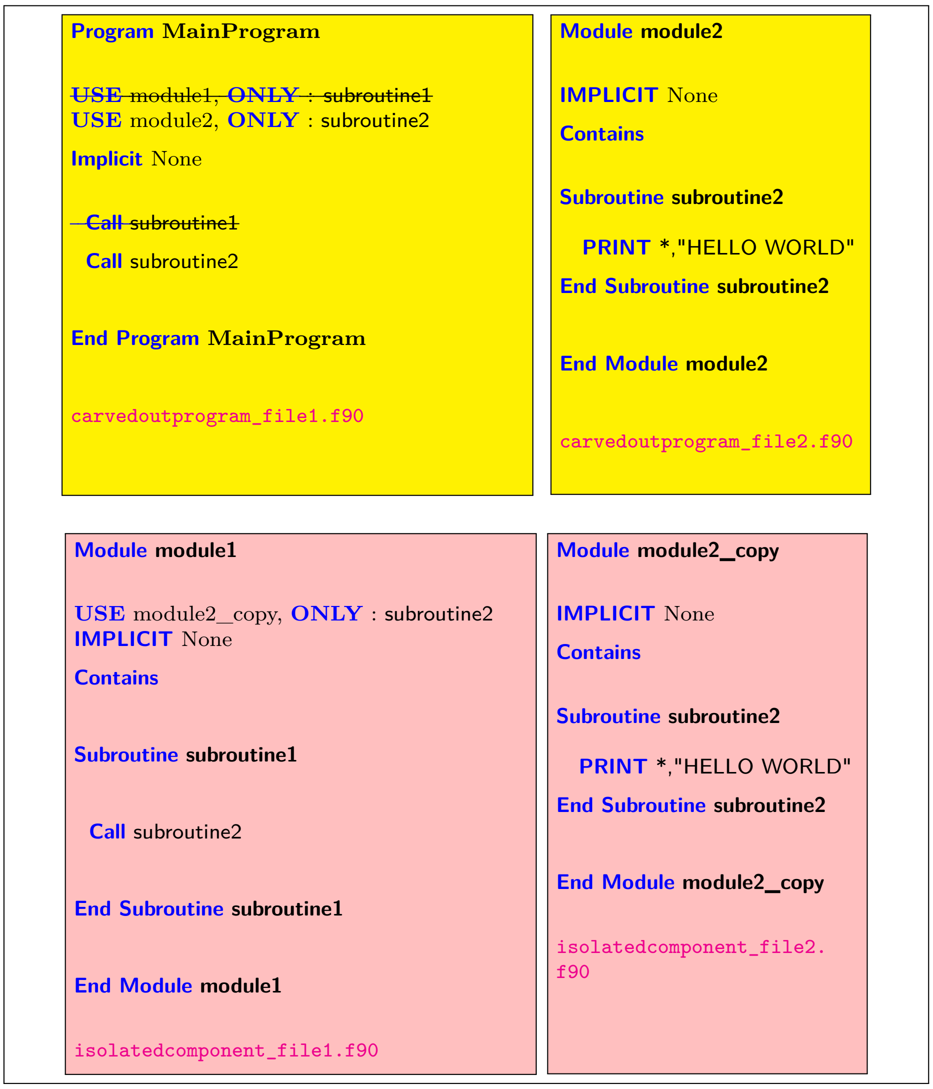
**Figure 6.44.:** (Example 4: dependency information) Applying Component Isolation to the Fortran program in Figure 6.43 and extracting the information about the namespaces of the program and component.



```
Module module1

USE module2, ONLY : subroutine2
IMPLICIT None
Contains


Subroutine subroutine1


  Call subroutine2


End Subroutine subroutine1


End Module module1


extractedcomponent_file1.f90
```

```
Module module2

IMPLICIT None
Contains

Subroutine subroutine2

  PRINT *,"HELLO WORLD"
End Subroutine subroutine2


End Module module2


extractedcomponent_file2.f90
```

**Figure 6.45.:** (Example 4: extracted component) The extracted component generated from the Fortran program in Figure 6.43 by the procedure of Component Extraction.



```
Program MainProgram

~~USE module1, ONLY : subroutine1~~
USE module2, ONLY : subroutine2
Implicit None

 ~~Call subroutine1~~
  Call subroutine2


End Program MainProgram


carvedoutprogram_file1.f90
```

```
Module module2

IMPLICIT None
Contains

Subroutine subroutine2

  PRINT *,"HELLO WORLD"
End Subroutine subroutine2


End Module module2


carvedoutprogram_file2.f90
```

```
Module module1

USE module2_copy, ONLY : subroutine2
IMPLICIT None
Contains


Subroutine subroutine1


  Call subroutine2


End Subroutine subroutine1


End Module module1


isolatedcomponent_file1.f90
```

```
Module module2_copy

IMPLICIT None
Contains

Subroutine subroutine2

  PRINT *,"HELLO WORLD"
End Subroutine subroutine2


End Module module2_copy

isolatedcomponent_file2.
f90
```

**Figure 6.46.:** (Example 4: isolated component and carvedout program) The isolated component (in pink) and carvedout program (in yellow) generated from the Fortran program in Figure 6.43 by Component Isolation.

## 6.7. Extracting implicit coupling fields

In a Fortran program containing a component, there is usually a data flow between the component and the other parts of the program. Such a data flow can be implemented explicitly or implicitly. An explicit data flow (in a shared address space) can be implemented through standard parameters of the entry points of the component. However, an implicit data flow is implemented through the global variables that are shared between the component and the other parts of the program. These shared variables are denoted by *implicit coupling fields*, which end up in the shared modules namespaces. Extracting the implicit coupling fields is the last task to achieve the general goal of this dissertation.

Although shared modules contain the implicit coupling fields, they also contain variables of other types. This hinders an easy extraction of the shared variables. Every shared module namespace in the program may contain the following types of variables:

- shared variables (between the component and the other parts of the program)
- non-shared variables (dedicated either to the component or to the other parts of the program, but not both).
- dead variables (neither used by the component nor by the program)

If non-shared and dead variables are removed from shared module namespaces, the remaining variables are shared between the component and the other parts of the program. We already know that the isolated component and the carvedout program contain a copy of every shared module namespace. Although the copies of each shared namespace end up differently in each slice, both contain the shared variables of the namespace. In addition, the dead variables are already removed from these copies (during the dead contents removal in Step 3 and Step 4 of Component Isolation). Hence, the set of the shared variables is just the intersection of the available variables defined in both copies. On this account, the following steps are required to detect the shared variables:

- Step 1: collecting the set of variables defined in a shared module namespace in the the carvedout program.
- Step 2: collecting the set of variables defined in the the peer copy of the shared module namespace in the isolated component.
- Step 3: the intersection of the two sets (from Step 1 and 2) gives the coupling fields defined in the target module namespace.
- Step 4: repeating step 1 to 3 for every shared module namespace

### 6.7.1. Step 1

When a shared module namespace is detected through the process of Component Isolation, the original copy of its definition is left in the carvedout program. However, any variable that is dead w.r.t. to the carvedout program will be eliminated from this copy in Step 4 of Component Isolation. As a result, the remaining set of the variables is the same set of the variables that are used by the carvedout program.

### 6.7.2. Step 2

Additionally, a copy of the original namespace is added to the isolated component. However, any variable that is dead w.r.t. to the component will be eliminated from this copy version in Step 3 of the process. As a result, the remaining set of the variables is the same set of the variables that are used by the original component.

### 6.7.3. Step 3

The intersection of the two sets generated in Step 1 and Step 2 is a subset of the implicit coupling fields between the component and the carvedout program.

### 6.7.4. Step 4

To collect the complete list of the implicit coupling fields between the component and the carvedout program, Step 1, Step 2 and Step 3 must be applied to each shared module namespace of the component. At the end, the union of all the sets gives the complete list of the implicit coupling fields between the component and carvedout program of the original Fortran program.

### 6.7.5. Adapting Component Isolation

Figure 6.47 shows how the procedure of Component Isolation has been modified to extract the implicit coupling fields. As shown, Step 5 has been added to enable the procedure to collect the shared variables of a shared namespace in each iteration of the process.

### 6.7.6. Example 5

In this example, as shown in Figure 6.48, the Fortran program of Example 4 has been modified to indicate how dead contents and shared variables are handled when isolating a component. We apply Component Isolation to this program and explain how it differs from the previous example. Like the previous examples, the isolated component is initially an empty space, but the complete source code of the original program is assigned to the carvedout program.

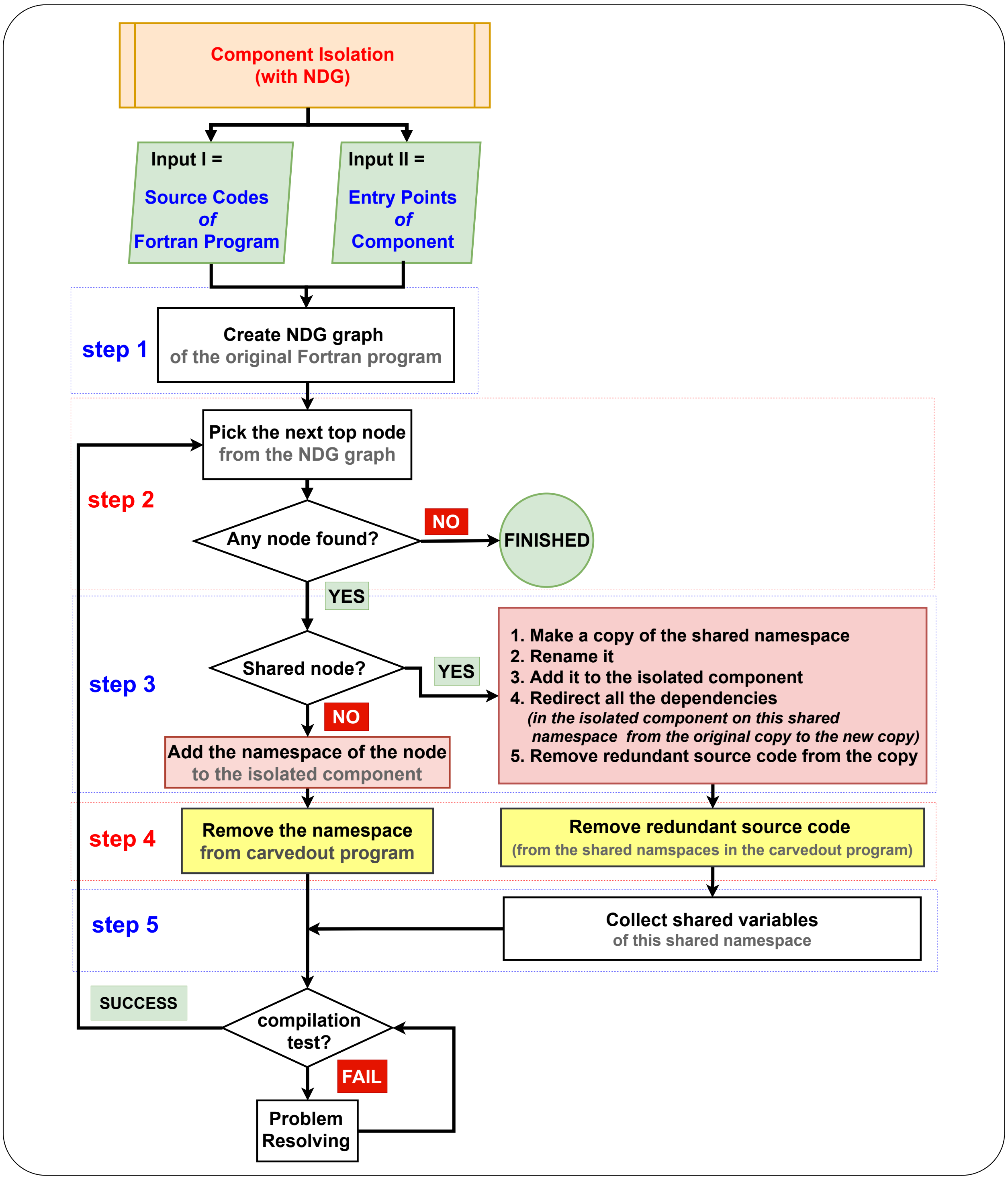

Component Isolation (with NDG)
Input I = Source Codes of Fortran Program
Input II = Entry Points of Component
step 1
Create NDG graph of the original Fortran program
step 2
Pick the next top node from the NDG graph
Any node found?
NO
FINISHED
YES
step 3
Shared node?
YES
NO
1. Make a copy of the shared namespace
2. Rename it
3. Add it to the isolated component
4. Redirect all the dependencies (in the isolated component on this shared namespace from the original copy to the new copy)
5. Remove redundant source code from the copy
Add the namespace of the node to the isolated component
step 4
Remove the namespace from carvedout program
Remove redundant source code (from the shared namspaces in the carvedout program)
step 5
Collect shared variables of this shared namespace
SUCCESS
compilation test?
FAIL
Problem Resolving

**Figure 6.47.:** Extracting the implicit coupling fields between a component and the carvedout program in a Fortran program using the procedure of Component Isolation.

**Step 1: Creating NDG graph** The NDG graph of this example is built similar to Example 4. However, an edge must be added to the graph between the vertices *"MainProgram"* and *"module1"* to show the the dependency of the main program unit on "*var1*". The NDG graph is shown in Figure 6.49.

### 6.7.6.1. Iteration 1

In the first iteration, the following steps are performed:

**Step 2: Picking the next top node** In this step, the graph is traversed, starting from the vertex of the entry point of the component, to pick the top vertex (which is *"module1"*).

**Step 3: Add the namespace to the isolated component** The namespace associated to the top vertex (i.e. *module1*) is a shared namespace as there is an edge from the vertex *"mainprogram"* to *"module1"*. There is, nevertheless, no dead contents with respect to the component in this namespace. Thus, a copy of the namespace is created and renamed to "*module1*_copy" and added to the isolated component in a new file "*isolatedcomponent_file1.f90*".

**Step 4: Remove the namespace from the carvedout program** In contrast to Example 4, the definition of "*module1*" is not removed from the carvedout program since it is a shared namespace. However, dead contents w.r.t. carvedout program must be removed from the namespace. The main program unit is dependent on "*module1*" through "*USE module1, ONLY:: var1, subroutine1*". Since the dependency of the main program unit on the entry point of the component is ignored, the definition of *"subroutine1"* is removed from *"module1"* and the namespace is placed in a new file *"carvedoutprogram_file1.f90"*.

**Step 5: Collect shared variables** In contrast to Example 4, this example contains shared variables. Thus, these variables must be detected and collected in each iteration once a shared namespace is detected. Since *"module1"* is a shared namespace, it may contain some of the shared variables between the component and the carvedout program. To collect the shared variables in this namespace, we must collect the variables that are shared between "*module1*" and "*module1_copy*". A close look shows that "*var1"* must be a shared variable as it exists in both namespaces.

### 6.7.6.2. Iteration 2

**Step 2: Picking the next top node** In this step, the graph is traversed again and the next top vertex (which is *"module2"*) is picked up.

**Step 3: Add the namespace to the isolated component** The namespace associated to the top vertex (i.e. *module2*) is a shared namespace as there is an edge from the vertex *"mainprogram"* to *"module2"*. Thus, a copy of the namespace is created, renamed to "*module2_copy*", and will be added to the isolated component in a new file "*isolatedcomponent_file2.f90*". However, "*var2*" and "*subroutine2*" are dead with respect to the component and, thus, their definitions are removed from the file.

**Step 4: Remove the namespace from the carvedout program** Similar to Example 4, the definition of "*module2*" is not removed from the carvedout program since it is a shared namespace. However, the dead contents with respect to the main program unit must be removed from the namespace. The main program unit is dependent on "*module2*" through "*USE module2, ONLY:: var2, subroutine2*". Thus, the definition of *"subroutine3"* is removed from *"module2"* as dead contents with respect to the main program unit.

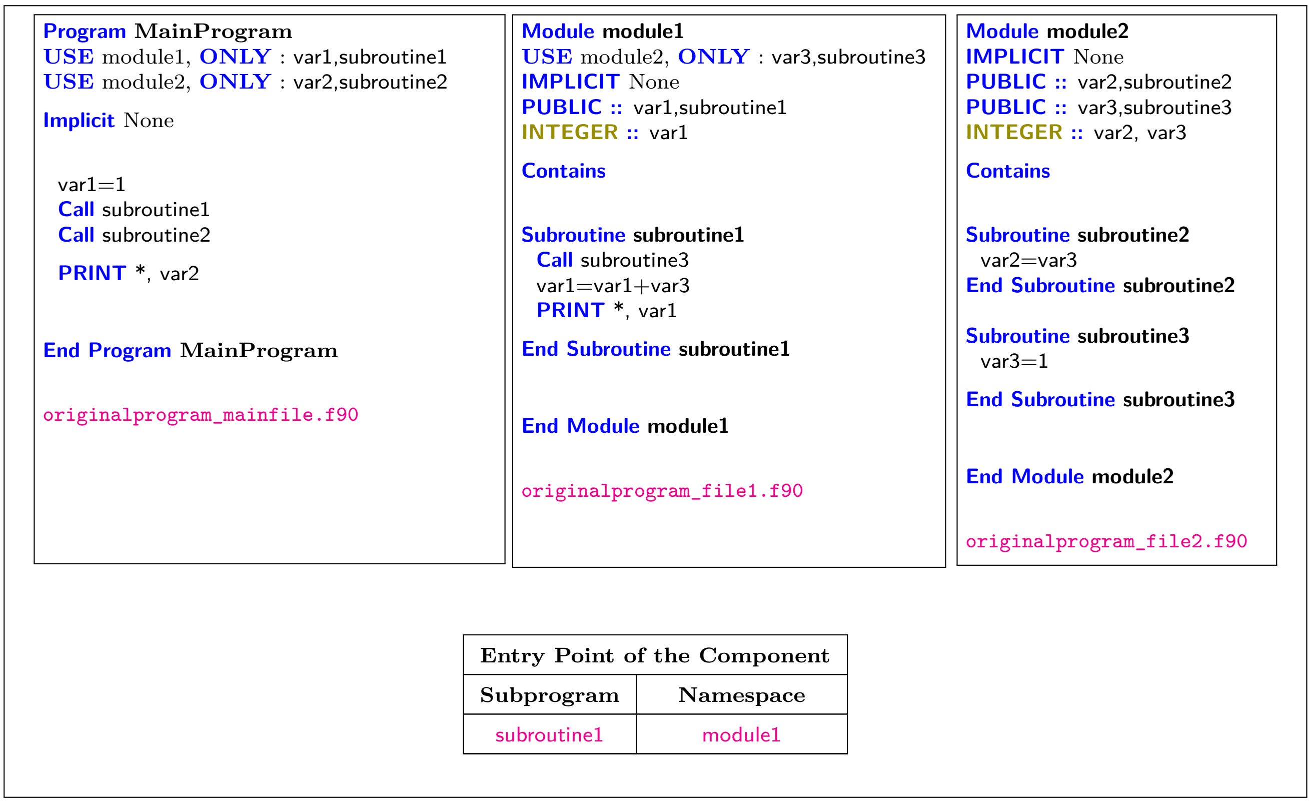
**Step 5: Collect shared variables** Similar to previous iteration, the top namespace is a shared namespace and contains a shared variable. Hence, we collect the variable that is shared between "*module2*" and "*module2_copy*". A close look shows that "*var3*" exists in both namespaces and, thus, it is shared between the component and the carvedout program.

#### 6.7.6.3. Iteration 3

Similar to Example 4, the practice stops in the third iteration.

#### 6.7.6.4. Results

Once the procedure stops, the isolated component and the carvedout program are available. As shown in Figure 6.51, the carvedout program contains "*MainProgram*", "*module1*" and "*module2*", whose definitions have been moved into three new files called *"carvedoutprogram_file1.f90"*, *"carvedoutprogram_file2.f90"* and *"carvedoutprogram_file3.f90"*, respectively. In addition, the dependency of the main program unit on the entry point of the component has been removed from *"carvedoutprogram_file1.f90"* to prevent the compilation failure of the carvedout program. Hence, the call-site (*call subroutine1*) and *"subroutine1"* in the dependency link (*USE module1, ONLY: var1,subroutine1*) are struck through (temporarily) in *"carvedoutprogram_file1.f90"*. Thus, the carvedout program is now expected to compile successfully independent of the isolated component. Note that these statements can be activated again once the isolated component is re-integrated back to the carvedout program to make the new program capable of generating bit-wise identical results to the output of the original program. In addition, the dead contents have been



```
Program MainProgram
USE module1, ONLY : var1,subroutine1
USE module2, ONLY : var2,subroutine2

Implicit None


  var1=1
  Call subroutine1
  Call subroutine2

  PRINT *, var2


End Program MainProgram


originalprogram_mainfile.f90
```

```
Module module1
USE module2, ONLY : var3,subroutine3
IMPLICIT None
PUBLIC :: var1,subroutine1
INTEGER :: var1

Contains


Subroutine subroutine1
  Call subroutine3
  var1=var1+var3
  PRINT *, var1
End Subroutine subroutine1


End Module module1


originalprogram_file1.f90
```

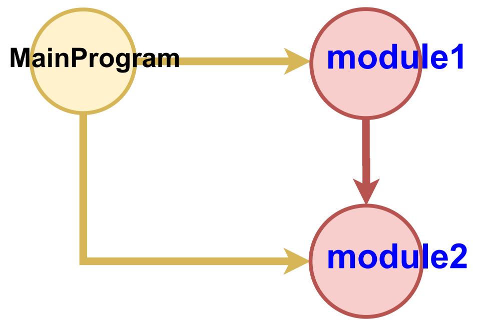
```
Module module2
IMPLICIT None
PUBLIC :: var2,subroutine2
PUBLIC :: var3,subroutine3
INTEGER :: var2, var3

Contains


Subroutine subroutine2
  var2=var3
End Subroutine subroutine2

Subroutine subroutine3
  var3=1

End Subroutine subroutine3


End Module module2


originalprogram_file2.f90
```

| Entry Point of the Component | |
|---|---|
| **Subprogram** | **Namespace** |
| subroutine1 | module1 |

**Figure 6.48.:** (Example 5: Original Program) A Fortran program with one component and three namespaces in the global namespace. The information about the entry point of the component is shown in the table.

removed from *"carvedoutprogram_file2.f90"* and *"carvedoutprogram_file3.f90"* as well by striking through the dead statements.

Additionally, the isolated component contains only "*module1_copy*" and "*module2_copy*" (thus the component has a depth of 2), whose definitions have been placed into new files "*isolatedcomponent_file1.f90*" and "*isolatedcomponent_file2.f90*" to increase the visibility of the component. Note that the definition of *"module1"* and *"module2"* have been renamed to "*module1_copy*" and "*module2_copy*" and placed in two new files *"isolatedcomponent_file1.f90" "isolatedcomponent_file2.f90"*. This should prevent any conflict with the original copies of the two namespaces left in *"carvedoutprogram_file2.f90"* and *"carvedoutprogram_file3.f90"* once the isolated component is re-integrated back to the carvedout program. Accordingly, any reference to "*module2*" in the isolated component has been redirected to *"module2_copy".* Thus, the statement *"USE module2, ONLY: var3, subroutine3"* has been converted to *"USE module2_copy, ONLY: var3,subroutine2"* in *"isolatedcomponent_file1.f90".* Hence, the isolated component is now expected to compile successfully independent of the carved-out program.



**Extracted Information of Component**

| File | | Module Namespace | | Subprogram Namespace | |
|---|---|---|---|---|---|
| non-shared | shared | non-shared | shared | non-shared | shared |
| — | X | — | X | — | — |

**Extracted Information of the carvedout program**

| File | | Module Namespace | | Subprogram Namespace | |
|---|---|---|---|---|---|
| non-shared | shared | non-shared | shared | non-shared | shared |
| X | X | — | X | X | — |

**Figure 6.49.:** (Example 5: dependency information) Applying Component Isolation to the Fortran program in Figure 6.48 and extracting the information about the namespaces of the program and component.

```
Module module1
USE module2, ONLY : var3,subroutine3
IMPLICIT None
PUBLIC :: var1,subroutine1
INTEGER :: var1

Contains


Subroutine subroutine1
  Call subroutine3
  var1=var1+var3
  PRINT *, var1

End Subroutine subroutine1


End Module module1


extractedcomponent_file1.f90
```

```
Module module2
IMPLICIT None
~~PUBLIC :: var2,subroutine2~~
PUBLIC :: var3,subroutine3
INTEGER :: ~~var2,~~ var3

Contains


~~Subroutine subroutine2~~
~~  var2=var3~~
~~End Subroutine subroutine2~~

Subroutine subroutine3
  var3=1

End Subroutine subroutine3


End Module module2


extractedcomponent_file2.f90
```

**Figure 6.50.:** (Example 5: extracted component) The extracted component generated from the Fortran program in Figure 6.48 by the procedure of Component Extraction.

```
Program MainProgram
USE module1, ONLY : var1~~,subroutine1~~
USE module2, ONLY : var2,subroutine2
Implicit None

  var1=1
~~  Call subroutine1~~
  Call subroutine2
  PRINT *, var2

End Program MainProgram
```

carvedoutprogram_file1.f90

```
Module module1
~~USE module2, ONLY : subroutine3,var3~~
IMPLICIT None
PUBLIC :: var1~~,subroutine1~~
INTEGER ::var1
Contains

~~Subroutine subroutine1~~
~~  Call subroutine3~~
~~  var1=var1+var3~~
~~  PRINT *, var1~~
~~End Subroutine subroutine1~~

End Module module1
```

carvedoutprogram_file2.f90

```
Module module2
IMPLICIT None
PUBLIC :: var2,subroutine2
PUBLIC :: var3 ~~, subroutine3~~
INTEGER :: var2, var3
Contains

Subroutine subroutine2
  var2=var3
End Subroutine subroutine2

~~Subroutine subroutine3~~
~~  var3=1~~
~~End Subroutine subroutine3~~

End Module module2
```

carvedoutprogram_file3.f90

```
Module module1_copy
USE module2_copy, ONLY : var3, subroutine3
IMPLICIT None
PUBLIC :: var1,subroutine1
INTEGER :: var1
Contains

Subroutine subroutine1
  Call subroutine3
  var1=var1+var3
  PRINT *, var1
End Subroutine subroutine1

End Module module1_copy
```

isolatedcomponent_file1.f90

```
Module module2_copy
IMPLICIT None
~~PUBLIC :: var2,subroutine2~~
PUBLIC :: var3,subroutine3
INTEGER :: ~~var2,~~ var3
Contains

~~Subroutine subroutine2~~
~~  var2=var3~~
~~End Subroutine subroutine2~~

Subroutine subroutine3
  var3=1
End Subroutine subroutine3

End Module module2_copy
```

isolatedcomponent_file2.f90

**Figure 6.51.:** (Example 5: isolated component and carvedout program) The isolated component (in pink) and carvedout program (in yellow) generated from the Fortran program in Figure 6.48 by Component Isolation.

## 6.8. Chapter Summary

This chapter presents the required procedures to achieve the general goal of this dissertation. It describes a novel static program analysis approach for *extracting a component* from a Fortran program, *isolating a component* in a Fortran program and extracting the shared variables between the component and the other parts of the program.

*Extracting a component* refers to the process of identifying the complete source code of a component in a Fortran program and is performed with a technique denoted as *Component Extraction*. This technique generates only one slice (which contains the complete source code of the component) from the Fortran program. In contrast, *isolating a component* aims at separating the source code of the component from the source code of the other parts of the program. This process is performed with another technique in this dissertation denoted as *Component Isolation*, which generates two slices from the Fortran program called *the isolated component* and *the carvedout program*. The isolated component contains the complete source code of the component and the carvedout program contains the source code of the other pats of the program. Furthermore, the shared variables between the component and the other parts of the program can be extracted with this technique.

Component Isolation guarantees that the isolated component and the carvedout program share no source code. This goal is achieved by renaming those Fortran modules and subprograms in the isolated component that are shared with the carvedout program. Hence, it makes it possible to re-integrate the isolated component to the carvedout program within a new Fortran program in pursuit of generating bit-wise identical results to the output of the original program (after creating a consistent memory between both slices).

The approach presented in this chapter benefits from a new program dependency graph called *the NDG graph* that indicates the dependencies between the namespaces of a Fortran program. Using this graph, all the namespaces of a component can be collected from the program and the shared namespaces between the component and other parts of the program will be identified. In addition, dead codes will be removed from the copies of the shared namespaces in the extracted component or the isolated component and the cravedout program.

# 7. Validation and Evaluation

*This chapter describes the validation procedure of the research carried out within this dissertation. Hence, Section 7.1 and Section 7.2 evaluate the achievements regarding the general and primary goals of this dissertation.*

## 7.1. Validation of the general goal

In Chapter 6, we introduced the approach of Component Isolation to achieve the general goal of this dissertation, which is to extract the source code and shared vaairables of a component from a Fortran program and isolate it in the program. This section describes two techniques to validate this approach. These techniques are as follows:

1. Syntactic validation
2. Semantic validation

### 7.1.1. Syntactic validation

A syntactic validation refers to the use of a static program analysis to examin the slices of a Fortran program (containing a component) generated by Component Isolation in order to determine the success of the procedure. A success is achieved firstly when the isolated component contains all the program statements that the original component depends on, and, secondly, when the carvedout program contains all the program statements that the other parts of the program depend on. Hence, compilation tests can be used in this regard to perform the required analyses as the isolated component and the carvedout program are supposed to compile successfully independent of each other and the original program. Note that all the call-sites to the entry points of the original component are removed from the carvedout program, and, thus, this slice bears no external dependency. Figure 7.1 presents the scheme of a syntactic validation. It is noteworthy that the syntactic validation has already been embedded in the practice of Component Isolation in chapter Chapter 6.

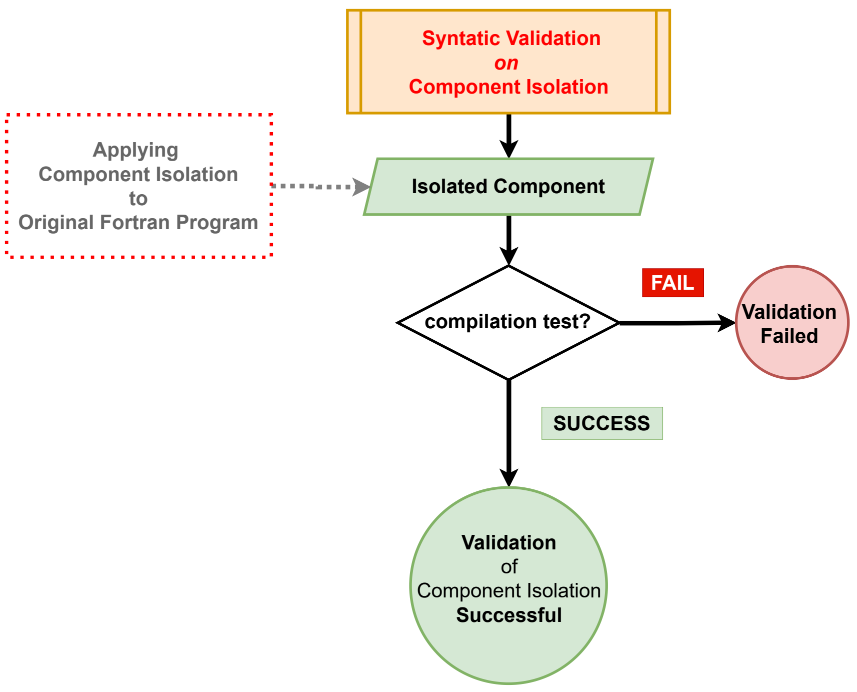


**Figure 7.1.:** A syntactic validation of the practice of Component Isolation.

## 7.1.2. Semantic validation

In the semantic validation, the output of an isolated component and its corresponding carvedout program is examined to see if they generate correct results in comparison to the original Fortran program. The implementation details may, however, render such tests challenging as the output of the original component might not be easily available. In practice, it is advisable to build a new Fortran program composed of the isolated component and carvedout program such that mimicking the functionality of the original program becomes a valid expectation. Thus, the output of the new program is inspected to see if it is bit-wise identical to the the output of the original program, given the same input data. As shown in Figure 7.2, the procedure of the semantic validation is implemented in the following manner:

- re-integrating the isolated component to the carvedout program, which requires re-establishing the (removed) call-sites to the (entry points of the) component in the carvedout program.
- creating memory consistency: since the isolated component and the carvedout program see different copies of the shared variables, this creates memory inconsistency between these two parts inside the new Fortran program. Thus, the same data flow of the original program must be implemented between the isolated component and the carvedout program in the new program. This step, therefore, requires a data flow analysis in the original Fortran program to cap-

ture the data flow specifications between the component and the other parts of the program. In the next step, the memory consistency must be implemented between the isolated component and the carvedout program by implementing the same data flow between these two partitions.

- applying the same input data to the original and new programs and comparing the results to see if they are bit-wise identical.

It should also be noted that a semantic validation already involves a syntactic validation as a successful compilation of an isolated component and a carvedout program is a prerequisite to the successful compilation and execution of the new program.

### 7.1.3. Incremental validation

So far, it was assumed that a syntactic or semantic validation are performed at the end of the process of Component Isolation. However, it is also helpful to apply such a validation during the process in order to facilitate the problem solving procedure. Hence, a validation phase can be added to every iteration of Component Isolation to prevent error propagation to the following iterations.

## 7.2. Validation of Primary Goal 1

Chapter 4 described the implementtion procedure of building a new version of the atmospheric model ECHAM6 with the isolated radiation scheme, which is Primary Goal 1 of this dissertation. The procedure applied Component Isolation to the original source code of the atmospheric model ECHAM6 and generated two slices from the model, namely the isolated radiation component and the carvedout model. In addition, it re-integrated these two slices and built a new version of model that still adopts the sequential radiation scheme of original ECHAM6. Hence, this raises an ample opportunity to use the semantic validation, described in Section 7.1.2, to indicate that the new model can still generate bit-wise identical results to the output of the original model. On this account, we compare the simulation results of the old and new models.

ECHAM6 generates some restart files that store the values of a wide range of the variables of the model that contain the essential simulation data. These variables indicate the state of the model at the end of one course of simulation. The model can continue the simulation in a follow-up run by loading the most recent state of the model from the restart files. These restart files have a NetCDF format (UCAR Community Programs, last access: 19 Mar 2022). To compare the simulation results between the new and original models, we use the Climate Data Operators (CDO)

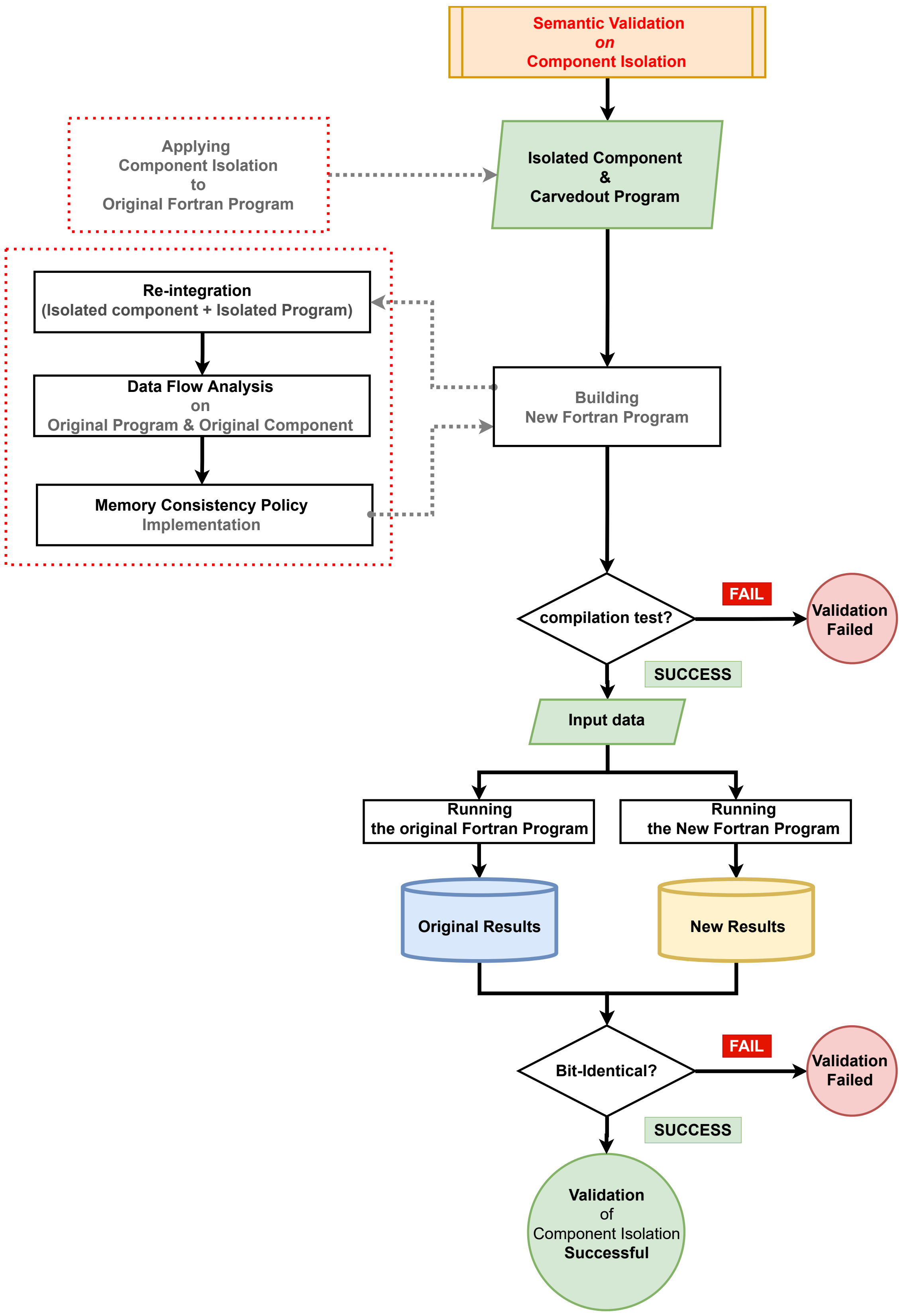


**Figure 7.2.:** A semantic validation of the isolated component.

(Max-Plank-Institute fuer Meteorologie, last access: 17 January 2022), which is a collection of operators for standard processing of climate and forecast model data - including simple statistical and arithmetic functions, data selection and subsampling tools, and spatial interpolation. In addition, it provides support for multiple climate data file formats including NetCDF.

To perform the validation, we compare the following restart files generated by the new model with their peers generated by the original model:

- restart_echam.nc
- restart_accw.nc
- restart_co2.nc
- restart_jsbach.nc
- restart_g3bm.nc
- restart_glm.nc
- restart_spm.nc
- restart_surf.nc
- restart_veg.nc
- restart_yasso.nc

The comparison is conducted using the *cdo diff* command, which compares the contents of two datasets field by field. The input datasets need to have the same structure and their fields need to have the same header information and dimensions. Although this command provides useful detailed reports for climate scientists once the datasets are different, our semantic validation is only interested in the success or failure of the test. Exit status is 0 if datasets are the same and 1 if they differ. Figure 7.3 shows the results of applying a *cdo diff* to the above-mentioned restart files of ECHAM6. The figure shows that the command compares the new and the original model on 27 millions of values (from almost 1300 variables) over two time steps and reports no difference.

This semantic validation proves that the procedure of building a new model with the isolated radiation component has been conducted successfully. In other words, it shows that the procedure of Component Isolation has been performed successfully on the atmospheric model ECHAM6 and implies that the source code of the radiation component and its shared variables have also been extracted correctly from the original model. In addition, the validation results show that the integration of the isolated radiation component to the carvedout model has ended up in a new model with the same functionality of the original model, thus making it capable of generating bit-wise identical results to the output of the original model.

```
$ cdo diff   original/restart_co2.nc     isolated/restart_co2.nc
cdo diff: Processed 138432 Values From 34 Variables Over 2 Timesteps [0.04s 11MB]

$ cdo diff   original/restart_accw.nc     isolated/restart_accw.nc
cdo diff: Processed 24912 Values From 20 Variables Over 2 Timesteps [0.01s 10MB]

$ cdo diff   original/restart_Echam.nc     isolated/restart_Echam.nc
cdo diff: Processed 15657984 Values From 452 Variables Over 2 Timesteps [0.20s 92MB]

$ cdo diff   original/restart_g3bm.nc     isolated/restart_g3bm.nc
cdo diff: Processed 783552 Values From 82 Variables Over 2 Timesteps [0.07s 19MB]

$ cdo diff   original/restart_glm.nc     isolated/restart_glm.nc
cdo diff: Processed 1299648 Values From 10 Variables Over 2 Timesteps [0.06s 25MB]

$ cdo diff   original/restart_jsbach.nc     isolated/restart_jsbach.nc
cdo diff: Processed 3952248 Values From 222 Variables Over 2 Timesteps [0.12s 56MB]

$ cdo diff   original/restart_spm.nc     isolated/restart_spm.nc
cdo diff: Processed 300096 Values From 12 Variables Over 2 Timesteps [0.07s 13MB]

$ cdo diff   original/restart_surf.nc     isolated/restart_surf.nc
cdo diff: Processed 2156736 Values From 472 Variables Over 2 Timesteps [0.13s 36MB]

$ cdo diff   original/restart_veg.nc     isolated/restart_veg.nc
cdo diff: Processed 1959252 Values From 172 Variables Over 2 Timesteps [0.08s 33MB]

$ cdo diff   original/restart_yasso.nc     isolated/restart_yasso.nc
cdo diff: Processed 1236192 Values From 84 Variables Over 2 Timesteps [0.07s 24MB]
```

**Figure 7.3.:** Using the *cdo diff* command to compare the contents of the restart files generated by the new ECHAM6 (built by integrating the isolated radiation component to the carvedout model) with their counterparts generated by the original ECHAM6. The test is performed on 27 millions of values (from almost 1300 variables) over two time steps. If the exit status of the command is 0 (as it is the case in this figure), it shows that the restart files are the same - implying that the two models generate bit-wise identical results. If the restart files were different, the command would generate an exit status of 1 and report on the variables that have different values. These tests clearly show that the procedure of Component Isolation has been conducted correctly and the new model composed of the isolated radiation component and the carvedout model generates bit-wise identical results to the output of the original model as no difference between the restart files were detected.

It is noteworthy that the complete procedure of applying Component Isolation to the source code of the classical version of the atmospheric model ECHAM6 was carried out incrementally in more than 1000 iterations. Hence, the syntactic and semantic validations were also applied to each iteration of the process. These results provide solid evidence that Primary Goal 1 and (implicitly) the general goal of this dissertation have already been achieved so far.

## 7.3. Validation of Primary Goal 2

Chapter 3 described the idea of building a new version of the atmospheric model ECHAM6 with the concurrent radiation scheme, which is Primary Goal 2 of this dissertation. Chapter 4 described the implementtion procedure to achieve this goal, which benefitted from the new version of the model generated for Primary Goal 1 and applied the concurrency scheme to the isolated radiation component. This section describes the process of validating this new version of the model (that adopts the concurrent radiation scheme). A semantic validation is not, however, applicable at this stage since the new version of the model with the concurrent radiation scheme is not designed (by nature) to generate bit-wise identical results to the original model. This approach is, nevertheless, inevitable to improve the computational performance of the original model. On this account, we propose an intermediary step in which a new version of the model with a different radiation scheme will be built.

This intermediary version has a striking similarity to both the sequential and the concurrent radiation schemes. Figure 7.4 shows how the new scheme works with two interesting features. Like the concurrent radiation scheme, it arranges the radiation component on separate MPI processes as shown in Figure 7.4 and implements the same data communication mechanism. In contrast to the concurrent radiation scheme, however, the intermediary version does not solve the radiative transfer and other atmospheric processes simultaneously (concurrently). Instead, it acts like the classical radiation scheme (of the the original atmospheric model ECHAM6) and executes the radiation component sequentially with other atmospheric components (albeit on different MPI processes in a synchronous controlled fashion). Hence, it is said that the intermediary version benefits from *the concurrent* ***synchronous*** *radiation scheme*, which is able to generate bit-wise identical results to the original model. This interesting trait allows for a semantic validation of this scheme. A successful validation implies the following messages:

1. The radiation component has been extracted successfully from the main model.
2. The shared variables of the radiation component has been extracted successfully.
3. The radiation component and the main model have been arranged succesfully on two separate sets of MPI processes.

4. Data communications between the radiation component and the main model has been implemented correctly.
5. Memory consistency has been correctly established between the radiation component and the main model.

To perform the semantic validation, we conduct the restart files of the model that was presented in Section 7.2, except for the fact that this time we compare the simulation results generated by (the model using) the *the concurrent **synchronous** radiation scheme* with the output of the original model.

After the successful implementation of the new version of the atmospheric model with *the concurrent **synchronous** radiation scheme*, we can easily implement the final version of the model with *the concurrent **(asynchronous)** radiation scheme*. Although this last version cannot guarantee bit-wise identical results and does not lend itself to a semantic validation, an independent scientific validation was carried out by a domain expert to indicate that the concurrent radiation scheme is still capable of generating satisfactory simulation results albeit different from the classical. A comprehensive report of the scientific evaluation has been published by (Heidari et al., 2021). This dissertation provided some technical support during the validation process. On this account, we can also conclude that Primary Goal 2 has also been achieved and validated successfully by this dissertation.

Figure 7.5 reflects the validation procedure of the process of building the new version of the atmospheric model ECHAM6 with *the concurrent **(asynchronous)** radiation scheme*. As it shown, the new versions of the model with *the **isolated** radiation component* and *the concurrent **synchronous** radiation scheme* are indeed implemented incrementally. In other words, the radiation component is isolated from the atmospheric model ECHAM6 step by step, and, in each iteration, a synchronous radiation scheme is gradually built. Therefore, the validation of these two phases is performed incrementally. Once the whole component is isolated in the main model, the concurrent radiation scheme will finally be implemented and a scientific validation will be performed. This measure prevents error propagation from one iteration of the process to the others.

## 7.4. Chapter Summary

This chapter evaluates the implementation procedures required to achieve the goals of this dissertation. Concerning the general goal, we presented a syntactic and a semantic validation technique to evaluate the approach of Component Isolation

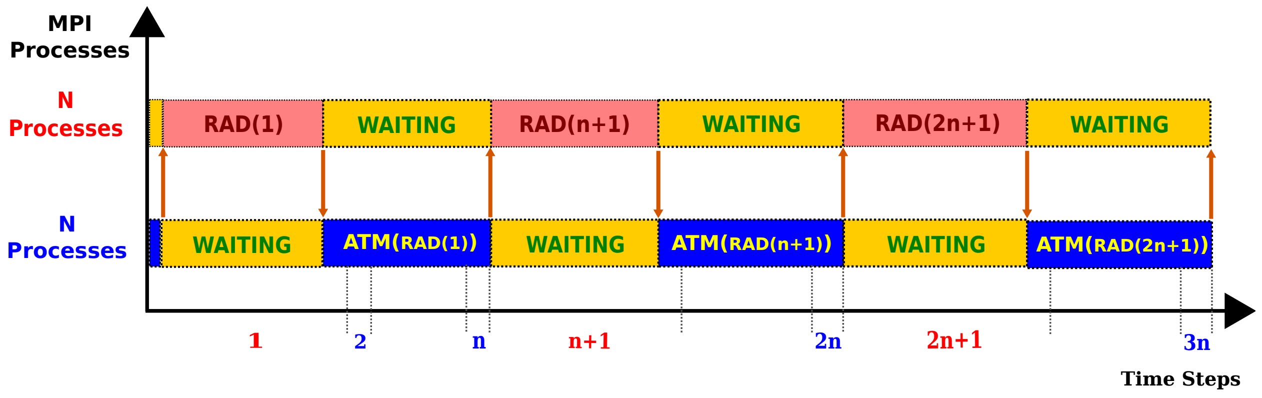


**Figure 7.4.:** A synchronous radiation scheme which generates bit-identical results to the classical radiation scheme.

described in Chapter 6. The syntactic validation indicates that the slices from a Fortran program by Component Isolation contain all the syntactically required program statements. The semantic validation, however, shows that these slices can still function correctly. The syntactic validation can be applied to the isolated component and carvedout program generated using a static program analysis (of a Fortran compiler, for example). To implement a semantic validation, the isolated component and the carvedout program are re-integrated again and a consistent memory is created in order to make the new version of the Fortran program capable of generating bit-wise identical results to the output of the original.

The procedure to achieve Primary Goal 1 takes advantage of Component Isolation and applies it to the radiation component of the atmospheric model ECHAM6. Thus, it generates two slices from the model, namely *the isolated radiation component* and *the carvedout model*. Hence, the validation of this procedure follows the same steps described above to validate Component Isolation. In other words, a syntactic and a semantic validation can show that the isolated radiation component and the carvedout model have been generated correctly and the new version of ECHAM6 that is built by re-integrating these two slices can generate bit-wise identical results to the original model's output. On this account, it can be concluded that the implementation procedure of Primary Goal 1 is also validated. This shows the general goal and Primary Goal 1 have been achieved.

Primary Goal 2 is, however, in pursuit of replacing the classical radiation scheme in ECHAM6 with the concurrent radiation scheme. Since the new scheme resolves the radiative transfer asynchronously in parallel with other atmospheric processes, it cannot thus generate bit-wise identical results to the original model by nature. Hence, no semantic validation is applicable at this stage. However, we introduced *the concurrent* ***synchronous*** *radiation scheme* in which we expect that the radiation component and the main model run sequentially with respect to each other as the classical scheme, but on separate sets of MPI processes as the concurrent ra-

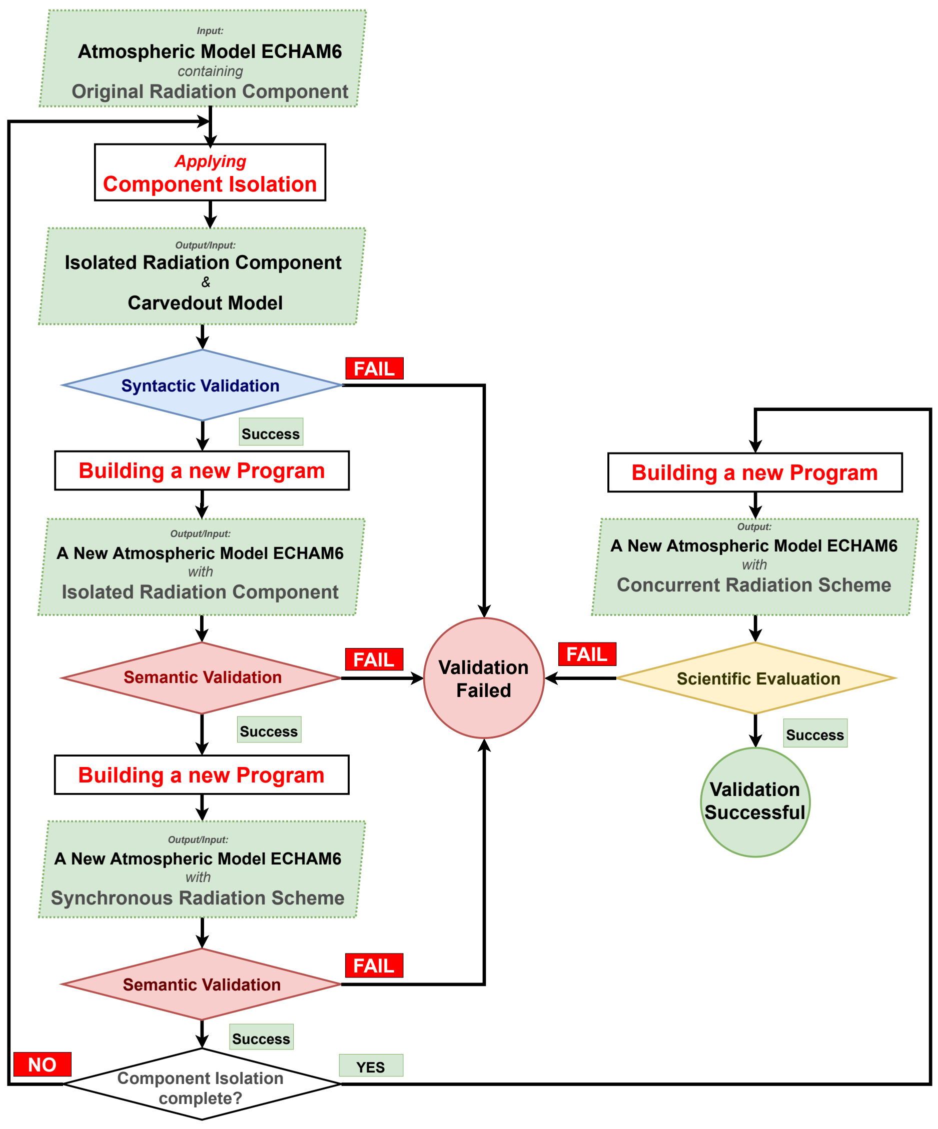


**Figure 7.5.:** The validation procedure of the concurrent radiation scheme in the atmospheric model ECHAM. A synchronous radiation scheme is built as an intermediary phase to increase the validation accuracy. Thus, the procedure includes a syntactic validation of the isolated radiation component and carvedout model, a semantic validation of a new model composed of the two slices and a semantic validation of the synchronous scheme. In practice, the procedure is performed incrementally in each iteration of isolating the component from the main model. Once the component is completely isolated from the model, the concurrent radiation scheme is being built and a scientific validation is applied.

diation scheme. Consequently, the main model starts running once the calculation of radiative transfer is complete and the results are sent back to the main model. Hence, it is expected that ECHAM6 with the synchronous radiation scheme generates bit-wise identical results to the original model and thus a semantic validation is applicable. As a result, it is possible to conclude that the concurrency and the data communication between the radiation component and the main model have also been implemented correctly in *the concurrent* ***asynchronous*** *radiation scheme.*

In addition, the new versions of the model with *the* ***isolated*** *radiation component* and *the concurrent* ***synchronous*** *radiation scheme* were implemented incrementally. Thus, the intended semantic validations were also performed in each step to prevent error propagation from one step to the other. Furthermore, a rigorous scientific validation was carried out by domain scientists to show that the simulation results from the concurrent radiation scheme are still satisfactory. The report of this evaluation is available as a separate peer-reviewed publication from this dissertation. On this account, we conclude that the implementation procedure of Primary Goal 2 is also validated and the goal has been achieved.

# 8. Summary and Conclusion

*This chapter provides a summary of the research conducted within this dissertation. It starts by giving a brief overview of each chapter in Section 8.1, and, then, presents the selected results and conclusions.*

## 8.1. Summary

This dissertation was motivated by the low performance of the atmospheric model ECHAM6 due to the limited number of grid points at the low and coarse-grained spatial resolutions. Hence, improving the performance of the model is opportune for paleoclimate simulations within the PalMod initiative, which benefit from the same settings. Two solutions (including the concurrent and single-precision arithmetic radiation schemes) are, therefore, proposed to improve the scalability of the model. Both solutions are, however, challenged by the unknown code coverage and shared variables of the radiation component as well as its code sharing with other components. Hence, it is argued that a component extraction solution as well as separating the source code of the radiation component from the other parts of the component are prerequisite to the optimization of the model. Such solutions are also beneficial for improving the scalability of the other legacy climate models on heterogenous high performance computing systems. On this account, Primary Goal 1 and 2 of this dissertation are as follows:

- Primary Goal 1: building a new version of the atmospheric model ECHAM6 with *the isolated radiation component.*
- Primary Goal 2: building a new version of the atmospheric model ECHAM6 with *the concurrent radiation scheme.*

By the same toke, the general goal of this dissertation is:

1. Extracting a component from a Fortran program.
2. Extracting the shared variables between the component and the other parts of the program.
3. Isolating the component in the program.

**Background and related works:**

In Chapter 2, three levels of data-parallelism in ECHAM6 are discussed and it is shown how a new level of task-parallelism can be added to the model. This chapter also compares the component extraction and isolation solutions introduced in this dissertation with previous works and argues that none of the existing tools provide a complete solution to the problem in this thesis.

**The concurrent radiation scheme:**

In Chapter 3, it is shown that the radiation component in ECHAM6 traditionally takes up to 58% of the total simulation time at the CR resolution, but it is much more scalable than the main model. However, the sequential component organization of the model constrains this benefit. The concurrent radiation scheme reduces the high computational profile of the component by calculating radiative transfer concurrently with the main model. Additionally, it allows for adopting different domain decomposition and allocating separate MPI processes (from the main model) for the radiation component.

**Implementation of the concurrent radiation scheme:**

Chapter 5 describes the implementation procedures to achieve Primary Goal 1 and 2 of this dissertation. It benefits from the novel approach of Component Isolation (introduced in Chapter 6) to isolate the radiation component in the atmospheric model ECHAM6, which is Primary Goal 1. This version has three important features that will be useful for Primary Goal 2.

1. the extracted shared variables between the radiation component and the other parts of the model.
2. the explicit data exchange between the radiation component and the main model.
3. the consitent memory between the radiation component and the main model.

Thus, we benefit from these features to build another version of the model with the concurrent radiation scheme, which is Primary Goal 2. The new scheme opts for the MPI framework and organizes the main model and the radiation component on two separate sets of MPI processes in order to create a new level of concurrency in ECHAM6. In addition, synchronization is created using a client-server model. The data exchange interfaces of the previous version are augmented with the YAXT library to implement the required data communication between the concurrent radiation component and the main model. The library provides a solution for data redordering between the MPI processes assigned to the radiation component and the main model. This feature allows the radiation component to adopt different domain decomposition from the main model, which can be helpful in load-balancing and creating a consistent model.

**Performance results:**

Chapter 5 presented a detailed performance analysis of the atmospheric model ECHAM6 using the concurrent radiation scheme in comparison to the performance of the model using the classical radiation scheme. It was shown the best performance of the model with the classical radiation scheme is around 550 SYPD (Simulated Years Per Day) if the model is set up to run on 576 MPI processes. In contrast, the concurrent radiation scheme can gain 734 SYPD if both the model and the radiation component individually allocate the same number of 576 MPI processes (i.e. a total of 1152 MPI processes). The same performance is arguably attainable even if the radiative transfer is calculated on a lower number of resources. Nevertheless, this is a clear indication that the new scheme improves both the performance and the scalability of the model.

A further investigation also showed that the concurrent radiation scheme can achieve a maximum speedup of 1.9x over the classical scheme with a minimum parallel efficiency of 80%. In addition, the radiation component demonstrated higher scalability than the main model, a feature that allows for creating physical consistency in the model.

**Component Isolation:**

Chapter 6 presented a novel static program analysis to achieve the general goal of this dissertation. This approach describes a solution for identifying the complete source code of a component in a Fortran program (denoted as *component extraction*), separating the source code of the component from the source code of the other parts of the program (denoted as *component isolation*), and extracting the shared variables between the component and the other parts of the program. This approach was described as *Component Extraction* and *Component Isolation*.

Component Extraction generates one slice from a Fortran program, which is *the extracted component*. In contrast, Component Isolation generates two slices from a Fortran program, which are *the isolated component* and *the carvedout program*. Although the cross-dependenies between these two slices are resolved by replicating the shared source code for each slice, they can be re-integrated to generate a new program capable of generating bit-wise identical results to the output of the original program. Furthermore, Component Isolation can extract the shared variables between the component and the other parts of the program.

**Validation and Evaluation:**

Chapter 7 evaluated the implementation procedures required to achieve the goals of this dissertation. Concerning the general goal, a syntactic and a semantic validation technique were described to evaluate the output of Component Isolation. The syntactic validation indicates whether the the isolated component and the carved-out program (generated from a Fortran program by Component Isolation) contain all the syntactic program statements of the component. The semantic validation, however, shows if these slices can still function correctly.

Concerning Primary Goal 1, a syntactic and a semantic test were applied to *the isolated radiation component* and *the carvedout model* (generated by Component

Isolation from the atmospheric model ECHAM6), and, then, it was shown that ECHAM6 with the isolated radiation component is still capable of generating bit-wise identical results to the output of the original model.

Concerning Primary Goal 2, the validation of ECHAM6 with the concurrent radiation scheme was performed in two phases as a semantic validation was not directly applicable. First, an intermediary version of the model (capable of generating bit-wise identical results to the output of the original model) was built, and, thus, a semantic validation was succesfully applied. Since this new version runs the main model and the radiation component on different sets of MPI processes, the concurrency mechanism and the data communication supports of the concurrent radiation scheme were also validated. Additionally, a rigorous scientific validation on the atmospheric model ECHAM6 with the concurrent radiation schme (conducted by an expert) also showed that the simulation results from the concurrent radiation scheme are still satisfactory.

## 8.2. Conclusion

The new developed concurrent radiation scheme in the atmospheric model ECHAM6 meets Primary Goal 1 and 2 of this dissertation. Radiative transfer is now calculated concurrently with the other atmospheric processes of the model after becoming isolated from the other calculations. Hence, the new scheme allows for scaling ECHAM6 further beyond the old limitations and achieving an unbeatable performance of 734 SYPD that was not attainable by the classical radiation scheme. The new scheme also offers a maximum speedup of 1.9x with a minimum parallel efficiency of 80% across the scaling curve. Additionally, arbitrary domain decomposition shows a promising feature to push the model to much further efficient resource utilization and improve the workload distribution between the MPI processes asigned to the concurent components. This salient feature is expected to eventually decrease the discrepancy between the radiation time step $\Delta t_{rad}$ and normal atmospheric time step $\Delta t_{atm}$ with the objective of creating more physical consistency in the model.

The novel approach of Component Extraction and Isolation described in this thesis also realized the general goal of the model. This method is now capable of extracting a component from a Fortran program, isolating the component fom the other parts in the program and extracting the shared variables between the component and the other parts of program. It was applied to the atmospheric model ECHAM6 and two slices of the model including the isolated radiation component and the carvedout model were generated. In addition, the shared variables between the component and the main model were extracted. These results were directly used in implementing the concurrent radiation scheme in the atmospheric model ECHAM6 in this dissertation. Furthermore, the same results were used in an independent project (Cotronei and Slawig, 2020) to apply single-precision arithmetic to the radiation calculations in ECHAM6. The report shows that the single-precision radiation scheme was accelerated by about 40%.

Component Extraction and Isolation can also be applied to other components in ECHAM6 or to other legacy climate models. This empirical study serves as a successful example that can stimulate research on other concurrent components in climate modeling whenever scalability becomes challenging.

# 9. Future Works

*This chapter describes two proposals for the extensions to the presented works in this dissertation. The proposed ideas augment the novel approach presented in Chapter 6 as well as the concurrent radiation scheme presented in Chapter 3.*

## 9.1. Support Tools for Static Program Analysis

Applying the static program analysis approach described in this dissertation to Fortran programs, especially to large codebases, can become non-trivial without having appropriate support tools around. Hence, we envisage a set of tools to support users in their endeavors to extract or isolate arbitrary components from complex Fortran programs such as legacy climate models. These tools are divided into two different sets as described below.

### 9.1.1. Component Extraction and Isolation Tools

The first set of tools include eight individual tools that are required during the process of Component Extraction and Isolation. These tools are the building blocks of an automation process, but can also be used to conduct the procedures manually as well. Below, we present a brief account of the functionality of these tools.

#### 9.1.1.1. Tool I: Graph Generator

The first required tool that should be designed is a graph generator that must be able to generate two graphs from a Fortran program containing a component: the control flow graph (CFG) and the namespace dependence graph (NDG). The tool takes the source code of the Fortran program as well as the list of entry points of the component as input and generates two data structure representing graphs as output. The NDG graph must clearly show the start node assigned to the only single entry point of the program and all the start nodes assigned to the entry points of the component. Figure 9.1 indicates the block diagram of the graph generator tool.

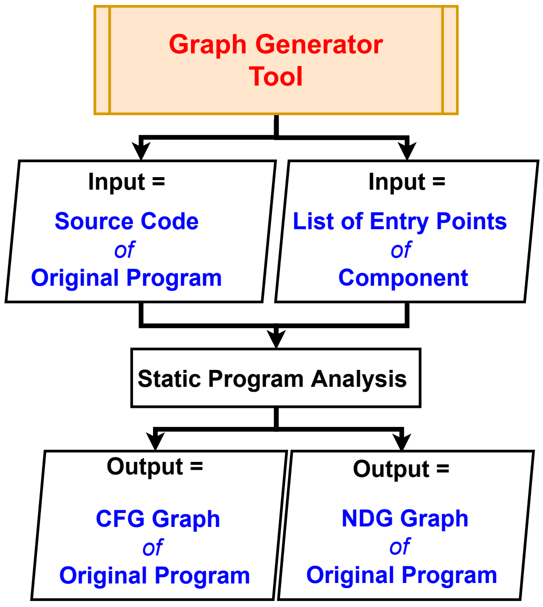

Graph Generator Tool
Input = Source Code of Original Program
Input = List of Entry Points of Component
Static Program Analysis
Output = CFG Graph of Original Program
Output = NDG Graph of Original Program

**Figure 9.1.:** A proposed tool for generating the CFG and NDG graphs of a Fortran program that contains a component. This tool is called a graph generator.

#### 9.1.1.2. Tool II: Namespace Collector

The second tool is a namespace collector that collects the namespaces of a Fortran program that contains the source code of a component and stores them into a set of new Fortran files dedicated to the extracted component. It also collects the namespaces of the program that contain the source code of the corresponding carvedout program and stores them into another set of new Fortran files dedicated to the carvedout program. Hence, the tool takes the source code and the NDG graph of the program as input and generates two slices from the original Fortran program. One slice is composed of a set of Fortran files containing the namespaces of the component and the other slice is composed of a set of Fortran files containing the namespaces of the carvedout program. Since the namespaces dedicated to the component may contain some dead codes w.r.t. the component, they contain a superset of the source code of the component. By the same token, the namespaces dedicated to the carvedout program may also contain some dead codes w.r.t. the carvedout program. Thus, they contain a superset of the source code of the carvedout program. Figure 9.2 indicates the block diagram of the namespace collector tool.

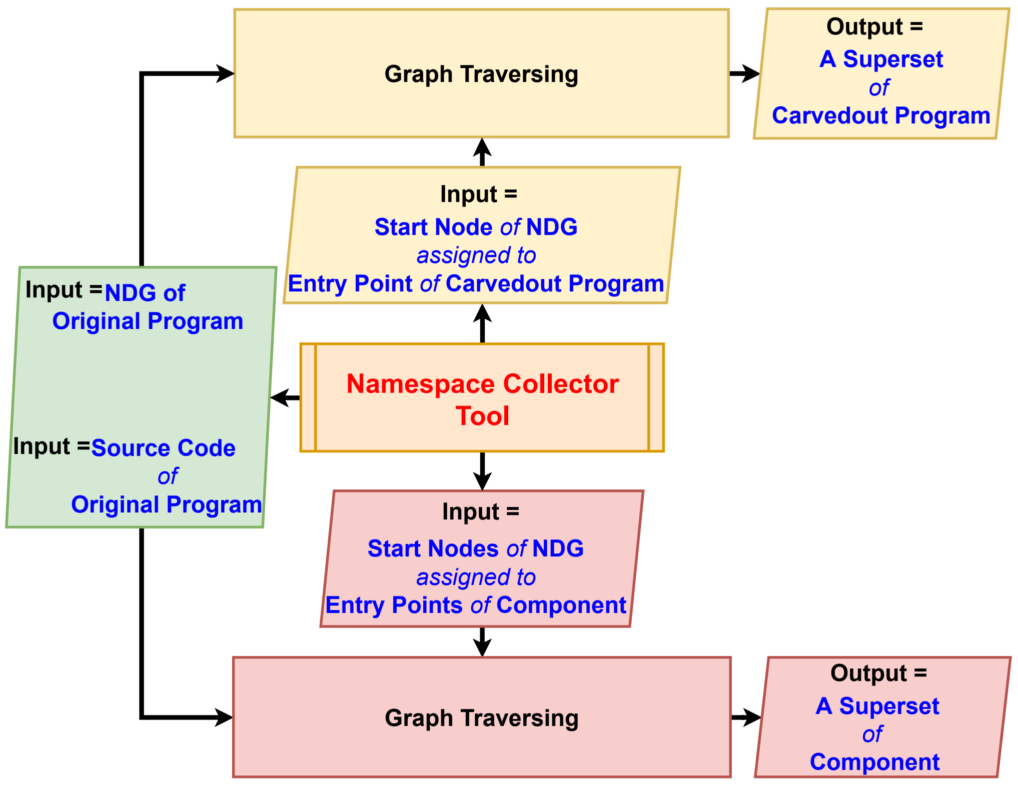

Graph Traversing
Output = A Superset of Carvedout Program
Input = Start Node of NDG assigned to Entry Point of Carvedout Program
Input = NDG of Original Program
Input = Source Code of Original Program
Namespace Collector Tool
Input = Start Nodes of NDG assigned to Entry Points of Component
Graph Traversing
Output = A Superset of Component

**Figure 9.2.:** A proposed tool for collecting the namespaces of a component from a Fortran program and/or the namespaces of its corresponding carvedout program. This tool generates a superset of the component and a superset of the carvedout program. This tool is called a namespace collector.

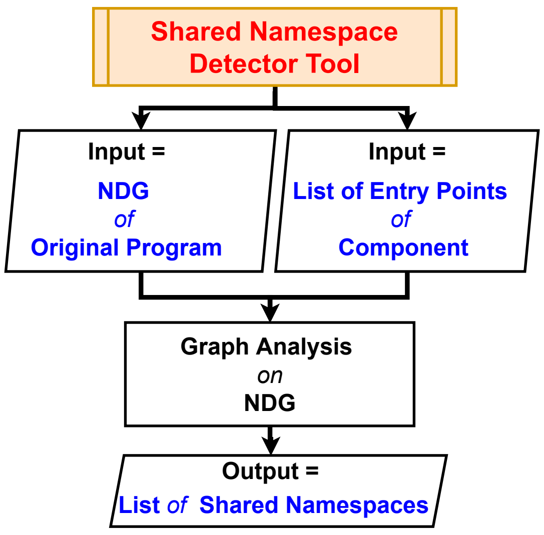

Shared Namespace Detector Tool
Input = NDG of Original Program
Input = List of Entry Points of Component
Graph Analysis on NDG
Output = List of Shared Namespaces

**Figure 9.3.:** A proposed tool for detecting the shared namespaces between a component (in a Fortran program) and its corresponding carvedout program. This tool is called a shared namespace detector.

#### 9.1.1.3. Tool III: Shared Namespace Detector

The third tool is a shared namespace detector. This tool finds the namespaces shared between a component of a Fortran program and the corresponding carvedout program. It traverses the NDG graph of the program and finds the nodes reachable from both the entry point of the program and the entry points of the component. These nodes will indicate the shared namespaces. Thus, this tool takes the NDG graph and the entry points of the component as input and generates the list of the shared namespaces. Figure 9.3 indicates the block diagram of the shared namespace detector tool.

#### 9.1.1.4. Tool IV: Dead Source Code Remover

The fourth tool is a dead source code remover which removes the dead source code from the copies of the shared namespaces. Since shared namespaces contain the source code of both the component and the carvedout programs, they may carry some dead source code w.r.t. to either of them - which must be prevented from leaking into the carvedout program or the extracted/isolated component. The fourth tool removes the dead source w.r.t. the component and the carvedout program from the supersets of the component and carvedout program (generated by the namespace collector tool as described in Section 9.1.1.2), respectively. On this account, the third tool requires the list of shared namespaces and the CFG of the original Fortran

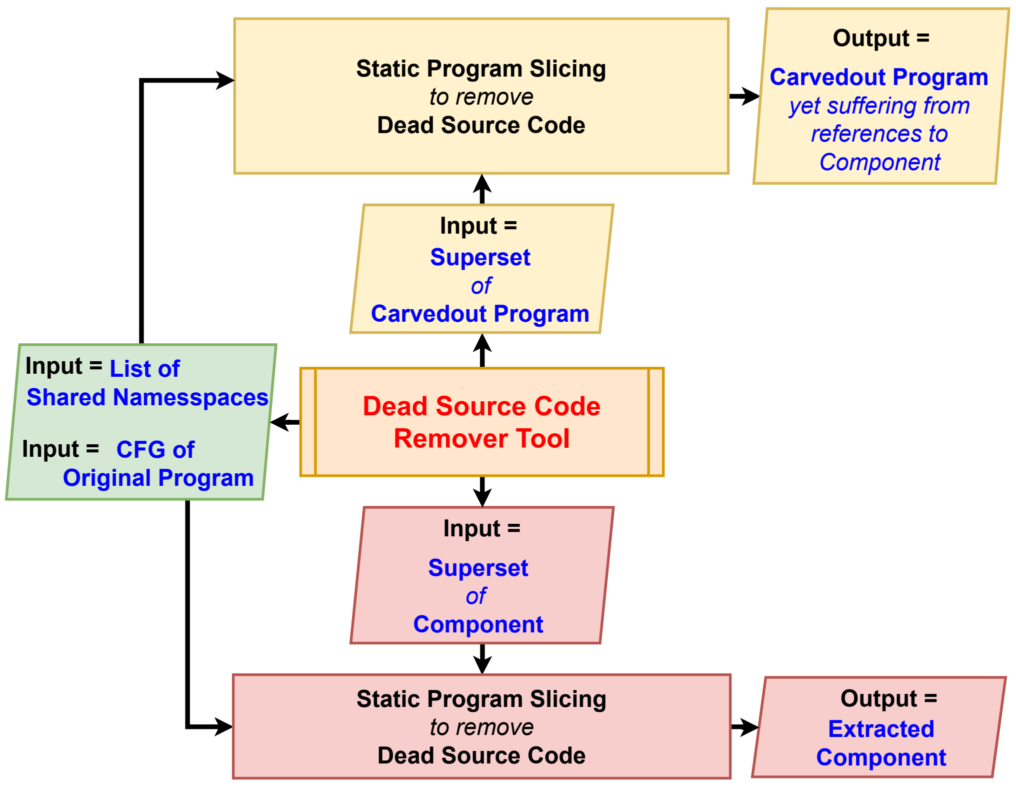


**Figure 9.4.:** A proposed tool for removing the dead source code (w.r.t. a component in a Fortran program) from the superset of the component (generated by the namespace collector tool in Section 9.1.1.2) as well as removing the dead source code (w.r.t. the corresponding carvedout program) from a superset of the carvedout program (generated by the same tool). This tool is called a dead source code remover.

program and generates two slices from the original Fortran program. The first slice is the extracted component and the second slice is almost the carvedout program though it yet contains the call-sites for the entry points of the component (which must be removed by another tool). Figure 9.4 indicates the block diagram of the dead source code remover tool.

#### 9.1.1.5. Tool V: Namespace Separator

The fifth tool is a namespace separator which is responsible for separating the namespaces of a component (in a Fortran program) from the namespaces of its corresponding carvedout program by converting shared namespace to non-shared. To do this, the tool renames the copies of the shared namespace created in the extracted component (which was generated by the fourth tool as described in Section 9.1.1.4) and

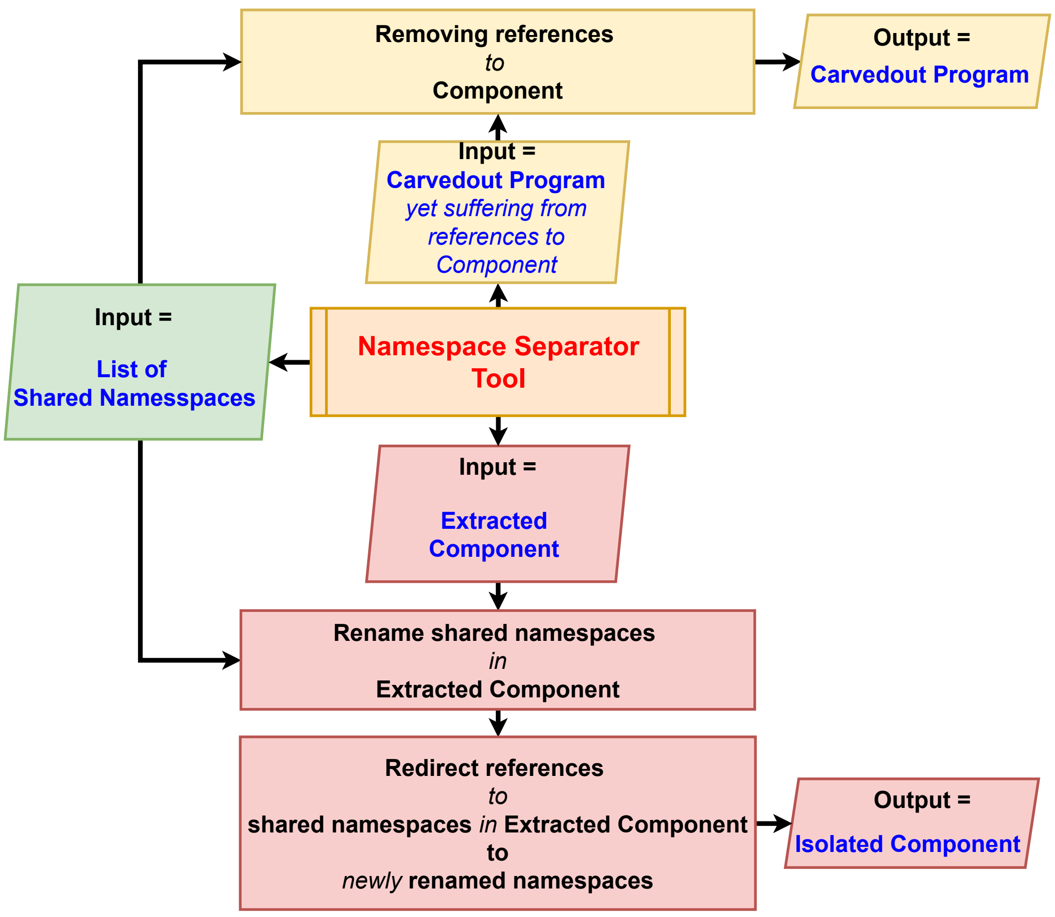


**Figure 9.5.:** A proposed tool for separating the (shared) namespaces of a component (in a Fortran program) from the namespaces of its corresponding carvedout program and generating the isolated component and the carvedout program. This tool is called a namespace separator.

redirects any references to the shared namespaces inside the extracted component to the renamed namespaces, accordingly. On this account, the namespace separator tool takes the list of the shared namespaces and the slices generated by the fourth tool as input and generates the carvedout program and the isolated component. Figure 9.5 indicates the block diagram of the namespace separator tool.

#### 9.1.1.6. Tool VI: Shared Variables Extractor

The sixth tool is a shared variables extractor which is responsible for collecting the shared variables between a component (of a Fortran program) and its corresponding carvedout program. This tool will compare the copy of each shared namespace in

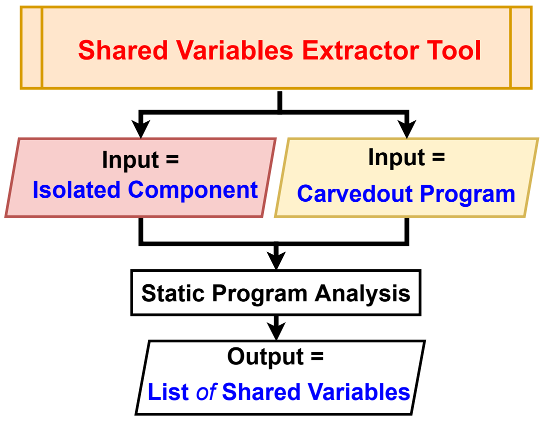

Shared Variables Extractor Tool
Input = Isolated Component
Input = Carvedout Program
Static Program Analysis
Output = List of Shared Variables

**Figure 9.6.:** A proposed tool for extracting the shared variables between a component (of a Fortran program) and its corresponding carvedout program. This tool is called a shared variables extractor.

the isolated component with its peer copy in the carvedout program and collects the same global variables that exist in both. The set of such variables comprises the shared variables between the component and the carvedout program. Figure 9.6 indicates the block diagram of the shared variables extractor tool.

## 9.1.2. Re-integration Tools

In addition to the set of tools described in Section 9.1.1 for extracting and isolating a component from a Fortran program, a couple of more tools will be required for reusing an extracted component in another Fortran program or integrating an isolated component with its corresponding carvedout program. If users should integrate an isolated component and a carvedout program to build a new Fortran program that generates bit-wise identical results to the original program or they decide to reuse the extracted component within another Fortran program to simulate the same behavior of the original component, the tools proposed so far are not enough and more support tools will thus be helpful in this regard. These tools are described in this section.

### 9.1.2.1. Tool VII: Input and output variables detector

The seventh tool is an input and output variables detector that is responsible for detecting which shared variables bring the input data to a component (in a Fortran program) and which variables carry the output data out of the component. To achieve this, an inter-procedural data flow analysis is required to detect the data

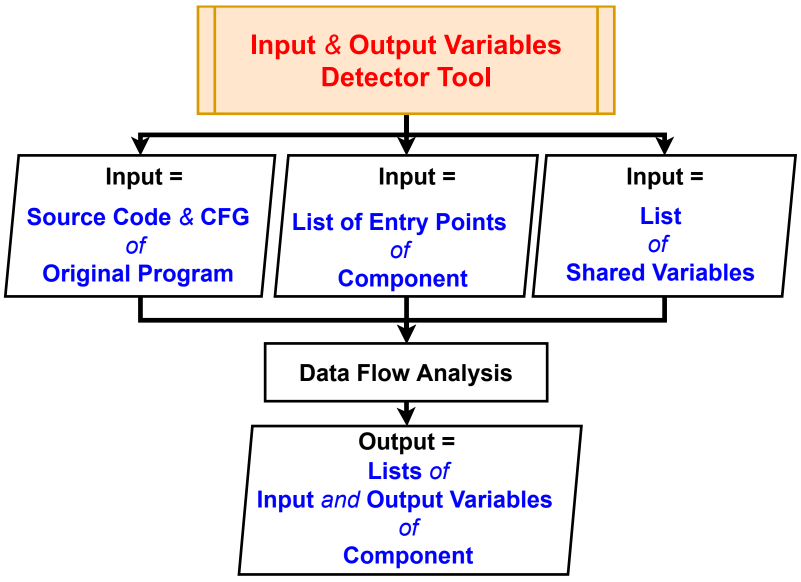


**Figure 9.7.:** A proposed tool for identifying which shared variables bring the input data to a component in a Fortran program and which take the output data from a component to the other parts of the program. This tool is called an input and output variables detector.

dependencies over the shared variables between the component and its corresponding carvedout program. This analysis is performed in the original Fortran program and must find the true data dependencies from the other parts of the program to the component and vice versa. The variables that create true data dependencies from the other parts of the program to the component are the input variables to the component. By the same token, the variables that create true data dependencies from the component to the other parts of the program are the output variables of the component.

A taint analysis can solve this problem. In this approach, the variables generated by one or more sources are propagated through the program and a "leak" is reported whenever one of those variables reaches a so-called sink. Hence, a taint analysis must be instantiated in the entry point of the component to treat any global variables that is shared between the component and he other parts of the original Fortran program. A data-flow solver is then required to automatically propagate these variables through the inter-procedural control-flow graph. Once the solver reaches the end of the entry point, all the program points that access these variables starting from the entry point and throughout the paths along which the data-flow information has been propagated) can be reported.

Figure 9.7 indicates the block diagram of the input and output variables detector tool.

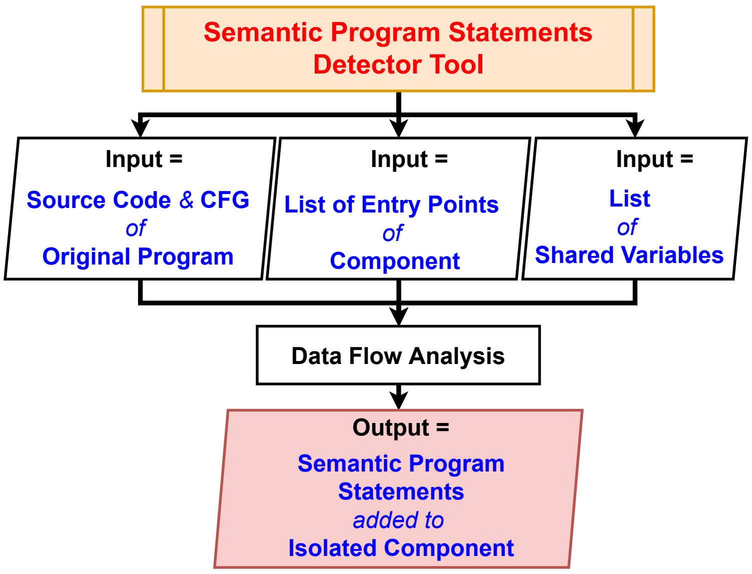

Semantic Program Statements Detector Tool
Input = Source Code & CFG of Original Program
Input = List of Entry Points of Component
Input = List of Shared Variables
Data Flow Analysis
Output = Semantic Program Statements added to Isolated Component

**Figure 9.8.:** A proposed tool for detecting semantic program statements of a component in a Fortran program. This tool is called a semantic program statements detector.

#### 9.1.2.2. Function VIII: Semantic Program Statements Detector

The eighth tool is a semantic program statements detector. This tool is required for reusing an extracted component in another Fortran program or integrating an isolated component to its corresponding carvedout program. The epitome of such statements are the allocation of the dynamic shared variables that are not captured by the static program analysis method described in Chapter 6. The semantic program statements detector must be able to detect such statements and add them to the extracted or isolated component. This tool can benefit from a data flow analysis. Figure 9.8 indicates the block diagram of the semantic program statements detector tool.

### 9.1.3. Epilogue

It is an attractive option to build the tools suggested in this section on top of PhASAR (PhASAR webpage, last access: 17 January 2022). As explained in Chapter 2, PhASAR offers different algorithms to compute some properties of a program such as call-graph and data-flow information. These algorithms can provide the building blocks of the support tools. PhASAR becomes a more interesting option as it allows for program analysis based on the LLVM intermediate representation (IR). Thus, the tools built on top of PhASAR can become applicable to any programming language supported by LLVM including C/C++.

LLVM offers a front-end called Flang for compiling Fortran applications. However, Flang is still a work in progress and not yet able to generate LLVM IR codes (Flang webpage, last access: 4 May 2022). There is an alternative Fortran compiler called LFortran, which is a modern open-source (BSD licensed) interactive Fortran compiler built on top of LLVM. LFortran is also in development (alpha stage) and its reliability has not been approved by PhASAR (LFortran official webpage, last access: 4 May 2022).

Furthermore, it is noteworthy that one of the key challenges in building the tools proposed in this section is how to ensure high precision and efficiency in the static program analysis. As known from Turing and Rice, all non-trivial properties of the behavior of programs written in common programming languages are mathematically undecidable (Møller and Schwartzbach, 2012). Reps (2000) shows that the problem of context-sensitive, structure-transmitted data-dependence analysis is undecidable for first-order languages (both functional and imperative). He argues that some questions in the static program analysis often turn out to be undecidable in their most general forms for several independent reasons (for example, it is undecidable whether a given statement is ever executed; it is undecidable whether a given path is ever executed).

Hence, it would be impossible to create a precise algorithm for context-sensitive, structure-transmitted data-dependence analysis. This means that automated reasoning of software's behaviour generally must involve approximation by handling uncertain information. Thus, we need to circumvent undecidability (when building supporting tools for the required static program analysis discussed in this dissertation) by using useful algorithms that compute approximate, but safe, solutions to the target problems. In practice, manual interventions may become inevitable to overcome some uncertainties. For example, function pointers can potentially create undecidable problems for an automatic call-graph computation if the analyzer is unsure to which function these pointers are pointing. In such cases, it might safely assume that all the functions that match the function pointers signature are called. Consequently, it might decide to add all the matching functions to the call-graph, thus ruining the precision of the analysis. In these case, resolving the ambiguity manually may help to improve the precision of the program analysis.

## 9.2. Improving the concurrent radiation scheme using machine learning techniques

The second proposal for future works suggests improving the concurrent radiation scheme by adding a machine learning technique to improve the accuracy of the atmospheric simulations. In the concurrent radiation scheme, the calculation of radiative transfer starts one radiation time step earlier. For example in Figure 3.8, every $n$ time steps are radiation time steps, i.e. $\Delta t_{\text{rad}} = n * \Delta t_{\text{atm}}$. In other words, time steps 1, *n+1* and *2n+1* are radiation time steps 1, 2 and 3, respectively. The calculation of *RAD(n+1)* starts at time step 1 or the calculation of *RAD(2n+1)* starts at time step *n+1*. However, this time-coupling scheme creates an important problem as discussed in Chapter 3. In fact, the calculation of radiative transfer for each radiation time step requires the input data generated by the model at the same radiation time step. For example, calculating *RAD(n+1)* requires the results of the main model at time step *n+1*. Since the concurrent radiation scheme starts calculating *RAD(n+1)* at radiation time step 1 (i.e. one radiation time step earlier), the model cannot provide the expected input. As a result, *RAD(n+1)* is calculated with the model's results generated at radiation time step 1. In other words, the radiation component sees a lagging state of the model for calculating radiative transfer for the next radiation time step. This problem, therefore, affects the accuracy of the model.

To reduce the negative impact of this problem, one solution could be using machine learning techniques for estimating the input data of the following radiation time step based on the model's results at each radiation time step. A schematic of this solution is shown in Figure 9.9. As the figure shows, one machine learning block is placed before calculating radiative transfer to make an estimation of the atmospheric states at the following radiation time step. For example, at time step *n+1*, the calculation of radiative transfer of the next radiation time step (i.e. *RAD(2n+1)*) starts based on the current atmospheric state at time step *n+1*. However, in Figure 9.9, one machine learning block estimates how the actual atmospheric states would look like at time step *2n+1* before *RAD(2n+1)* is calculated.

The proposed machine learning block can fortunately benefit from online learning. In batch learning, a set of training examples is first collected and the machine learning algorithm is trained based on the examples later. In some practical problems, examples arrive sequentially one by one, but the machine leaning algorithm cannot wait for a training session on the entire example set. This is exactly the situation in Figure 9.9. The actual atmospheric state required for calculating radiative transfer is generated sequentially in every radiation time step. However, the machine learning technique in Figure 9.9 must start making estimations from the first radiation time step, and, therefore, it should be trained on the actual atmospheric states arriving one by one. This principle is realized in online algorithms, in which the last arrived example is used for updating parameters of the network. Baysian inference offers an attractive solution to implement an online learning algorithm regarding the estimation of the atmospheric states in the concurrent radiation scheme (Opper, 2021). This dissertation does not go further in describing this proposal, but it is worth mentioning that adding a machine learning block in Figure 9.9 should not put any burden on the actual computation cost of the concurrent radiation scheme. This is because the new scheme can potentially take advantage of the idle time slots of the radiation component (as shown in Figure 3.7) and use the free resources already assigned to the radiation component (as shown in Figure 9.9).

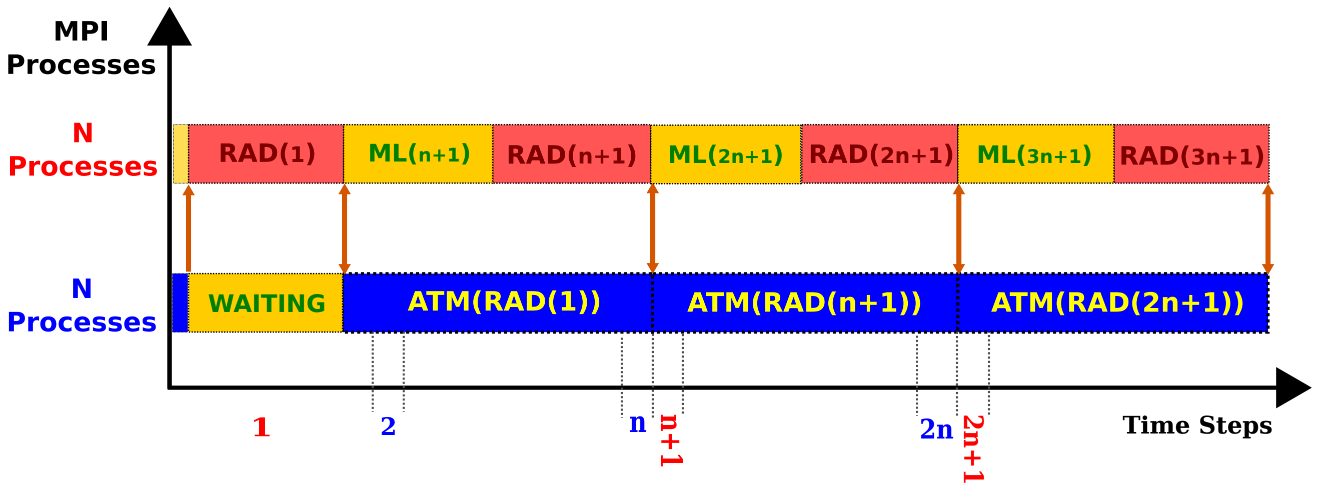


**Figure 9.9.:** Using machine learning techniques to estimate the correct input data for calculating radiative transfer of the following radiation time step.

# Bibliography

Ahmaro, I., Abualkishik, A. M., and Yusoff, M. Z. M.: Taxonomy, definition, approaches, benefits, reusability levels, factors and adaption of software reusability: a review of the research literature, Journal of Applied Sciences, 14, 2396, 2014.

Aida, K. and Casanova, H.: Scheduling mixed-parallel applications with advance reservations, Cluster Computing, 12, 205–220, 2009.

Alexander, K. and Easterbrook, S. M.: The software architecture of climate models: a graphical comparison of CMIP5 and EMICAR5 configurations, Geoscientific Model Development, 8, 1221–1232, 2015.

Alizadeh, O.: Advances and challenges in climate modeling, Climatic Change, 170, 1–26, 2022.

Allen, F. E.: Control flow analysis, ACM Sigplan Notices, 5, 1–19, 1970.

Ampatzoglou, A., Bibi, S., Chatzigeorgiou, A., Avgeriou, P., and Stamelos, I.: Reusability index: A measure for assessing software assets reusability, in: International Conference on Software Reuse, pp. 43–58, Springer, 2018.

Andreas Knuepfer and Others: Score-P - Scalable Performance Measurement Infrastructure for Parallel Codes, URL http://www.score-p.org, 2022.

ArgonneLab webpage: A quick overview of MPI's send modes, URL https://www.mcs.anl.gov/research/projects/mpi/sendmode.html, last access: 17 January 2022.

Asăvoae, I. M., Asăvoae, M., and Riesco, A.: Towards a formal semantics-based technique for interprocedural slicing, in: International Conference on Integrated Formal Methods, pp. 291–306, Springer, 2014.

Bai, J. and Zong, X.: Global solar radiation transfer and its loss in the atmosphere, Applied Sciences, 11, 2651, 2021.

Bal, H. E. and Haines, M.: Approaches for integrating task and data parallelism, IEEE concurrency, 6, 74–84, 1998.

Balaji, V.: Climate computing: the state of play, Computing in Science & Engineering, 17, 9–13, 2015.

Balaji, V., Benson, R., Wyman, B., and Held, I.: Coarse-grained component concurrency in Earth System modeling, 2016.

Baniassad, E. L. and Murphy, G. C.: Conceptual module querying for software reengineering, in: Proceedings of the 20th international conference on Software engineering, pp. 64–73, IEEE, 1998.

Bauer, P., Thorpe, A., and Brunet, G.: The quiet revolution of numerical weather prediction, Nature, 525, 47–55, 2015.

Behrens, J., Hanke, M., and Jahns, T.: How to use MPI communication in highly parallel climate simulations more easily and more efficiently., EGUGA, p. 13569, 2014.

Bonan, G. B. and Doney, S. C.: Climate, ecosystems, and planetary futures: The challenge to predict life in Earth system models, Science, 359, eaam8328, 2018.

Cap, C. H.: JIT99: Java Informations Tage 1999, Springer-Verlag, 2013.

Cleanscape webpage: FortranLint static source code analysis tool, URL https://stellar.cleanscape.net/products/fortranlint/, last access: 17 January 2022.

Codework webpage: Fortran Static Analyzer, URL https://codework.com/solutions/developer-tools/forcheck-fortran-analysis/, last access: 17 January 2022.

Cotronei, A. and Slawig, T.: Single-precision arithmetic in ECHAM radiation reduces runtime and energy consumption, Geoscientific Model Development, 13, 2783–2804, https://doi.org/10.5194/gmd-13-2783-2020, URL https://gmd.copernicus.org/articles/13/2783/2020/, 2020.

Craig, A., Valcke, S., and Coquart, L.: Development and performance of a new version of the OASIS coupler, OASIS3-MCT_3. 0, Geoscientific Model Development, 10, 3297–3308, 2017.

DDM webpage: Domain Decomposition Methods (DDM), URL http://www.ddm.org/, last access: 4 April 2022.

Deutsches Klimarechenzentrum: Topological sorting, URL https://www.dkrz.de/de, last access: 4 May 2022.

Durran, D. R.: Numerical methods for fluid dynamics: With applications to geophysics, vol. 32, Springer Science & Business Media, 2010.

Durran, D. R.: Numerical methods for wave equations in geophysical fluid dynamics, vol. 32, Springer Science & Business Media, 2013.

Egele, M., Scholte, T., Kirda, E., and Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools, ACM computing surveys (CSUR), 44, 1–42, 2008.

Flang webpage: Flang Release Notes, URL https://releases.llvm.org/11.0.0/tools/flang/docs/ReleaseNotes.html, last access: 4 May 2022.

Flato, G., Marotzke, J., Abiodun, B., Braconnot, P., Chou, S. C., Collins, W., Cox, P., Driouech, F., Emori, S., Eyring, V., et al.: Evaluation of climate models, in: Climate change 2013: the physical science basis. Contribution of Working Group I to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change, pp. 741–866, Cambridge University Press, 2014.

Gallagher, K. and Binkley, D.: Program slicing, in: 2008 Frontiers of Software Maintenance, pp. 58–67, IEEE, 2008.

Gholamshahi, S. and Hasheminejad, S. M. H.: Software component identification and selection: A research review, Software: Practice and Experience, 49, 40–69, 2019.

Giorgetta, M. A., Jungclaus, J., Reick, C. H., Legutke, S., Bader, J., Böttinger, M., Brovkin, V., Crueger, T., Esch, M., Fieg, K., et al.: Climate and carbon cycle changes from 1850 to 2100 in MPI-ESM simulations for the Coupled Model Intercomparison Project phase 5, Journal of Advances in Modeling Earth Systems, 5, 572–597, 2013a.

Giorgetta, M. A., Roeckner, E., Mauritsen, T., Bader, J., Crueger, T., Esch, M., Rast, S., Kornblueh, L., Schmidt, H., Kinne, S., et al.: The atmospheric general circulation model ECHAM6-model description, 2013b.

GNU Webpage: GNU Build System, URL https://www.gnu.org, last access: 25 Mar 2022.

Gordon, A., Grace, W., Byron-Scott, R., and Schwerdtfeger, P.: Dynamic meteorology, Routledge, 2016.

GrammaTech webpage: CodeSurferÂ®, URL https://news.grammatech.com/grammatech-releases-codesurfer-1-6-for-c, last access: 22 Mar 2022.

Grossman, D. and Anderson, R. E.: Introducing parallelism and concurrency in the data structures course, in: Proceedings of the 43rd ACM technical symposium on Computer Science Education, pp. 505–510, 2012.

Haslett, J. and Parnell, A.: A simple monotone process with application to radiocarbon-dated depth chronologies, Journal of the Royal Statistical Society: Series C (Applied Statistics), 57, 399–418, 2008.

Heidari, M. R., Song, Z., Degregori, E., Behrens, J., and Bockelmann, H.: Concurrent calculation of radiative transfer in the atmospheric simulation in ECHAM-6.3.05p2, Geoscientific Model Development, 14, 7439–7457, https://doi.org/10.5194/gmd-14-7439-2021, URL https://gmd.copernicus.org/articles/14/7439/2021/, 2021.

Held, I. M.: The gap between simulation and understanding in climate modeling, Bulletin of the American Meteorological Society, 86, 1609–1614, 2005.

Hoffner, T.: Evaluation and comparison of program slicing tools, Citeseer, 1995.

Hogan, R. J. and Bozzo, A.: Mitigating errors in surface temperature forecasts using approximate radiation updates, Journal of Advances in Modeling Earth Systems, 7, 836–853, 2015.

Hogan, R. J. and Hirahara, S.: Effect of solar zenith angle specification in models on mean shortwave fluxes and stratospheric temperatures, Geophysical Research Letters, 43, 482–488, 2016.

Holton, J. R.: An introduction to dynamic meteorology, American Journal of Physics, 41, 752–754, 1973.

Ilyina, T., Six, K. D., Segschneider, J., Maier-Reimer, E., Li, H., and Núñez-Riboni, I.: Global ocean biogeochemistry model HAMOCC: Model architecture and performance as component of the MPI-Earth system model in different CMIP5 experimental realizations, Journal of Advances in Modeling Earth Systems, 5, 287–315, 2013.

Iqbal, M.: An introduction to solar radiation, Elsevier, 2012.

Ito, S.: Semantical equivalence of the control flow graph and the program dependence graph, arXiv preprint arXiv:1803.02976, 2018.

Jones, J.: Abstract syntax tree implementation idioms, in: Proceedings of the 10th conference on pattern languages of programs (plop2003), pp. 1–10, 2003.

Jungclaus, J., Fischer, N., Haak, H., Lohmann, K., Marotzke, J., Matei, D., Mikolajewicz, U., Notz, D., and Von Storch, J.: Characteristics of the ocean simulations in the Max Planck Institute Ocean Model (MPIOM) the ocean component of the MPI-Earth system model, Journal of Advances in Modeling Earth Systems, 5, 422–446, 2013.

Kalkuhl, M. and Wenz, L.: The impact of climate conditions on economic production. Evidence from a global panel of regions, Journal of Environmental Economics and Management, 103, 102 360, https://doi.org/https://doi.org/10.1016/j.jeem.2020.102360, URL https://www.sciencedirect.com/science/article/pii/S0095069620300838, 2020.

Kaur, A. and Nayyar, R.: A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code, Procedia Computer Science, 171, 2023–2029, 2020.

Khaldi, D., Jouvelot, P., Ancourt, C., and Irigoin, F.: Task parallelism and data distribution: An overview of explicit parallel programming languages, in: International Workshop on Languages and Compilers for Parallel Computing, pp. 174–189, Springer, 2012.

Kim, Y., Dennis, J., Kerr, C., Kumar, R. R. P., Simha, A., Baker, A., and Mickelson, S.: KGEN: A Python tool for automated Fortran kernel generation and verification, Procedia Computer Science, 80, 1450–1460, 2016.

Kreowski, H.-J.: Is parallelism already concurrency? Part 1: Derivations in graph grammars, in: International Workshop on Graph Grammars and Their Application to Computer Science, pp. 343–360, Springer, 1986.

Lasslop, G., Moeller, T., D'Onofrio, D., Hantson, S., and Kloster, S.: Tropical climate–vegetation–fire relationships: multivariate evaluation of the land surface model JSBACH, Biogeosciences, 15, 5969–5989, 2018.

Lattner, C.: LLVM and Clang: Next generation compiler technology, in: The BSD conference, vol. 5, 2008.

Lawson, J. R., Potvin, C. K., Skinner, P. S., and Reinhart, A. E.: The vice and virtue of increased horizontal resolution in ensemble forecasts of tornadic thunderstorms in low-CAPE, high-shear environments, Monthly Weather Review, 149, 921–944, 2021.

LFortran official webpage: Lfortran, URL https://lfortran.org/, last access: 4 May 2022.

Li, B., Nychka, D. W., and Ammann, C. M.: The value of multiproxy reconstruction of past climate, Journal of the American Statistical Association, 105, 883–895, 2010.

Lin, Y., Sun, J., Tran, L., Bai, G., Wang, H., and Dong, J.: Break the dead end of dynamic slicing: Localizing data and control omission bug, in: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, pp. 509–519, 2018.

Liu, M.: Inter-Procedural Program Slicing in LLVM, URL https://marc.info/?l=llvm-dev&m=136712747610842&q=p6, 2013.

LLVM webpage: The LLVM Compiler Infrastructure, URL https://llvm.org/, last access: 17 January 2022.

Lopes, B. C. and Auler, R.: Getting started with LLVM core libraries, Packt Publishing Ltd, 2014.

Lyle, J. R. and Wallace, D. R.: Using the unravel program slicing tool to evaluate high integrity software, Technology, 301, 975–3270, 1997.

Makka, S. and Sagar, B.: A New Approach for Optimization of Program Dependence Graph using Finite Automata, Indian Journal of Science & Technology, 9, 2016.

Mann, C. C.: The end of Moore's law?, Technology Review, 103, 42–42, 2000.

Marx, A., Beck, F., and Diehl, S.: Computer-aided extraction of software components, in: 2010 17th Working Conference on Reverse Engineering, pp. 183–192, IEEE, 2010.

Mastroeni, I. and Zanardini, D.: Abstract program slicing: An abstract interpretation-based approach to program slicing, ACM Transactions on Computational Logic (TOCL), 18, 1–58, 2017.

Masud, A. N. and Lisper, B.: Semantic correctness of dependence-based slicing for interprocedural, possibly nonterminating programs, ACM Transactions on Programming Languages and Systems (TOPLAS), 42, 1–56, 2021.

Max-Plank-Institute fuer Meteorologie: The Climate Data Operators (CDO), URL https://code.mpimet.mpg.de/projects/cdo, last access: 17 January 2022.

Meurant, G.: Insect Outbreaks, Elsevier Science, URL https://books.google.de/books?id=v2Hz2blt61IC, 2012.

Møller, A. and Schwartzbach, M. I.: Static program analysis, Notes. Feb, 2012.

Morcrette, J.-J.: On the effects of the temporal and spatial sampling of radiation fields on the ECMWF forecasts and analyses, Monthly weather review, 128, 876–887, 2000.

Morcrette, J.-J., Mozdzynski, G., and Leutbecher, M.: A reduced radiation grid for the ECMWF Integrated Forecasting System, Monthly weather review, 136, 4760–4772, 2008.

MPI official webpage: MPI Forum, URL https://www.mpi-forum.org/, last access: 4 April 2022.

MPICH webpage: MPI Intercommunication, URL https://www.mpich.org/static/docs/v3.3/www3/MPI-Intercomm-create.html, last access: 17 January 2022.

Müller, W. A., Jungclaus, J. H., Mauritsen, T., Baehr, J., Bittner, M., Budich, R., Bunzel, F., Esch, M., Ghosh, R., Haak, H., et al.: A higher-resolution version of the Max Planck institute earth system model (MPI-ESM1. 2-HR), Journal of Advances in Modeling Earth Systems, 10, 1383–1413, 2018.

Myers, D. R.: Solar radiation: practical modeling for renewable energy applications, CRC press, 2017.

Nguyen, H. V., Kästner, C., and Nguyen, T. N.: Cross-language program slicing for dynamic web applications, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 369–380, 2015.

Nielsen, F.: Introduction to MPI: the message passing interface, in: Introduction to HPC with MPI for Data Science, pp. 21–62, Springer, 2016.

N'Takpe, T., Suter, F., and Casanova, H.: A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms, in: Sixth International Symposium on Parallel and Distributed Computing (ISPDC'07), pp. 35–35, https://doi.org/10.1109/ISPDC.2007.1, 2007.

Opper, M.: A Bayesian Approach to Online Learning, Aston University Birmingham, 2021.

Ottenstein, K. J. and Ottenstein, L. M.: The program dependence graph in a software development environment, in: ACM Sigplan Notices, vol. 19, pp. 177–184, ACM, 1984.

PalMod webpage: PalMod official webpage, URL https://www.palmod.de, last access: 17 January 2022.

Parkhomenko, V.: Modeling of global and regional climate response to solar radiation management, in: Journal of Physics: Conference Series, vol. 1141, p. 012057, IOP Publishing, 2018.

Pauluis, O. and Emanuel, K.: Numerical instability resulting from infrequent calculation of radiative heating, Monthly weather review, 132, 673–686, 2004.

PhASAR webpage: An LLVM-based static analysis framework written in C++, URL https://phasar.org/phasar/, last access: 17 January 2022.

Pincus, R. and Stevens, B.: Paths to accuracy for radiation parameterizations in atmospheric models, Journal of Advances in Modeling Earth Systems, 5, 225–233, 2013.

Radulescu, A. and Van Gemund, A. J.: A low-cost approach towards mixed task and data parallel scheduling, in: International Conference on Parallel Processing, 2001., pp. 69–76, IEEE, 2001.

Radulescu, A., Nicolescu, C., Jonker, P. P., et al.: CPR: Mixed task and data parallel scheduling for distributed systems, in: Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001, pp. 9–pp, IEEE, 2001.

Rasp, S.: Statistical methods and machine learning in weather and climate modeling, Ph.D. thesis, lmu, 2019.

Ravitch, T.: How WLLVM works, URL https://github.com/travitch/whole-program-llvm, 2022.

Reps, T.: Undecidability of context-sensitive data-dependence analysis, ACM Transactions on Programming Languages and Systems (TOPLAS), 22, 162–186, 2000.

Roeckner, E., Bäuml, G., Bonaventura, L., Brokopf, R., Esch, M., Giorgetta, M., Hagemann, S., Kirchner, I., Kornblueh, L., Manzini, E., et al.: The atmospheric general circulation model ECHAM 5. PART I: Model description, 2003.

Salby, M. L.: Fundamentals of atmospheric physics, Elsevier, 1996.

Sasirekha, N., Robert, A. E., and Hemalatha, D. M.: Program slicing techniques and its applications, arXiv preprint arXiv:1108.1352, 2011.

Schär, C., Fuhrer, O., Arteaga, A., Ban, N., Charpilloz, C., Di Girolamo, S., Hentgen, L., Hoefler, T., Lapillonne, X., Leutwyler, D., et al.: Kilometer-scale climate models: Prospects and challenges, Bulletin of the American Meteorological Society, 101, E567–E587, 2020.

SciTools: Understand, URL https://www.scitools.com, last access: 17 January 2022.

Shatnawi, A., Mili, H., El Boussaidi, G., Boubaker, A., Guéhéneuc, Y.-G., Moha, N., Privat, J., and Abdellatif, M.: Analyzing program dependencies in java ee applications, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 64–74, IEEE, 2017.

Shipman, G. M.: Programming Models in HPC, Tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2016.

Singh, A. P. and Tomar, P.: Estimation of component reusability through reusability metrics, International Journal of Computer, Electrical, Automation, Control and Information Engineering, 8, 1965–1972, 2014.

Smith, K. and Appelbe, B.: Interactive conversion of sequential to multitasking FORTRAN, in: Proceedings of the 3rd international conference on Supercomputing, pp. 225–234, ACM, 1989.

Smith, K. S.: Pat: an interactive fortran parallelizing assistant tool, 1988.

SourceForge webpage: JSlice Program Slicer, URL http://jslice.sourceforge.net/, last access: 17 January 2022a.

SourceForge webpage: WALA Program Slicer, URL http://wala.sourceforge.net/wiki/index.php/UserGuide:Slicer, last access: 17 January 2022b.

Srinivasan, V. and Reps, T.: An improved algorithm for slicing machine code, ACM SIGPLAN Notices, 51, 378–393, 2016.

Stevens, B., Giorgetta, M., Esch, M., Mauritsen, T., Crueger, T., Rast, S., Salzmann, M., Schmidt, H., Bader, J., Block, K., et al.: Atmospheric component of the MPI-M Earth system model: ECHAM6, Journal of Advances in Modeling Earth Systems, 5, 146–172, 2013.

Stocker, T.: Climate change 2013: the physical science basis: Working Group I contribution to the Fifth assessment report of the Intergovernmental Panel on Climate Change, Cambridge university press, 2014.

Suter, F.: Scheduling ΔCritical Tasks in mixed-parallel applications on a national grid, IEEE, 2007.

Sweeney, J., Salter-Townshend, M., Edwards, T., Buck, C. E., and Parnell, A. C.: Statistical challenges in estimating past climate changes, Wiley Interdisciplinary Reviews: Computational Statistics, 10, e1437, 2018.

Tabari, H., De Troch, R., Giot, O., Hamdi, R., Termonia, P., Saeed, S., Brisson, E., Van Lipzig, N., and Willems, P.: Local impact analysis of climate change on precipitation extremes: are high-resolution climate models needed for realistic simulations?, Hydrology and Earth System Sciences, 20, 3843–3857, 2016.

Thapar, S. S. and Sarangal, H.: Quantifying reusability of software components using hybrid fuzzy analytical hierarchy process (FAHP)-Metrics approach, Applied Soft Computing, 88, 105 997, 2020.

Theis, T. N. and Wong, H.-S. P.: The end of moore's law: A new beginning for information technology, Computing in Science & Engineering, 19, 41–50, 2017.

UCAR Community Programs: Network Common Data Form (NetCDF), URL https://www.unidata.ucar.edu/software/netcdf/, last access: 19 Mar 2022.

Wallace, J. M. and Hobbs, P. V.: Atmospheric science: an introductory survey, vol. 92, Elsevier, 2006.

Wang, D., Post, W., and Wilson, B.: Climate change modeling: Computational opportunities and challenges, Computing in Science & Engineering, 13, 36–42, 2010.

Washington, W. M., Buja, L., and Craig, A.: The computational future for climate and Earth system models: on the path to petaflop and beyond, Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 367, 833–846, 2009.

Washizaki, H. and Fukazawa, Y.: A technique for automatic component extraction from object-oriented programs by refactoring, Science of Computer programming, 56, 99–116, 2005.

WDC-Climate: The source code of the atmospheric model ECHAM6 prepared for performing experiments with both the clasical and concurrent radiation schemes, URL https://doi.org/10.35089/WDCC/SC_PalMod_ECHAM6/, 2022.

Wedi, N. P.: Increasing horizontal resolution in numerical weather prediction and climate simulations: illusion or panacea?, Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 372, 20130 289, 2014.

Weiser, M.: Program slicing, IEEE Transactions on software engineering, pp. 352–357, 1984.

Wikipedia: Flynn's taxonomy, URL https://en.wikipedia.org/wiki/Flynn%27s_taxonomy#cite_note-autogenerated1-16, last access: 17 January 2022.

Wikipedia: Concurrent Computing, URL https://en.wikipedia.org/wiki/Concurrentcomputing, last access: 31 March 2022a.

Wikipedia: Parallel Computing, URL https://en.wikipedia.org/wiki/Parallel_computing, last access: 31 March 2022b.

Wikipedia: SPMD, URL https://en.wikipedia.org/wiki/SPMD, last access: 4 April 2022.

Wikipedia: Topological sorting, URL https://en.wikipedia.org/wiki/Topological_sorting, last access: 4 May 2022.

Wilde, N.: Understanding program dependencies, Citeseer, 1990.

Wilson, R., French, R., Wilson, C., Amarasinghe, S., Anderson, J., Tjiang, S., Liao, S.-W., Tseng, C.-W., Hall, M., Lam, M., et al.: An overview of the SUIF compiler system, Unpublished manuscript, Stanford University, 1995.

Xu, K.-M. and Randall, D. A.: Impact of interactive radiative transfer on the macroscopic behavior of cumulus ensembles. Part I: Radiation parameterization and sensitivity tests, Journal of the atmospheric sciences, 52, 785–799, 1995.

Zenodo Repository: The output results from the experiments performed by the atmospheric model ECHAM6 using both both the clasical and concurrent radiation schemes, URL https://doi.org/10.5281/zenodo.4589140/, last access: 17 January 2022.

# A. List of Publications

Some parts of this dissertation have already been published in the following journal paper:

**Heidari, M. R.**, Song, Z., Degregori, E., Behrens, J., and Bockelmann, H.: Concur- rent calculation of radiative transfer in the atmospheric simulation in ECHAM- 6.3.05p2, Geoscientific Model Development, 14, 7439–7457, https://doi.org/ 10.5194/gmd-14-7439-2021, URL https://gmd.copernicus.org/articles/14/ 7439/2021/, 2021.

# B. Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, wurden als solche kenntlich gemacht.

Diese Versicherung bezieht sich auch auf alle in der Arbeit enthaltenen Grafiken, Zeichnungen und andere bildliche Darstellungen.

Diese Doktorarbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt.

| | |
|---|---|
| Ort, Datum | Unterschrift |