DISSERTATION

Frameworks for Micro- and Nanoelectronics Device Simulation

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik
von

JOSEF WEINBUB

Matr. Nr. 0225909

Wien, im Jänner 2014

Abstract

Approaches for software frameworks, tackling the increasingly challenging tasks of micro- and nanoelectronics device simulations, are investigated. In particular, the developed approaches focus on the key requirements defined to be most important for today's research simulation software, those being reusability, flexibility, usability, maintainability, and expandability.

Research software experiences new challenges primarily due to the fast pacing developments in physical modeling. Simulation tools are typically one step behind the evolution of future devices, in the sense that today's tools have to predict the properties of tomorrow's devices. Research modeling software projects - especially in the area of micro- and nanoelectronics device simulation - attempt to tackle these challenges on their own, thus sacrificing valuable resources for the development of non-modeling related aspects, which introduces a significant loss of synergy effects. In universities, primarily highly specialized simulation tools based on monolithic software design are implemented in a closed-source manner to uphold an advantage over competitors.

In this work, software engineering aspects related to developing frameworks are investigated, particularly focusing to improve the availability of publicly accessible simulation tools relevant to the field of micro- and nanoelectronics device simulation. The advantages of freely accessible simulation source code as well as of decoupling implementations into reusable libraries are elaborated. The developed approaches enable to wrap already available functionality into reusable components.

More concretely, a device simulation framework, a component execution framework, and an interactive simulation framework is investigated. Where a device simulation framework allows to compute the device characteristics, a component execution framework enables to execute a set of components on highly parallel computing targets. Interactive simulation frameworks provide a high-usability access via modular graphical user interfaces. Challenges and requirements are highlighted as well as concrete approaches in form of developed software tools which are freely available under open source licenses. Application examples underline the feasibility of the depicted approaches. The developed frameworks serve as modern and long-term simulation platforms, favoring reusability, flexibility, usability, maintainability, and expandability; all of those aspects are particularly important in the fast developing area of micro- and nanoelectronics device simulation.

Kurzfassung

Herangehensweisen für Softwaregerüste, welche die zunehmend herausfordernden Aufgaben von Mikro- und Nanoelektronischer Bauelemente-Simulation bewältigen, werden untersucht. Im Besonderen fokussieren sich die entwickelten Herangehensweisen auf die Schlüsselanforderungen, die für heutige Forschungssimulationssoftware von höchster Bedeutung sind, wie Wiederverwendbarkeit, Flexibilität, Bedienbarkeit, Wartbarkeit und Erweiterbarkeit.

Forschungssoftware erfährt neue Herausforderungen primär durch die schnell voranschreitenden Entwicklungen in physikalischer Modellierung. Simulationswerkzeuge sind typischerweise einen Schritt hinter der Entstehung von zukünftigen Bauelementen, im Sinne der Vorhersage der Eigenschaften von zukünftigen Bauelementen durch heutige Werkzeuge. Forschungsmodellierungssoftwareprojekte - insbesondere im Bereich der Mikro- und Nanoelektronischen Bauelemente-Simulation - versuchen diese Herausforderungen alleine zu bewältigen, demnach opfern sie wertvolle Ressourcen für die Entwicklung von nicht-modellierungsbezogenen Aspekten, was einen signifikanten Verlust von Synergieeffekten mit sich bringt. An den Universitäten werden primär hochspezialisierte Simulationswerkzeuge, basierend auf monolithischem Softwaredesign und in einer quelltext-geschlossenen Art, implementiert, um einen Vorteil gegenüber Konkurrenten zu haben.

In dieser Arbeit werden Softwareentwicklungsaspekte bezüglich der Entwicklung von Gerüsten untersucht, insbesondere liegt der Fokus auf der Verbesserung der Verfügbarkeit von öffentlich zugänglichen Simulationswerkzeugen, relevant für das Gebiet der Mikro- und Nanoelektronischen Bauelemente-Simulation. Die Vorteile von frei verfügbaren Simulationsquelltexten und der Entkopplung von Implementierungen in wiederverwendbaren Bibliotheken werden ausgearbeitet. Die entwickelten Herangehensweisen ermöglichen die Umhüllung von bereits verfügbaren Funktionalitäten in wiederverwendbare Komponenten.

Im Konkreten werden ein Bauelemente-Simulationsgerüst, ein Komponentenausführungsgerüst und ein Interaktivsimulationsgerüst untersucht. Während ein Bauelemente-Simulationsgerüst die Berechnung von Bauelemente-Characteristika ermöglicht, erlaubt ein Komponentenausführungsgerüst die Ausführung einer Menge von Komponenten auf hochgradig parallelen Berechnungszielen. Interaktiv-Simulationsgerüste stellen einen Hochbedienbarkeitszugang durch modulare grafische Benutzeroberflächen bereit. Herausforderungen und Anforderungen werden behandelt wie auch konkrete Herangehensweisen in der Form von entwickelten Softwarewerkzeugen, welche frei unter quelltext-offenen Lizenzen zugänglich sind. Anwendungsbeispiele unterstreichen die Machbarkeit der aufgezeigten Herangehensweisen. Die entwickelten Gerüste dienen als moderne und langfristige Simulationsplattformen, welche Wiederverwendbarkeit, Flexibilität, Bedienbarkeit, Wartbarkeit und Erweiterbarkeit fördern; all jene Aspekte sind im Speziellen in dem sich schnell entwickelnden Bereich der Mikro- und Nanoelektronischen Bauelemente-Simulation wichtig.

Acknowledgement

First and foremost I want to express my deepest gratitude to my supervisor and mentor Professor Siegfried Selberherr who not only enabled me to embark on this astonishing journey in the first place but also provided me with unwavering support throughout the entire time. Professor Selberherr allowed me to find my own paths, honed my ability to take on responsibilities, and always - and I mean always - found some kind of error in my manuscripts, continuously improving my eye for details. Without his belief in me and my skills I would not be where I am today.

Karl(i) Rupp deserves my sincere thanks as a colleague with whom I had the pleasure working for over three years. In him I found a colleague who always provided me with his honest opinion in a respectful manner. Karli was always willing to venture out on new (rather time-consuming) side-projects, such as the Google Summer of Code programs which we could claim for our university not just for the first time but also three times in a row (so far..).

I want to thank the informal *gentlemen's club* for their support but most of all for great moments and adventures, especially Lado Filipovic (the *Canuck*), Mihail Nedjalkov (Mixi), Philipp Schwaha (Lord Phil), Stanislav Tyaginov (Stas), Alexander Makarov (Alex), Dmitry Osintsev (Dima), Ivan Starkov (Vanja), Stanislav Vitanov (Stani), Roberto Orio (Mr Brazil), and Johann Cervenka (Cerv). I will never forget looking for the sun in Sozopol, the surprisingly challenging task of finding a suitable restaurant in Kyoto, the slightly disturbing *white nights* in Saint Petersburg, the challenging trip to Munich, the campaign in Bratislava, the occasional office sessions, and the visits in my hometown.

I also want to express my gratitude to other members of the institute for their support over the recent years, among them are Professor Erasmus Langer, for providing an excellent work environment, Florian Rudolf, for joining our software team and providing a fresh view (also for proof-reading this thesis), Markus Bina, for advanced device simulation knowledge, Franz Schanovsky, for cluster support, Viktor Sverdlov, for honest discussions, Manfred Katterbauer, for maintaining my hardware and enduring my requests, René Heinzl, for igniting my interest in software engineering, and Franz Stimpfl, for the fun we had.

I had the honour of visiting the Device Modelling Group at the University of Glasgow and EPCC at the University of Edinburgh, Scotland, UK in 2012, which was funded by the *HPC-Europa2* program. For this, I owe Professor Asen Asenov my most sincere thanks as it would not have been possible without him. Also, I am honoured that Professor Asenov acts as a second examiner for this thesis, which in the light of his undoubtedly tight schedule is no small undertaking. Overall, I want to thank the group members in Glasgow, among them are Stanislav Markov, Gordon Stewart, Ewan Towie, Vihar Georgiev, and Campbell Millar, for extensive talks, their interest in my work, their hospitality, and their support. At EPCC, Catherine Inglis did an amazing job of organizing the overall visit whereas Daniel Holmes and especially Mario Antonioletti devoted their valuable time to discuss my research and possible improvements. I learned a lot about software engineering for supercomputers, which in this case was HECToR.

I also want to express my gratitude to a few additional fellow researchers. A big thanks to Václav Hapla, whom I met the first time at a conference in Helsinki and then again by accident at a summer school in New York City. In the end, he invited me to give a lecture at a workshop at his university in Ostrava. Thank you so much for you hospitality and the opportunity. Also, I am thankful for getting to know Jean Michel (JM) Sellier whom I met at a conference in Madison, Wisconsin and since then stayed in touch. JM and I share the need to support the open source movement in the area of technology computer-aided design. I had the pleasure to get to know Peter Gottschling at a conference in Rhodos in 2010, where we had extensive discussions on software engineering, resulting in me realizing that I have a long way ahead of me.

The projects I participated in and the people I had the honour to meet and work with would not have been possible without the projects provided by publicly funded institutions, such as the European research council, the Austrian science fund, and the partnership for advanced computing in Europe as well as ultimately the tax payers. Thank you for enabling young researchers to develop themselves in an astonishingly fast pacing research environment.

My progress is not only influenced by professional connections but also by a highly supporting social base. By that I mean firstly my dear friends who were always there for me and with whom I spent amazing times, allowing me the occasional much required distance from work. Among them are Mathias (LG), Ralph (Extraordinaire), Thomas (Tommi), Susi (Susi!), Dominik (Ewok), Clemens (Friedrich), Karin (Karin!), Florian (Flo), Philipp (Chip), Georg (Bussi Schorsch), Christian (Puxi), Andreas (Nemsi), and Markus (Gotchy). Thanks for the various *oakings*, *BOBs*, gaming sessions, barbecues, and whatnot; I am honoured to call you friends. Secondly, I owe my dear family my deepest gratitude for their unconditional support, most of all my caring mother Margarethe, my unwavering father Josef, my beloved sister Iris (Billi), as well as my extended family around the various *clans* based in Limberg (the *Steinschadens*, the *Kastners*, the *Vetters*, the *Dolezals*, and the *Goldas*), Fahndorf (the *Weinbubs*, the *Hintermayers*, and the *Hangels*), and Tulln (the *Grills* and the *Friedls*). Thirdly and most importantly, I thank my smart, lovely, funny, lively, understanding, and caring girlfriend Christiane for enduring me and my devotion to my work.

Josef Weinbub

on a rainy January 21, 2014

Contents

List of Acronyms

AMR	adaptive mesh refinement
API	application programming interface
BSD	Berkeley source distribution
CBSE	component-based software engineering
cc	cache-coherent
CCA	common component architecture
CPU	central processing unit
CSE	computational science and engineering
CSG	constructive solid geometry
DAG	directed acyclic graph
DAGuE	directed acyclic graph unified environment
DD	drift-diffusion
DDPM	distributed data parallel mode
DSO	dynamic shared object
DTD	document type definition
DTPM	distributed task parallel mode
ESMF	earth system modeling framework
FEA	finite element analysis
FEM	finite element method
FLOSS	free/libre open source software
FVM	finite volume method
GPL	general public license
GPU	graphics processing unit
GUI	graphical user interface
HPC	high performance computing
ID	identification
LCSD	library-centric software design
LGPL	lesser general public license
MIT	Massachusetts Institute of Technology
MNDS	micro- and nanoelectronics device simulation
MOSFET	metal-oxide-semiconductor field-effect transistor
MPI	message passing interface
NUMA	non-uniform memory access
PDE	partial differential equation
SM	serial mode
TCAD	technology computer-aided design
UMA	uniform memory access
VTK	visualization toolkit
XML	extendible markup language
XPath	XML path language

Chapter 1

Introduction

1.1 Micro- and Nanoelectronics Device Simulation

Micro- and nanoelectronics device simulation (MNDS) is a sub-category of technology computer-aided design (TCAD), which plays a crucial role in semiconductor product development, enabling to replace many cost- and time-intensive experiments by computer simulations [1]. TCAD, in general, deals with the modeling of device fabrication[1], device operation[2], and circuit simulation. Where device fabrication deals with, for instance, simulating the doping process of semiconductors, device operation describes the electrical behavior of a device. In turn, circuit simulation models the behavior of an ensemble of devices. Typically, device fabrication provides the input data for the device operation step, being doping profiles as well as a mesh, modeling the simulation domain. The device operation yields electrical characteristics, enabling to setup models required for circuit simulations. Thus, the individual stages of TCAD form a sequence of simulation steps (Figure 1.1).

A peculiarity of TCAD is the fact that the complexity of some problems scales with the currently available computational hardware, used to carry out the simulations. For instance, the design of future central processing units (CPUs) - requiring ever-more complicated modeling and thus increasingly demanding computational resources - is addressed on current generation hardware. This fact puts pressure on the available simulation approaches to provide the necessary means to satisfy the modeling needs. This circumstance is also characterized by the fact that research on device and process modeling in certain technologies, such as silicon carbide, lack the maturity of the silicon technology, although commercial products are already available. Overall, the role of TCAD is fundamentally fueled by the industry's urge to satisfy Moore's Law as well as the *More than Moore* scenario [2], thus continually demanding cutting-edge research in all areas related to TCAD.

Another aspect of MNDS is that due to the ever-ongoing advances in physical modeling, a plethora of highly specialized simulation tools is available. These tools are typically focused on the modeling aspect rather than the software design[3]. For instance, primarily monolithic software designs are applied due to a decreased initial development effort. On the contrary, a different approach would require a separation of functionality into reusable libraries.

[1] Device fabrication is frequently referred to as process simulation.
[2] MNDS, or short *device simulation*, refers to *device operation*.
[3] *Software design* deals with software system abstractions and their relationships [3].

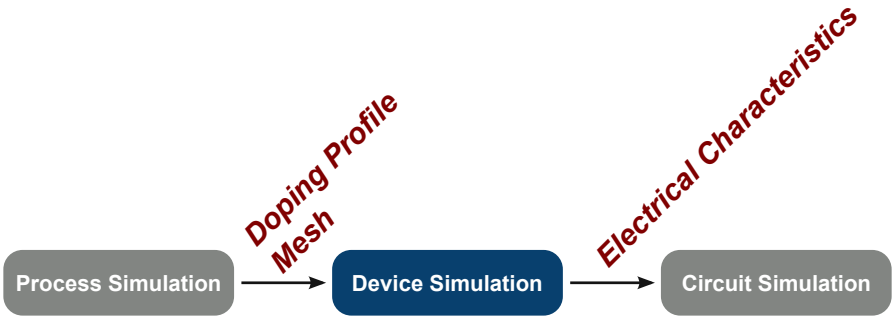


Figure 1.1: The individual simulation steps involved in TCAD. Process simulation provides doping profiles on top of a mesh to the device simulation step. In turn, device simulation computes the electrical characteristics used for circuit simulations.

Such a separation, however, would in turn trigger increased initial development overhead but also would allow the build-up of a reusable set of libraries, enabling minimal-effort applications by merely interfacing with the libraries. However, the emphasis is usually on the exploitation of the developed physical model, as this is significantly more valuable in a scientific sense. The scientific modeling community and, by extension, the research journals tend to concentrate primarily on new modeling aspects rather than the software which enabled the simulation in the first place. With publications representing the actual currency in the academic environment, higher publication or citation counts are attributed much higher importance than the development and maintenance of scientific software. The increasing competition for grants and positions in a strained economic climate puts additional pressure on this process.

Related to the modeling-focus, the simulation software ecosystem in the field of MNDS lacks behind with respect to free/libre open source software (FLOSS) [4][5]. It is common practice that simulation software is either developed and distributed in a commercial or in a closed-source manner. Where the first is primarily performed by companies, the latter is usually applied by the academic MNDS research community. Software is usually seen as an asset, ensuring an advantage over competitors. Where binary versions in principle allow access to the simulation mechanism, it hides the actual implementation. This *black box* approach is especially problematic if researchers need to investigate the implementations, usually essential for development, analysis, and debugging of physical models. Additionally, commercial simulation tools in the field of MNDS are typically expensive, further straining the budget of research institutions. Providing FLOSS-based implementations along with research results in scientific journals, also satisfies the fundamental scientific principle of science, being *reproducibility* [6][7]. Another relevant aspect of FLOSS is that the software acts as compensation for publicly funded research projects, as is the case for any publication in journals. Although the lack of FLOSS is not directly relevant from an academic point of view, it ultimately reduces the long-term net research time. Precious efforts have to be devoted to implement the software basis prior to, for instance, implementing scientifically valuable physical models. Therefore, a lack of FLOSS is indeed relevant to uphold the high-pacing field of MNDS, and thus by extension makes it relevant from an academic research perspective. However, FLOSS offers drawbacks as well, most prominent of all is the potential for code hijacking.

As the code is publicly available the code can be accessed and distributed under a different label by external people, who take credit for the work conducted by the original authors. Another potential drawback is that due to the limited funding for developing open source software the tools tend to not offer the same level of reliability than proprietary software. Also related to a shortage of funding, FLOSS lacks advanced software support such as phone or email-based services, being of particular importance to companies for resolving issues in a timely manner.

The lack of software reuse is especially troublesome in an academic context. Increased development time is clearly a drawback in the short term, causing a research group to lag behind groups, where software engineering[4] is not a concern. The additional need for trained programmers is also particularly challenging in academia, since the workforce is typically smaller compared to the private sector. Also, there is a higher fluctuation of software engineers due to the inherent flow of scientific personnel, particularly PhD students. Furthermore, the majority of young researchers in physical modeling areas are usually quite skilled in physics and mathematics, whereas programming skills are typically underdeveloped. Overall, the number of physical modeling scientists with an advanced software engineering background is small, further contributing to the challenge of scientific software engineering in academia. It is thus of considerable importance to increase the level of reusability in order to support the continuing progress in physical modeling.

1.2 Software Users

The raised issues regarding software engineering for MNDS indicate different types of software users. The fact that this work deals with software approaches ultimately aimed to be utilized by users, requires an analysis of the available user types to allow for efficient code development tuned to the individual requirements. Overall, three user categories are identified, those being the *end user*, the *advanced user*, and the *developer*.

The *end user* is a person who utilizes software tools and is not interested in the technical details of the tool, either due to a lack of interest, time, or skill. The end user interprets software as a *black box* and thus only cares about the input and output quantities. An example from academia would be a physicist who develops a mathematical model to describe a physical phenomenon. In this case the researcher is solely interested in software tools allowing to implement and to apply the mathematical model with minimum effort. Concerning the private sector, in particular the field of MNDS, the end user is typically represented by engineers, being highly skilled in a specific field. From a development perspective, the end user requires particular treatment regarding the usability, meaning that technicalities must be hidden as good as possible. Also, automatisms have to be provided, initializing certain options to reasonable default values. Overall, the effort of using the software has to be minimized as much as possible.

The *advanced user* is similar to the end user albeit offering additional skills and flexibility to investigate (possibly new) software tools. Overall, the advanced user requires access to technical details, either to improve the quality of the software output, to reduce the computation time, or simply out of interest.

[4] *Software engineering* involves all steps of software production, including design, development, operation, and maintenance [3].

This particular type of user is primarily found in academia, as the scientific freedom associated with research positions allows to investigate available approaches, such as represented by software tools. A typical example would be a researcher with a background in semiconductor device simulation as well as mesh generation. In this particular case, the advanced user potentially wants to investigate the mesh quality prior to conducting the actual device simulation. On the contrary, the private sector rarely harbors such advanced users due to the fact that it is usually too expensive for a company to assign highly skilled engineers to investigate details on techniques and approaches required for generating results. Concerning software engineering, supporting an advanced user requires to expose technical details and customizable options.

The *developer* - as the name indicates - develops software used by either advanced or end users. Contrary to advanced or end users a developer is only marginally active in generating results of applications. However, he or she is highly interested in the means to generate them. In academia a developer relates to a researcher who develops or maintains software. Aside from the software responsibilities, a developer in academia usually has to publish. On the contrary, in the private sector, developers are typically represented by software engineers who have to focus entirely on the software tasks and are usually not required to publish their work. Developers require low-level access to the software, e.g., an application programming interface (API) and corresponding documentation.

1.3 Frameworks

In essence, a framework can be abstractly described as a structure providing the ability for adaption and expansion of components ultimately enabling to create a specialized application. In the context of software engineering, a framework can provide reusable and customizable software components to be utilized and specialized for specific applications [8]. Among the differences between a framework and a software library is the fact that a framework offers the so-called *inversion of control property*. A user customizes specific framework components and yields the actual execution of the final application to the framework. Frameworks make sense for different types of users, such as end users who could, for instance, interact (i.e. customize) with graphical user interface (GUI) components which are part of a larger simulation platform. On the contrary, developers could utilize compiler frameworks, allowing to generate specialized compiler tools.

Due to the plethora of available simulation tools, framework approaches merit special consideration when designing new future-proof and flexible simulators. For instance, different simulation tools can be wrapped into components, thus becoming available to the framework infrastructure via defined interfaces. These tools can, by extension, also be utilized in a unified manner, not only significantly increasing usability, but also introducing synergy effects for the wrapped simulation tools. For instance, a GUI-based framework can provide its simulation components access to the visualization backend. Also, frameworks - due to their modular nature - natively support hybrid closed source and FLOSS projects. The framework itself can be made open source, while some components, holding the potentially restricted code (represented by closed source implementations), can be individually made public or private.

1.4 Research Goals

The goal of this work is to tackle the previously introduced challenges by developing and applying software design concepts and techniques to implement reusable software libraries and upon those develop frameworks to ultimately provide flexible simulation tools for the field of MNDS. The developed implementations are freely available under open source licenses, specifically aimed to strengthen the field of MNDS, currently offering an under-developed FLOSS ecosystem. Although this work focuses on MNDS, the developed approaches are applicable to other areas of the general field of computational science and engineering (CSE).

The conducted research work particularly focuses on the following aspects:

- **Reusability** refers to utilizing an implementation in different applications.
- **Flexibility** allows to change the setup of an application with minimal effort.
- **Usability** refers to the ease with which users interact with the software.
- **Maintainability** allows to uphold the code quality.
- **Expandability** denotes the ability to add functionality.

Several exemplary applications are depicted, validating the approaches for practical use. For instance, classical semiconductor device simulations are conducted via a so-called device simulation framework and an interactive simulation framework, whereas finite element simulations are decoupled and executed by a so-called component execution framework. The given application examples underline the increase in reusability and flexibility of the resulting simulation applications compared to the currently available simulation tools.

1.5 Outline

In the following, an overview of the thesis is given:

Chapter 2 discusses related research work, categorized in frameworks, simulation tools, as well as software libraries and tools.

Chapter 3 presents significant software engineering methods and tools, which are applied in the presented approaches. A short overview of relevant software programming paradigms is provided. The motivation for using the programming language C++ is introduced, which is the primary language for the presented implementations. Also, the concepts of component-based software engineering (CBSE), being an applied framework engineering method, are discussed as well as increasing the general level of reusability in software projects by applying library-centric software design (LCSD).

Chapter 4 discusses device simulation frameworks. In particular, the basic semiconductor equations are sketched, providing the mathematical basis for conducting device simulations. The requirements and challenges of a device simulation framework are investigated, followed by the introduction of the ViennaMini project [9], tackling the raised issues.

Chapter 5 investigates component execution frameworks. A solid overview of high performance computing (HPC) platforms is given - especially of interest to frameworks - followed by an analysis of the requirements and challenges of such component-based approaches. A framework approach - based on the ViennaX project [10] - focusing on the discussed issues is introduced.

Chapter 6 introduces an interactive simulation framework. The peculiar requirements and challenges are investigated, which is succeeded by the presentation of the ViennaMOS project [11], facing the introduced challenges.

Chapter 7 evaluates the presented research work. The approaches and techniques are summarized and an outlook on future work is provided. Finally, a conclusion is given relative to the initially defined research goals.

Chapter 2

Related Work

This chapter provides an overview of relevant research work. Both, software aspects and physical modeling aspects, are investigated, clearly depicting the need for flexible simulation approaches. Frameworks in the general field of CSE are introduced followed by an overview of simulation tools in the field of MNDS as well as a selection of relevant software libraries and tools.

2.1 Frameworks

In the rather broad field of CSE a plethora of frameworks is available, focusing on different application areas. In this section, an overview of notable frameworks is given in alphabetical order.

ANSYS [12] provides several different commercial engineering software products, such as an finite element analysis (FEA)-based multiphysics package. Different application areas are covered, for instance, automotive, aerospace, energy, and electronics. A comprehensive selection of pre- and postprocessing functionality is provided.

Cactus [13][14] is a multi-purpose framework, which has its roots in the field of relativistic astronomy. The framework is available under the GNU lesser general public license (LGPL) and focuses on data parallel approaches. The design follows a modular approach and supports different target architectures as well as collaborative code development. The central part of the framework (called ”flesh”) connects the individual application modules (called ”thorns”), typically containing the implementations of the actual simulation. Communication between thorns is realized via the framework's API. Connections between thorns are defined in configuration files, which are processed during compile-time.

The **common component architecture (CCA)** [15][16][17] is a standard and applies so-called component-based software engineering (Section 3.3) to encapsulate units of functionality into components. Data communication between components is implemented via so-called ports. An interface definition language is used to describe the interfaces of components by simultaneously being independent of the underlying programming language.

The actual connection mechanism of the individual components via the interfaces requires end user interaction. The CCA standard has been applied in several projects [16], such as the high-performance computing framework CCAFFEINE [18] and the distributed computing frameworks XCAT [19], Legion [20], and SCIRun2 [21].

COMSOL Multiphysics [22] is a commercial FEA simulation software for engineering and physics applications, supporting multiphysics simulations. The tool provides several essential pre- and postprocessing facilities such as visualization and mesh generation.

The **COOLFluiD** [23][24][25] project enables multiphysics simulations based on a component framework and is primarily designed for data parallel applications in the field of computational fluid dynamics. The source code is available under the LGPL. The core is a flexible plugin system, coupled with a data communication layer based on so-called data sockets. Each plugin can set up data sockets which are in turn used to generate a dependence hierarchy driving the overall execution.

The **directed acyclic graph unified environment (DAGuE)** [26][27] enables scientific computing on large-scale distributed, heterogeneous environments. The source code is available under a license similar to the Berkeley source distribution (BSD) license. The basis of DAGuE is a directed acyclic graph (DAG)-based scheduling engine, where the nodes are sequential computation tasks and the edges refer to data movements between the tasks. Computational tasks are encapsulated into sequential kernels. A DAGuE-specific language is used to describe the data flow between the kernels.

The **earth system modeling framework (ESMF)** [28][29] provides the setup of flexible, reusable, and large-scale simulations in climate, weather, and data assimilation domains. The source code is publicly available under the University of Illinois-National Center for Supercomputing Applications License. The software design is based on a component approach, enabling the division of functionality into reusable components offering a unified interface. The parallelization layer, such as a distributed memory model, is abstracted by a virtual machine approach and focuses on data parallel and basic task parallel approaches.

Uintah [30][31] is a large-scale multi-physics computation framework available under the Massachusetts Institute of Technology (MIT) License. Uintah solves reacting fluid-structure problems on structured meshes, supporting adaptive mesh refinement. Uintah enables task and data parallel applications. The primary area of application is computational mechanics and fluid dynamics. The framework is based on a DAG representation of parallel computation and communication to express data dependencies between multiple components. Each node in the graph corresponds to components which in turn represent a set of tasks. The data dependencies between components are modeled by edges in the DAG.

The FLOSS frameworks provide a set of reusable components which can be combined to form the actual application [32]. The frameworks primarily support large-scale data parallelism by, for instance, providing distributed data structures based on the message passing interface (MPI). The focus is clearly on parallelizing and decoupling the computationally intensive part of scientific simulations. Although such an approach provides a high level of reusability by simultaneously supporting large-scale parallel applications, usability is limited as these frameworks are typically utilized on supercomputers, inherently not favoring GUIs. However, GUIs are an integral part of a high-usability application. Therefore, additional tools required for pre- and postprocessing, such as data visualization, are outsourced to external tools.

In contrast, commercial simulation software provides applications with a high level of usability. However, in addition to being cost-intensive, commercial software in general struggles to support alterations of the simulation's internals, due to its proprietary nature. Having access to simulation internals, such as the implementation of physical models, is of utmost importance to physical modeling research, allowing for an evaluation of new models or techniques. Typically, commercial products try to tackle this problem by providing an API to allow a controlled external access to internal mechanisms; however, this attempt obviously does not provide as much flexibility as a FLOSS approach. Also, commercial tools tend to focus on mainstream functionality, thus specialized customer requests are often ignored or deemed too expensive.

2.2 Micro- and Nanoelectronics Device Simulation Tools

The field of MNDS offers highly specialized publicly available simulation tools, of which an overview is presented in the following in alphabetical order.

Archimedes [5][33] is a two-dimensional Monte Carlo Boltzmann transport based simulation tool for submicron and nanoscale semiconductor devices. Various physical effects and transport models can be investigated for electrons and heavy holes with respect to a rich set of materials. Heterostructures as well as electrostatic and magnetic fields by solving Poisson's and Faraday's equation are supported. The tool is released under the general public license (GPL) and coded in the C programming language. Additionally, an online, GUI-based version of Archimedes is provided via the nanoHUB platform.

Genius [34] is available as a publicly accessible version under the GNU GPL. The FLOSS version supports two-dimensional device simulation based on the drift-diffusion (DD) model. Lattice heating is taken into account by, for instance, a temperature corrected DD model. A rich set of functionality is provided, such as various mobility models, an energy transport model, and several impact-ionization models.

Gold Standard Simulations [35] specializes in simulating statistical variability in nano-CMOS devices and provides corresponding commercial simulations tools. More specifically, the tools support the physical simulation of statistical variability, statistical compact model extraction, and statistical circuit simulation.

Minimos-NT [36][37] is the successor of the Minimos [38] simulator and is commercially supported. Minimos-NT is a general-purpose semiconductor device simulator, providing a general-purpose, multi-dimensional semiconductor device simulator. The simulator supports stead-state, transient, and small-signal analysis of arbitrary devices. Also, mixed-mode device and circuit simulations based on compact models are supported.

nanoHUB is a platform hosting scientific tools, primarily in the field of computational nanotechnology [39][40]. At the time of writing this thesis from the total number of 325 tools 17 tools (corresponding to 5.2%) are tagged as open source[1]. Figure 2.1 gives an overview of the accumulated code lines of each project. The Count Lines of Code [41] tool has been used to quantify the code base implemented in languages such as C/C++, Python, Matlab, and Fortran. Irrelevant data has been - to a large extent - ignored, like comments and building instructions. Of the 17 open source tools 41% have between 100 and 1 000, 35% offer 1 000 to 10 000, and 24% provide 10 000 to 100 000 lines of code. Therefore, the majority of the available free open source tools can be considered to be small to medium scale-size projects, further underlining the lack of FLOSS-based device simulation tools of considerable size. Overall, nanoHUB provides free registration for an online account, enabling the execution of tools directly from within a web browser. The computational resources are provided by the nanoHUB facilities, thus no compilation and/or installation procedure is required.

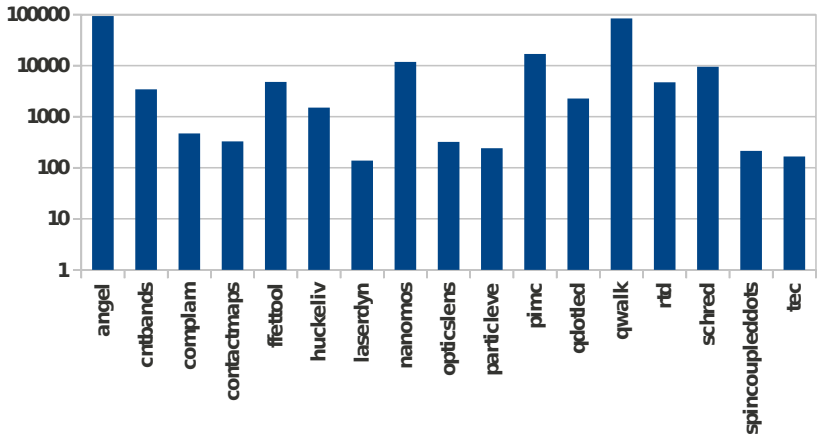


Figure 2.1: The accumulated code lines of the 17 open source nanoHUB simulation tools are depicted. Seven tools (41%) have 100 to 1 000 code lines, six tools (35%) have 1 000 to 10 000, and four tools (24%) offer 10 000 to 100 000 code lines, respectively.

[1] Discrepancies in nanoHUB's tagging system do not allow for an accurate count as the available tag-based filtering mechanisms yield slightly different results (±1-2 tools) for the *open source tool* tag.

NanoTCAD ViDES [42] supports the simulation of nanoscale devices through the self-consistent solution of the Poisson and the Schrödinger equations by means of the Non-Equilibrium Green's Function formalism. The tool allows for the simulation of transport in graphene nanoribbons, carbon nanotubes, and two-dimensional (bilayer) graphene field-effect transistors. The simulator is distributed as a Python module, utilizing high performance C and Fortran based subroutines. The package is released under the BSD License.

Silvaco [43] provides a broad set of commercial simulation tools for TCAD, interconnect modeling, and analog/mixed-signal/radio frequency analysis. A broad set of modeling and analysis tools are provided, allowing for a wide range of simulations and evaluations.

Synopsis [44] provides a plethora of commercial simulation tools, covering a variety of application categories, such as TCAD, verification, manufacturing, and system-level design. Extensive pre- and postprocessing facilities are provided, such as structure and mesh generation as well as visualization.

ViennaSHE [45][46] is a multi-dimensional, self-consistent semiconductor device simulator based on the deterministic solution of the Boltzmann Transport Equation using Spherical Harmonics Expansions. ViennaSHE provides a standalone simulation application as well as an API for utilizing the simulator by other implementations. The tool is released under the MIT License and written in C++.

The presented open source simulation tools are highly specialized. However, they share the requirement for certain pre- and postprocessing software components. For instance, each tool requires visualization capabilities to enable investigations of the simulation results. Another typical requirement is the generation of the simulation domain and the access to material parameters. A detailed analysis of the available open source simulation tools leads to the conclusion that these tools treat these aspects in a marginal manner. For instance, only a static set of material parameters is supported, which is hard coded into the simulation code. This specific aspect introduces the need for a flexible material mechanism, providing simulation tools access to various material database backends. With respect to the commercially distributed simulation tools, the advantages and disadvantages, as introduced in Section 2.1, apply here in a similar manner.

2.3 Software Libraries and Tools

This section provides an alphabetical overview of libraries and tools utilized for the implementations introduced in this thesis. Using these packages was vital to the developed techniques and implementations, ultimately reducing the development time considerably.

The Boost C++ Libraries [47] provide access to a vast set of functionality via individual libraries. The libraries are based on a peer-review system, aiming to impose quality standards upon new library members. The Boost License supports non-commercial and commercial use and cover a plethora of categories, such as data structures, algorithms, and concurrent programming. Table 2.1 lists certain Boost libraries in closer connection to this work.

Name	Description
Graph [48]	A generic graph library similar to the Standard Template Library (STL)
MPI [49]	A convenience C++ layer for the C-based MPI API
Serialization [50]	A library for decomposing an arbitrary set of C++ data structures into a sequence of bytes
Smart Ptr [51]	A library providing automatic and safe pointer classes
uBLAS [52]	A linear algebra library
Variant [53]	A safe, generic, stack-based discriminated union container

Table 2.1: An alphabetical selection of Boost libraries utilized in this work.

Mesh generation tools divide a physical domain into so-called mesh elements [54], such as triangles or cubes. This generation step is challenging, as usually several properties have to be met. For instance, a mesh must conform - as good as possible - to the shape of the simulation object, to enable a proper representation and thus by extension allow for meaningful simulation results. Also, the elements must be suitable shaped and sized to reduce discretization errors [55], and the number of mesh elements in total has to be kept small to minimize the subsequent simulation time. The challenge of mesh generation holds especially true for the field of CSE, where meshes are used to discretize equations in finite form via, for instance, finite volume methods. Several mesh generation tools are available, supporting different dimensions and meshing algorithms [56], such as advancing front, octree, and incremental Delaunay. Each meshing approach generates meshes with different properties, for instance, three-dimensional simplex, i.e., line, triangle, and tetrahedron meshes which satisfy the conforming Delaunay property [54]. Table 2.2 gives an overview of FLOSS-based meshing tools, the majority of which is being used by the ViennaMesh library [57].

ParaView [63] is a data analysis and visualization application supporting a variety of platforms, such as Windows and Unix-like systems. The application is coded in C++ and is available under a BSD license. ParaView utilizes the VTK library for the visualization backend and the Qt framework for the GUI frontend. Due to ParaView's popularity and modular GUI approach it acted as a reference for a developed interactive simulation framework (Section 6.2).

Name	Geometrical dimension	Topological dimension	Hypercube	Simplex	Incremental Delaunay	Advancing Front
CGAL [58]	2,3	2,3		•	•	
Gmsh [59]	2,3	2,3	•	•	•	•
Netgen [60]	2,3	2,3		•		•
Tetgen [61]	3	3		•	•	
Triangle [62]	2	2		•	•	

Table 2.2: Alphabetical list of popular FLOSS-based mesh generation tools and their properties. Simplex denotes triangular and tetrahedral mesh elements, which are usually implemented via an unstructured data structure. Hypercube relates to quardilateral and hexahedral mesh elements, typically implemented via structured meshes.

The Qt Framework [64] is a cross-platform application and GUI framework using primarily C++. Qt extends the standard C++ language features by macros and a code generator, the so-called meta-object compiler. Qt supports desktops as well as mobile platforms. Furthermore, interfaces are available to non-gui features, such as SQL databases, extendible markup language (XML), threading, and networking support. The Qt framework was used in this work for developing a modular GUI-based simulation framework (Chapter 6).

The ViennaX[2] Collection [65] is designed in the image of the Boost libraries, aimed to provide researchers with a rich set of ready-to-use and easily accessible FLOSS-based functionality. In essence, ViennaX is a set of libraries and applications. The ViennaX project's primary goal is to strengthen the open source movement in the field of MNDS. Although a couple of software packages are considered domain-specific, others are not and can thus be utilized in various application areas of CSE. For instance, ViennaCL- a general purpose linear algebra library - is utilized by, for instance, mechanical and electrical engineering applications. Table 2.3 and Table 2.4 depicts the current set of applications and libraries, respectively. During the course of research giving rise to this thesis, several of the ViennaX libraries and applications have been supported, maintained, extended, utilized, and initiated.

[2] ViennaX is pronounced *ViennaStar*.

Name	Description
ViennaMini [9]	A classical multi-dimensional device simulator
ViennaMOS [11]	A GUI-based modular framework tailored to the requirements of MNDS
ViennaProfiler [66]	A centralized code profiling application
ViennaSHE [45]	A deterministic Boltzmann solver based on spherical harmonics expansions for semiconductor devices
ViennaWD [67]	A stochastic device simulator in the classic and quantum domain
ViennaX [10]	A high-performance plugin execution framework for scientific computing

Table 2.3: Alphabetical list of Vienna* applications.

Name	Description
ViennaCL [68]	A linear algebra library using CUDA, OpenCL, and OpenMP
ViennaData [69]	A library for attaching application-specific data to arbitrary objects
ViennaFEM [70]	A finite element library with a symbolic math kernel
ViennaFVM [71]	A finite volume library with a symbolic math kernel
ViennaGrid [72]	A mesh data structure library
ViennaIPD [73]	A control language library for scientific simulations
ViennaMaterials [74]	A flexible material library
ViennaMath [75]	A symbolic math library for compile time and run time operations
ViennaMesh [57]	A library for mesh generation, adaption, classification of multi-segmented meshes and geometries

Table 2.4: Alphabetical list of Vienna* libraries.

The visualization toolkit (VTK) [76] library provides functionality in the field of computer graphics, image processing, and visualization. VTK is a cross-platform library and is based on a C++ class library with support for other languages, such as Python. VTK provides a wide range of visualization algorithms, such as vector methods, as well as modeling techniques, like Delaunay mesh generation. The library provides interaction support with GUI frameworks, such as Qt. VTK was used in the combination with Qt to provide flexible rendering facilities, both for charts as well as for 3D renderings including scalar and vector field visualization (Chapter 6).

Chapter 3

Methods and Tools

Research software engineering requires utilizing specific methods and tools, among those are programming languages and techniques, as well as design concepts and third-party libraries. Therefore, the methods and tools proven vital for the presented research are discussed in this chapter. Section 3.1 lists relevant programming paradigms. Section 3.2 introduces some relevant features of C++, the primary programming language used in this work. Section 3.3 presents the term CBSE whereas Section 3.4 discusses LCSD.

3.1 Programming Paradigms

Programming paradigms provide fundamental styles for implementing software. In the following, a short overview of the individual programming paradigms utilized in this work is given. Extensive comparisons are available in the literature [77][78].

- **Imperative programming** focuses on executing a sequence of instructions, operating on a state. Imperative programming is the primary utilized programming style in scientific implementations, due to its simplicity and intuitive approach. This paradigm is supported by all major languages relevant to CSE, such as Fortran, C, and C++.
- **Object-oriented programming** aims for modularity by wrapping functionality into reusable classes, offering defined interfaces for external utilization [79]. Polymorphism is implemented via so-called virtual class hierarchies, allowing to specialize implementations according to run-time information. The object-oriented approach is the most widely taught paradigm at universities, especially with mainstream languages like Java and C++. The paradigm has also some basic support by C and Fortran, although the features are not as extensive as with typical object-oriented languages, such as C++.
- **Functional programming** uses mathematical functions for formulating algorithms and programs by simultaneously avoiding states of objects and mutable data [80]. Functional programming supports so-called lambda expressions, based on the lambda calculus [81]. Pure functional programming languages are available, such as Haskell [82]. Since the new C++ standard 2011, C++ natively supports lambda functions, thus enabling functional style programming. However, previous C++ generations required external libraries to enable functional-style programming, such as Boost Lambda [83] and Boost Phoenix [84].

- **Generic programming** fosters code reuse by *lifting* algorithms or data structures from concrete implementations to their most general form [85][86][87]. For instance, in C++ generic programming approaches polymorphism from a different angle than the object-oriented paradigm. Where the latter relies on run-time decisions - introducing additional execution overhead - the first uses compile-time dispatches, omitting run-time dispatch costs. C++ supports generic programming, due to its type system and template mechanism. Also, C++'s standard template library makes extensive use of this paradigm, as the algorithms, such as `copy`, are separated from the datastructures, like `vector`, allowing to use an algorithm with different datastructures and vice versa.

- **Meta programming** enables programs to generate or manipulate themselves or other programs [88][89]. Meta programming techniques are considered to be rather exotic, and are typically not taught in engineering curricula. Template meta programming, as supported by C++, is a type of meta-programming, where the template system is used to generate source code at compile-time. Consequently, compilation times are increased significantly, impeding the development process due to the potentially increased idle time. In C++, template meta programming is used to implement the generic programming paradigm. Although meta programming in general promises significant reduction in run-time costs, it requires advanced programming skills to develop, debug, and maintain implementations. Closely connected to this issue is the term *leaky abstraction*, referring to the fact that intentionally hidden implementation details are exposed to the developers in case of errors, further complicating debugging. This exposure is triggered by the compile-time nature of template meta programming.

3.2 The C++ Programming Language

The implementations presented in this work are based on the C++ programming language due to several reasons.

C++ allows for high-level implementations by simultaneously supporting high performance. High-level refers to the fact that the language supports high abstraction levels, i.e., implementation specifics are hidden from the developer to ease the burden of development. Concerning high performance, C++ - as a multi-paradigm language [24][77][78] - supports generic and meta programming, in turn enabling high performance implementations. Examples of successful high performance applications based on C++ are the Matrix Template Library (MTL) [90], the finite element library deal.II [91][92], and the Blitz++ library [93].

Furthermore, C++ is mature [94] and is ranked to be one of the most popular programming languages [95]. The popularity seems to decline due to an substantial increase in developers for mobile devices based on Java and Objective-C. However, according to the long term trends, the popularity of C++ has only slightly decreased over the last six years. Additionally, a new C++ standard has been released recently [96], adding a vast amount of new functionality. Due to the popularity of C++ it is ensured that there will be continuing support from compiler and library developers and standardization committees to further advance C++.

Additionally, several high quality compilers are available, for instance, the open source compilers GNU GCC [97], Clang [98], and the proprietary Intel compiler [99]. As each compiler usually implements the standard differently and also provides varying optimization and debugging approaches, having access to different compilers enables to further stabilize and tune implementations by, for instance, using different debugging mechanisms.

A very important property of C++ is the ability to interface with other programming languages. This is especially important in the field of HPC, where C and Fortran are the predominant programming languages [100]. Additionally, Python becomes more and more popular due to it's flexibility and ecosystem of feature-rich extensions. Where C and Fortran code can be linked with C++ in a straightforward manner, for bridging Python with C++ in both directions the convenience library Boost Python can be used [101].

3.3 Component-Based Software Engineering

CBSE [24][32][102] focuses on reusability by separating functionality into reusable components. Communication between the components is achieved via defined interfaces, thus a component's sole link to the framework is restricted to its interface. CBSE can be seen as an advancement of an object-oriented class approach, as components allow for a higher degree of abstraction.

Components are considered to be the building blocks of actual applications, therefore they need to be connected and executed appropriately to become an application. This is the task of the previously introduced frameworks (Section 1.3). The key advantage of such a component approach lies in the ability to reuse components for different applications, thus drastically reducing development time. Figure 3.1 schematically depicts CBSE.

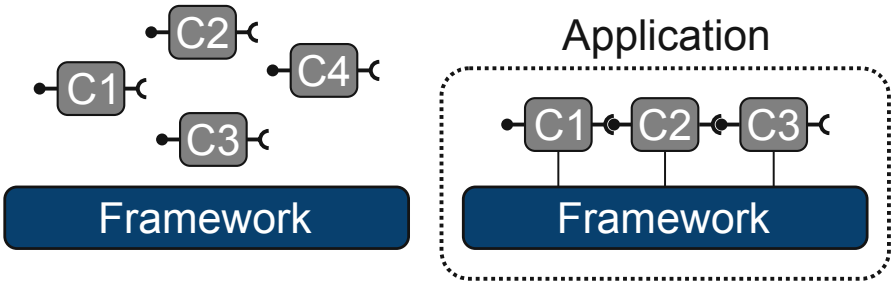


Figure 3.1: **Left:** A set of components (C1-C4) is at the framework's disposal. **Right:** An application is defined by connecting various components.

3.4 Library-Centric Software Design

Another approach to increase re-usability is to apply a LCSD which allows to increase the efficiency of application development in the sense that code reuse decreases development time [77]. LCSD can be seen as the natural companion of the generic programming paradigm, in the sense that functionality is not embedded into a structure but extracted and generalized into a standalone library offering its functionality via an API so it can be accessed by applications or other libraries.

LCSD - although introducing additional short-term development overhead - amortizes in the long run, when the number of utilizing applications or libraries increases. This holds especially true in an academic setting, where typically various simulation codes are available requiring a plethora of functionality. A popular example of the LCSD approach with respect to C++ is the Boost library collection [47].

Overall, LCSD fosters *slim* applications, meaning that the application itself merely interfaces with various libraries, thus allowing the opportunity to keep the application's code base to a minimum. This fact not only improves development time of the application, but also supports maintenance, as a small code base results in reduced maintenance time in a straightforward manner.

Figure 3.2 depicts the introduced design concepts with respect to initial development effort as well as degree of reusability and maintainability. The LCSD approach offers the highest initial development effort but also the highest degree of reusability and maintainability.

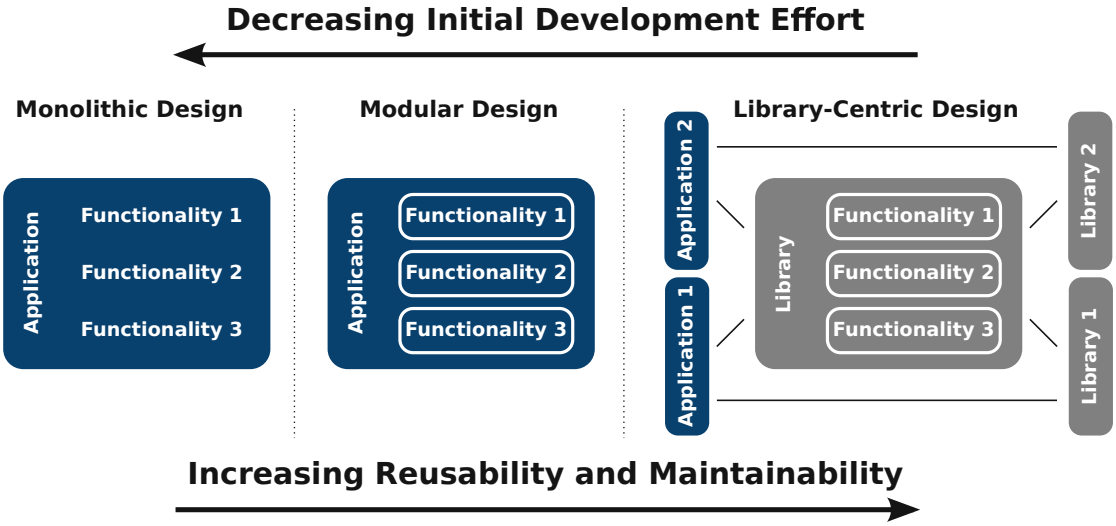


Figure 3.2: Comparison between monolithic, modular, and LCSD approaches. **Left:** With monolithic design, functionality is tailored to a specific application, and is thus challenging to reuse or maintain in other projects. **Middle:** Modular design encapsulates functionality into more accessible modules. **Right:** LCSD focuses on extracting functionality into libraries. This not only reduces the code base of applications but also enables other libraries to make use of synergy effects. An increasing level of reusability and maintainability typically claims an increased initial development effort, due to additional development overhead.

Chapter 4

Device Simulation Framework

A device simulator consists of several software components, required for computing the electrical characteristics of a given device, consisting of a mesh and additional physical information like a doping profile. For instance, a material database is required providing the material parameters used for the mathematical models. The interplay between the different software components usually gives rise to tightly interwoven, i.e., monolithic, implementations, resulting in inflexible simulation platforms. Changes to the implementation, such as switching linear solver backends, are thus usually cumbersome and require significant implementation efforts. This fact puts pressure on the software design to provide a flexible long-term solution. Interfacing the simulator with LCSD-based libraries merits special consideration, due to the significantly increased level of reusability provided by the already available functionality (Figure 4.1).

In this chapter, a typical set of basic equations required for device simulation is briefly discussed (Section 4.1). The primary challenges of implementing a device simulator are depicted as well as the input and output dependencies are identified (Section 4.2). Based on these evaluations, an approach for a device simulation platform tackling these challenges is discussed and complemented by several examples (Section 4.3).

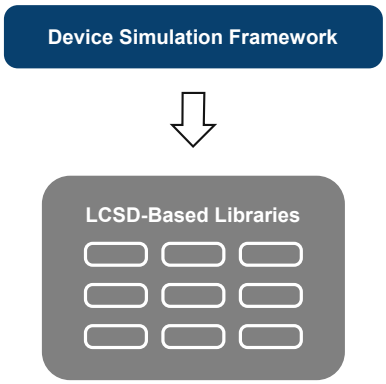


Figure 4.1: A device simulation framework significantly benefits from interfacing with LCSD-based libraries, due to access to already available functionality.

4.1 The Basic Semiconductor Equations

The basic semiconductor equations [1] consist of the Poisson equation (Equation 4.1),

$$\nabla \cdot (\varepsilon \cdot \nabla \varphi) = q(n - p - N_D + N_A) \tag{4.1}$$

the continuity equations for electrons (Equation 4.2) and holes (Equation 4.3),

$$\nabla \cdot \boldsymbol{J_n} - q\frac{\partial n}{\partial t} = qR \tag{4.2}$$

$$\nabla \cdot \boldsymbol{J_p} + q\frac{\partial p}{\partial t} = -qR \tag{4.3}$$

as well as the current relations for electrons (Equation 4.4) and holes (Equation 4.5)

$$\boldsymbol{J_n} = -q\mu_n(n\nabla\varphi - V_T\nabla n) \tag{4.4}$$

$$\boldsymbol{J_p} = -q\mu_p(p\nabla\varphi + V_T\nabla p) \tag{4.5}$$

based on the DD model [103].

ε denotes the permittivity, q the elementary charge, N_D the donor doping concentration, and N_A the acceptor doping concentration. With respect to the continuity equations and the current relations, $\boldsymbol{J_n}$ refers to the electron current density, $\boldsymbol{J_p}$ the hole current density, R the recombination rate, and V_T the thermal voltage.

The DD model considers two charge transport mechanisms, those being charge carrier drift and diffusion, respectively. In general, charge carrier drift due to the presence of an electric field, implemented by the first term of the DD model, containing the gradient of the potential. On the contrary, diffusion is a fundamental process, which aims to establish a thermodynamic equilibrium in an initially imbalanced physical system. In the case of semiconductor physics, diffusion is achieved by carrier migration from areas with high concentration to areas where the concentration of particles is lower. Therefore, the DD model includes the second term, incorporating the gradient of the charge carrier concentration.

Substituting the current densities in the continuity equations (Equations 4.2-4.3) for the current relations (Equations 4.4-4.5) yields a system of three partial differential equations (PDEs), coupled via the potential φ, the electron concentration n, and the hole concentration p. These PDEs are typically discretized with the finite volume method to conserve the current [1], stabilized by the Scharfetter-Gummel discretization [104], and solved via nonlinear iterative solvers, such as the Newton scheme or the Gummel method [1]. The solved potential, electron concentration, and hole concentration distributions are used to compute, for instance, the current densities, essential to evaluating the device performance via the current-voltage characteristics, where the evaluated current at terminal contacts is related to the individually applied terminal voltages.

Although the basic semiconductor equations were successfully applied in the last decades, with the continuing shrinking of the device dimensions the typically applied DD model struggles to provide reasonable predictions [46]. Therefore, additional models have been developed [105], underlining the need for a modern device simulation software to support an exchangeable and expandable modeling backend.

As the focus of this work is on investigating approaches for a flexible device simulation framework, the actual transport model required for conducting device simulations is of marginal importance. Therefore, the simple yet representative DD transport model is sufficient for the subsequent investigations.

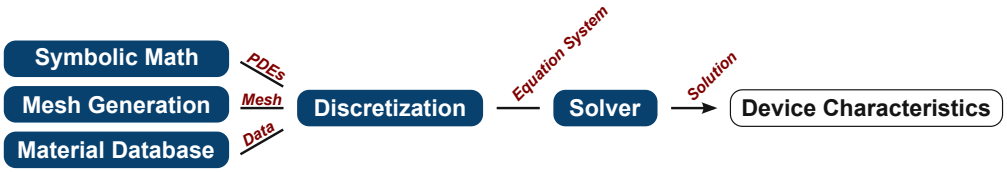


Figure 4.2: The essential components of a device simulator are shown. The discretization scheme assembles an equation system according to the defined PDEs, the input mesh, and the required material data. The equation system is solved via a solver component, computing the solution from which the device characteristics is derived.

4.2 Requirements and Challenges

The essential building blocks of a device simulator consist of several key components, as depicted in Figure 4.2. The discretization scheme, such as the finite volume method (FVM), assembles an equation system according to the defined PDEs, the input mesh, and the required material data. The equation system is solved via a solver component, computing the solution from which the device characteristics is derived. Not only is the implementation for each individual component a challenge, also indicated by the fact that most of them offer their own field of research, but also the interplay of the components. Typically, the components are not implemented orthogonally, which tends to result in monolithic implementations, not favoring exchangeability or expandability. The main challenge of a flexible device simulator is thus to focus on a high degree of orthogonality between these components.

In the following, we discuss the individual requirements and challenges for each of the introduced components of a device simulator.

4.2.1 Mesh Generation

Mesh generation divides a physical domain into so-called mesh elements. This particular task is complemented by mesh adaptation, enabling to change an already existing mesh according to specific requirements. For instance, geometrical predicates, such as the element shape, can be used to guide a mesh adaptation step, to ultimately improve the *worst* element in a mesh [106]. Aside from geometrical predicates, meshes can be adapted according to distribution-based tagging mechanisms. For instance, mesh regions carrying high gradients in quantity distributions, like doping, are *higher resolved*[1], to minimize discretization errors [55].

As already indicated, device simulation tools usually receive the simulation domain in form of a mesh from process simulation tools, accompanied by doping profiles. The meshes provided by process simulation tools are usually not suited for device simulation, due to different requirements imposed by a different set of physical model equations.

[1] In this context, higher resolved relates to an increased local density of mesh elements.

Among the possible manifestations are highly distorted elements, and insufficiently resolved meshes, especially in regions where high gradients in the device simulation solutions are expected, such as doping transitions. Such unsuited meshes may result in slow convergence or in the worst case in no convergence at all of the solution method used in the device simulator. This circumstance requires the availability of a meshing facility, allowing either to adapt the mesh or to generate the mesh from scratch according to extracted geometry information. However, in this particular case, the initial doping profiles provided by the process simulator must be interpolated to the new mesh, likely offering entirely new mesh elements. The interpolation process ensures that the initial doping distribution is mapped to the new mesh elements, required for the mesh element-wise assembly of the basic semiconductor equations, like Poisson's equation (Equation 4.1).

The use of process simulation tools yielding a mesh including doping informations may be omitted, for instance, for testing purposes. In this specific case, a device simulation tool has to work with a *bare* input mesh generated by mesh generation tools, such as Netgen [60]. In this context, a bare mesh indicates a mesh without any device-related information, meaning that although the mesh provides segments representing the individual parts of a device such as oxides, the mesh object has no knowledge about the physical relevance. For instance, no material information is associated with the segments or whether the individual segments represent a contact, an oxide, or a semiconductor region. Similar to the previous case, where a mesh is imported from a process simulator, the externally generated meshes usually lack domain knowledge thus potentially represents an unsuitable mesh, as already described. The lack of a process simulator requires that a doping profile has to be assigned to the mesh via, for instance, analytic functions, ultimately enabling to perform the actual device simulation. Overall, it is therefore essential to interface a device simulator with meshing facilities, enabling - possibly automatized - mesh adaptation steps to adapt an externally provided mesh according to domain-specific information.

Furthermore, so-called *template devices* are of interest, as they provide convenient, *simulation-ready* devices to end users. This template mechanism allows to entirely omit an external process simulation and mesh generation step, and thus augments the device simulator with a standalone simulation capability. Such a feature is usually of interest for showcases, tests, and tutorial-related purposes. These templates provide a specific device type, such as a two-dimensional metal-oxide-semiconductor field-effect transistor (MOSFET), but expose certain customizable device geometry parameters, such as the channel width. To support a reasonable mesh resolution, the template mechanism has to be coupled with a meshing backend, allowing to generate a high-quality mesh upon updated end user-provided geometry parameters. Similar to the previous case describing the application in the context of an external mesh generation tool, a doping profile has to be distributed on the mesh.

The three discussed application cases of mesh generation and adaption are characterized in Figure 4.3. The challenge is to interface the device simulator with mesh generation facilities, without relying on a single specific meshing backend. For instance, interfacing directly with Netgen confines the simulator to utilize this specific tool for all mesh generation and adaptation tasks.

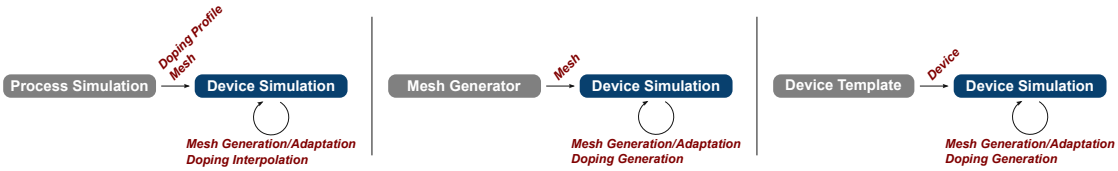


Figure 4.3: Mesh generation and adaptation requirements for device simulation tools are shown. **Left:** An external process simulator provides a mesh and doping profiles. The device simulator changes the mesh according to domain-specific information and interpolates the doping accordingly. **Middle:** A mesh generator provides a mesh to the device simulator. Aside from adapting the mesh according to domain-specific knowledge, a doping profile has to be assigned. **Right:** A device template mechanism allows to generate simulation-ready devices automatically. Based on customized geometry data a mesh and a doping profile is generated.

However, as already indicated different meshing tools - aside from supporting different mesh types such as three-dimensional tetrahedral meshes - implement different algorithms and strategies to generate high quality meshes. Having access to various meshing backends is especially important to the field of TCAD, where input geometries, consisting of thin layers and complex surfaces [107], require high quality mesh generation which is still a matter of ongoing research.

Also, mesh generation might be prone to scaling issues, induced by, for instance, meshes defined in the nanometer regime. Such cases involve very small numbers which - if remained unscaled relative to the device dimensions - might break numerical tolerance limits in the meshing backend, potentially prohibiting the generation mechanism to converge. Therefore, it is advantageous to perform the mesh generation step based on a normalized input, i.e., the mesh is scaled relative to its device dimensions. When the mesh has been generated, it is scaled to the intended regime. Such scaling mechanisms are of general interest to all meshing backends, thus these mechanisms have to be provided in an orthogonal manner, further favoring a unified meshing layer.

Overall, it is of utmost importance to interface the device simulator with a flexible meshing layer, in turn enabling access to the actual meshing backends (Figure 4.4). This way, the meshing tools utilized by the simulator can be exchanged and expanded without additional development effort. The selection of the actual meshing backends can be done either manually, driven by the individual properties of the meshing tools relative to the problem at hand, or automatically, by, for example, heuristical methods.

Figure 4.4: Interfacing a simulator with a single meshing backend limits flexibility with respect to mesh generation tasks (**left**). Instead, when a meshing layer is introduced, access to an expandable set of meshing backends is provided to the simulator via a unified interface (**right**).

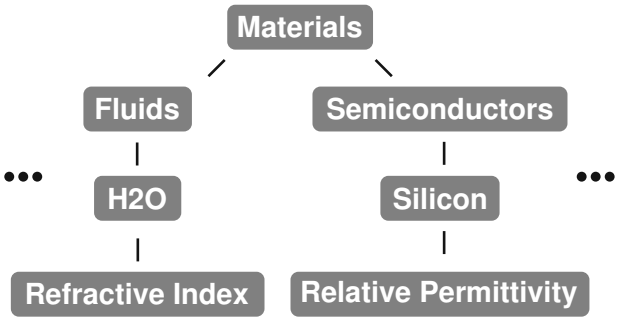


Figure 4.5: Exemplary material Properties schematically mapped on a Tree. Materials do not necessarily share the same properties.

4.2.2 Material Database

Simulation tools usually require a large set of material parameters to carry out scientific simulations, due to the use of equations which include material parameters to model the physical environment [108][109][110]. Among such equations, partial differential equations are especially widespread in the description of complex phenomena and are therefore of special interest for CSE in general. Considering the vast number of phenomena and the related sets of equations for which simulation environments have been and are currently being developed, it becomes apparent that many different material parameters have to be made available in a consistent and reliable manner. The challenge lies not only in the efficient storage of the material data but also in the convenient and fast data access. For instance, considering the case of carrier mobility, in the simplest case the mobility value for a given material is a constant. However, in more complicated cases, i.e., scattering models, the carrier mobility depends on a set of parameters and the solution variables. The storage and access methods need to support such setups.

Overall, accessing parameters is a fundamental task for CSE applications. In C++ parameters can be provided either during compile-time, by hard-coding them into the implementation, or during run-time, by, for example, feeding the parameters to the application via an input file. Where the compile-time approach is considered to be quick with respect to implementation time, the approach lacks reusability and flexibility. The parameters cannot be easily reused by different applications and changes require recompilation. On the contrary, providing the parameters during run-time allows for changing the parameters without recompilation.

Another challenge is to incorporate unit-aware material parameters which are especially important for robust numerical simulations [111]. Not only need the units be linked to each material parameter in the data set, but also automatic unit checks and - if possible - unit conversions have to take place. Such a unit system must support the dynamic nature of the material parameter database, induced by the fact that material parameters are loaded during run-time. In other words, the unit information is usually available as a string, thus the unit system has to be able to process string-based unit expressions.

Tree-based structures merit special consideration for storing material data. The underlying data associated with materials is inherently hierarchical, and can thus be mapped to a tree naturally (Figure 4.5). Also, the variation due to the fact that not all parameters are available or useful for all materials can also easily be accommodated by a tree-based structure.

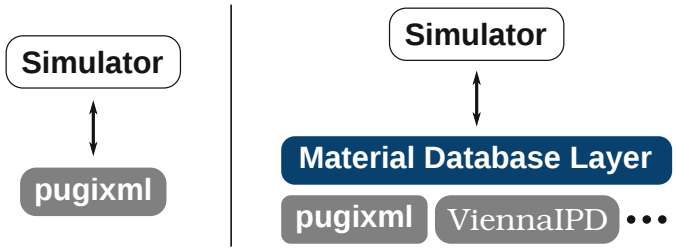


Figure 4.6: Interfacing a simulator with a single material database backend limits flexibility with respect to utilized material languages and access methods (**left**). Instead, when a material database layer is introduced, access to an expandable set of material database backends is provided to the simulator via a unified interface (**right**).

Tree-based data structures natively support path-wise data access, due to the supported intuitive descent along the structure of the tree holding the data. Languages which support a tree-based data structure and path-based data access are, for instance, XML via the XML path language (XPath) query language - as provided by pugixml [112] - or ViennaIPD [37][73]. XML has a distinct advantage, as the language is widespread, thus parameter files can immediately use synergy effects from the large available XML ecosystem, such as GUIs or other well-known XML libraries, like libxml2 [113]. On the contrary, the ViennaIPD language - especially designed for the field of semiconductor device simulation - is more convenient to read due to the absence of tags bounding scoped regions, as is being used by XML. Thus follows the challenge of implementing a flexible device simulator - similar to the previously discussed mesh generation and adaptation case - to facilitate the exchange and addition of material databases. A unified interface is required, ensuring that material database backend changes do not affect the simulation frontend (Figure 4.6).

4.2.3 Symbolic Math

Device simulation tools must have implemented the mathematical models, represented by equations. For instance, the sketched basic semiconductor equations (Section 4.1) are usually used in combination with the finite volume discretization scheme to assemble a system of equations. Typically, the assembly process is interwoven with the formulation of the PDE, confining a formulated PDE to be solved solely with a single specific discretization. Or in other words, the mathematical structure of a PDE is typically sacrificed when transferred to code, because only discretized versions are implemented, as is for example the case for the deal.II library [92].
The FEniCS project [114] provides a highly expressive approach to implement and solve PDEs, however, it relies on code generation and separate library routines, introducing other drawbacks such as inferior support for legacy code [115]. This particular aspect is especially problematic for the field of semiconductor device simulation as a plethora of such implementations is available.

Other highly expressive implementations for such an approach have been developed [77], however, a decoupling between formulating PDEs and the applied discretization is typically not applied. Usually no equation objects are provided which are decoupled from a discretization scheme. Hence, advanced users are required to discretize the equation by hand, as in-depth knowledge of the discretization scheme is required.

This offers utmost flexibility, but on the other side the missing convenience layer results in decreased usability, effectively excluding end users from utilizing such an approach. Therefore, prominent FEA tools like COMSOL Multiphysics [22] provide an automatic discretization approach, hiding the technical details. This enables to reach end users who can therefore focus on the physics, as primarily the PDEs modeling the physical problem are required by the simulation tool.

Also, the basic physical models represented by the utilized PDEs are usually augmented with further models, by, for instance, introducing additional terms into the equation or by adding additional equations such as the heat-flow equation. This augmentation step is essential for device modeling, as it allows researchers to investigate new mathematical approaches for simulating physical phenomena. For instance, such an approach is required for introducing recombination models, such as the Shockley-Read-Hall model [116][117], into the right-hand side of the continuity equations (Equations 4.2-4.3).

Overall, a particular challenge in this context is to retain the mathematical structure of a PDE. The simulation tool should be capable of automatically discretizing an equation object, e.g., a PDE, with an appropriate discretization schema. Such an approach requires a symbolic math kernel, supporting the setup and manipulation of mathematical expressions. Furthermore, these expressions must support run-time changes to enable, for example, the introduction of additional terms to extend the modeling via GUIs or scripting interfaces like Python.

4.2.4 Discretization Schemes

Handling partial differential equations in the computer domain requires means to implement algebraic approximations of these equations while preserving as much information as possible of the original problem [77]. Such algebraic approximations are typically obtained by numerical discretization schemes. For the field of device simulation, the finite difference method, the FVM, and the finite element method (FEM) are typically employed [1].

Consequently, the simulator design has to support different treatments of a particular PDE with different discretization schemes, resulting in different assembly approaches of the equations system (Figure 4.7). The challenge of supporting different discretization schemes is closely coupled to the raised challenges depicted with the symbolic math component (Section 4.2.3). The key aspect is the availability of an equation object which allows manipulations and evaluations.

4.2.5 Solver

The discretization of a PDE results typically in an equation system $Ax = b$. Solving a large system of equations is computational expensive and consequently direct methods, like the Gaussian elimination procedure, cannot be applied in practice due to inadmissible execution times. Therefore, iterative solving procedures, like the conjugate-gradient [118] and the generalized minimal residual method [119] methods, are usually applied.

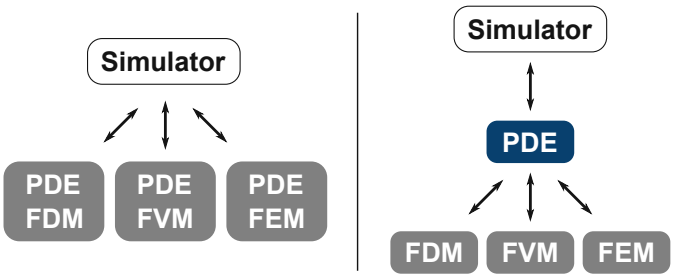


Figure 4.7: **Left:** The usual discretization approach in simulation tools is based on implementing each PDE for each discretization scheme. **Right:** If the PDE can be decoupled from the discretization scheme, the same equation object can be processed by different discretizations, thus increasing reusability.

The convergence behavior of these approaches can be further improved by preconditioners [120], such as the Jacobi method and the family of incomplete lower upper factorization methods. It is obvious, that the nature of the assembled PDE, whether it is linear or nonlinear, determines the solution procedure of the equation system. Nonlinear systems are commonly solved by Newton-type methods.

Several libraries are available which provide relevant tools for solving large systems of equations [68][121][122]. Consequently, the different solver approaches and the different libraries introduce a plethora of prerequisites by the respective API. In order to retain utmost flexibility for a device simulator, the design has to deal with these inconsistent solver interfaces by decoupling the solver backends from the actual implementation (Figure 4.8).

Figure 4.8: Typically a single solver library is utilized in device simulation tools (**left**). By introducing an additional layer, a unified interface to different solver libraries can be achieved, decoupling the simulator from a single specific solver backend (**right**). For instance, linear solvers provided by ViennaCL [68], PETSC [122], and Trilinos [121] can be utilized.

4.3 The ViennaMini Project

ViennaMini is a FLOSS-based device simulation framework [9], specifically tackling the previously introduced requirements and challenges for a device simulation framework (Section 4.2). The implementation makes extensive use of the Vienna* collection (Section 2.3), thus uses synergy effects introduced by LCSD (Section 3.4). Therefore, the actual device simulation tool, named ViennaMini, merely interfaces with these libraries and can thus focus on higher-level issues, such as orchestrating the simulations.

This fact represents the major motivation for applying LCSD in the first place, and thus ViennaMini can be seen as a *beneficiary* of such an approach.

Figure 4.9 depicts the library dependencies of ViennaMini. Note that the previously identified key components of a device simulator (Figure 4.2) map directly to the utilized Vienna* libraries. The mesh-related functionality is provided by ViennaGrid and ViennaMesh, respectively. ViennaGrid implements the mesh data structure and related traversal methods. ViennaMesh enables to generate and to adapt meshes via several meshing kernels. The material data is accessed via ViennaMaterials, providing flexible access to material parameters via different database backends. The symbolic math functionality is provided by ViennaMath, allowing to setup and manipulate mathematical expressions of the PDEs. The finite volume discretization scheme is implemented via ViennaFVM, performing the discretization of the PDEs. Also ViennaFVM assembles the system of equations $Ax = b$, and solves it via nonlinear solvers, in turn powered by exchangeable linear solver backends.

Section 4.3.1 gives an overview of the general simulator design. Section 4.3.2 discusses an approach to handle material parameters. Section 4.3.3 introduces a device object, holding the mesh data structure and additional meta information. Section 4.3.4 investigates the configuration object used to tune the simulation. Section 4.3.5 shows the ability to perform a series of simulations according to a contact parameter sweep, required for computing, for instance, the current-voltage characteristics. Section 4.3.6 discusses an approach for defining a flexible set of simulation setups. Section 4.3.7 deals with mesh-related tasks. Section 4.3.8 introduces the device template mechanism. Section 4.3.10 depicts several examples, underlining the capabilities of ViennaMini.

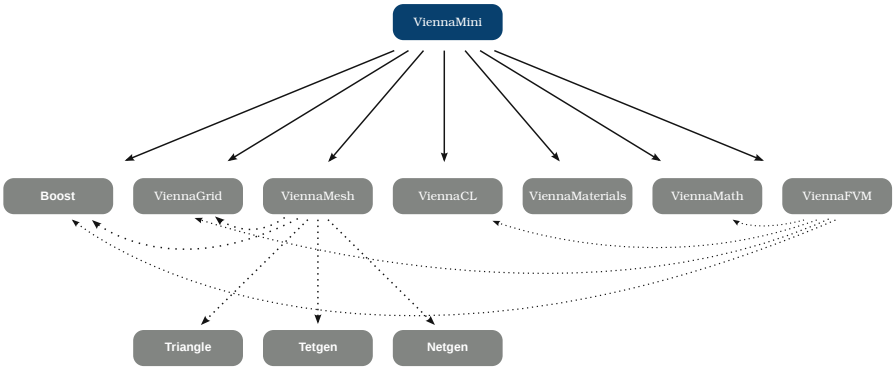


Figure 4.9: The dependencies of the device simulator ViennaMini are shown. The majority of functionality, such as finite volume-based discretization and assembly routines, is provided by the Vienna* collection and the Boost libraries. ViennaMesh additionally depends on external meshing tools, such as Triangle, Tetgen, and Netgen.

4.3.1 Design

Figure 4.10 shows the individual components of ViennaMini. The core of the simulator is its expandable *problem* facility, providing specific simulation setups, such as the basic semiconductor equations (Section 4.1) and related additional models. These problems use the ViennaMath library to setup the PDEs, and forward it, along with other information such as solver parameters, to the ViennaFVM library, taking care of the assembly and solution process.

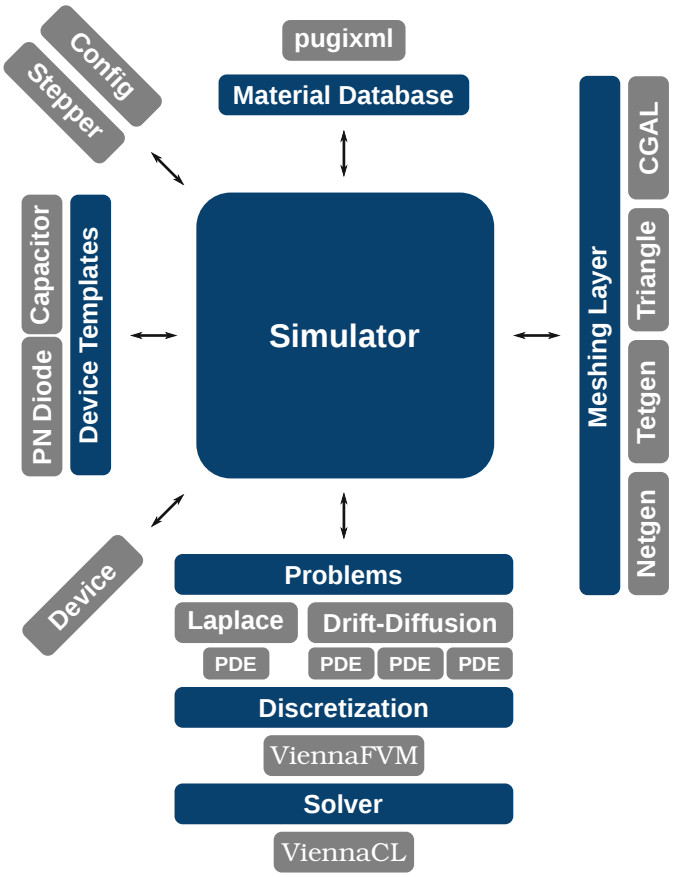


Figure 4.10: The currently available components of ViennaMini. The central simulator orchestrates the individual components, forming the actual simulation application. Blue components indicate exchange layers, whereas grey components relate to the actual kernels, such as *pugixml*. Note that additional kernels can be added, for instance, additional discretizations such as the FEM provided by ViennaFEM.

The device class stores a ViennaGrid mesh object in its state and allows to store and access additional meta information, such as the segment *roles*. A segment role associates device-specific part to a segment, like contact, oxide, or semiconductor. Also, an expandable device template class hierarchy enables to generate devices, which can be directly used for simulations. Among the currently implemented device templates are a two-dimensional pn-junction diode and a three-dimensional capacitor. The set of templates can be further extended by implementing additional devices, adhering to the device template interface imposed by the corresponding virtual class hierarchy. The following sections discuss the individual parts of ViennaMini in detail.

4.3.2 Material Database

The material database is governed by ViennaMaterials, enabling access to different tree-based database backends. In the following we focus primarily on an XML approach provided by the pugixml library.

The following snippet depicts an exemplary XML-based material database file, loaded at run-time.

```
<Materials>
  <Material>
    <Name>Silicon</Name>
    <Category>Semiconductor</Category>
    <Parameter>
      <Name>Bandgap</Name>
      <Value>1.107</Value>
      <Unit>eV</Unit>
    </Parameter>
  </Material>
</Materials>
```

The depicted XML setup is both highly flexible and readable, however, due to the tag-based nature it is not considered convenient for manual editing especially considering typical sizes of several hundreds of materials. This very fact underlines the importance of flexible database backends. In any case, additional categories and parameters can be easily added. Furthermore, the depicted approach can be extended to support different representations for values, for instance, additionally to the floating-point number a rational number can be stored. Storing different representations may be more suitable under certain circumstances, such as high-precision applications. Also, units are stored, enabling the interfacing application to extract the unit string and use it to form unit-aware physical quantities, providing automatic conversions using, for instance, the UDUNITS package [123]. Although not currently supported, tensor values can be used, by introducing a corresponding tensor XML tag, holding the tensor values. The material database can - upon access - use this particular tag to identify tensor valued data, and use appropriate import routines for reading the tensor values into a tensor object.

The access to data is implemented by using the path-based XPath query language. The values of the selected sub-trees can then be investigated as needed. Therefore, the use of a query language greatly enhances the flexibility of data access. For instance, the following query returns all semiconductor materials present in the database:

```
/Materials/Material[Category='Semiconductor']
```

Considering the previously depicted minimal material database, the following sub-tree is returned.

```
<Material>
  <Name>Silicon</Name>
  <Category>Semiconductor</Category>
  <Parameter>
    <Name>Bandgap</Name>
    <Value>1.107</Value>
    <Unit>eV</Unit>  </Parameter>  </Material>
```

The whole node is again returned in XML format. This enables further processing of the returned data using the same mechanisms, using relative query paths, e.g., using the path postfix `".//"` instead of `"/"`. This approach can be used to quickly partition and browse large databases.

However, in the case of a material database for simulations, the most common task is to directly access the numerical data values. An XPath query can be evaluated either to a node but also directly to a string, boolean, or a floating-point value. pugixml provides such a mechanism with it's `evaluate_number` function. For instance, using this evaluation function the following query yields directly a `double` value holding the requested parameter when executed on the previously generated sub-tree.

```
".//Material[Name='Silicon']/Parameter[Name='Bandgap']/Value";
```

Aside of storing numerical values, we can also store mathematical expressions inside the XML data. These mathematical string-expressions can be evaluated, for instance, by interfacing with the Lua library [124].

The fact that users[2] might change the XML input data, such as input files, introduces the need to verify the validity of the XML document. For instance, a material database implementation might expect a setup as introduced above. However, in the problematic case where an advanced user chooses not to follow this setup by, for instance, introducing new tags, it is still valid XML data, but may clash with requirements expected by the database implementation. Therefore, the document type definition (DTD) can be used, providing a set of markup declarations that define a document structure. Automatic verification of the XML input data relative to a DTD can be achieved by, for instance, utilizing the libxml2 [113] library and its `xmlValidateDtd()` function.

The query-based access to material data is convenient, however, performing the query in computational intensive code blocks is not meaningful. For instance, accessing material data within the mesh element traversal used in assembly routines will result in a considerable performance hit. Considering meshes with 10^6 elements would thus translate to 10^6 queries, resulting in considerable look-up times [125]. Therefore, the queries should be conducted outside of such expansive loops, or - if this is not possible - a cache system, such as *least recently used* [126], can be implemented and coupled to the query routines. Such a caching system would avoid unnecessary look-ups in the database and thus increase overall data access efficiency.

In general, material data is accessed from various simulation components. For instance, the device - discussed in the following - requires material data to extract parameters such as the relative permittivity for a given material name.

4.3.3 Device

As already indicated, a device holds the actual mesh object as well as additional meta information. The mesh data structure is provided by ViennaGrid, supporting different mesh configurations via distinct types. As the mesh configuration is not known during compile-time, a dynamic mesh storage object - based on a Boost Variant [53] data structure - is used to hold the actual mesh. This particular data structure can take on several different but fixed types, e.g., one- or two-dimensional simplex meshes: Instead of using the actual mesh types for the variant, smart pointers are used, provided by the Boost Smart Pointer library [51].

[2] In this context a user refers to an advanced user who wants to extend or alter the material database, contrary to an end user who does not interact with material properties at all and therefore requires no knowledge of XML.

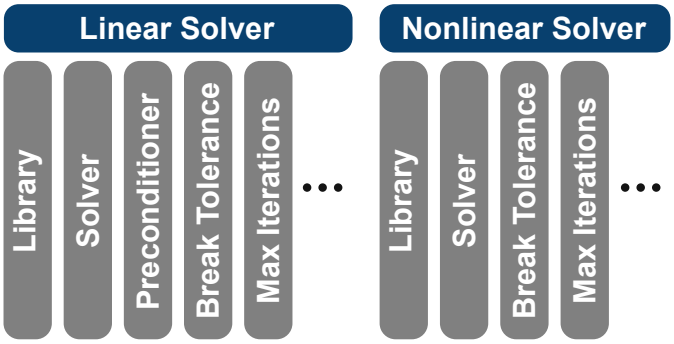


Figure 4.11: The configuration class holds several parameters essential for the simulation, such as linear and nonlinear solver parameters.

Smart pointers provide a safe - automatic memory deallocation - and a flexible way of destroying/recreating objects, because of their pointer mechanism, during run-time. The set of supported mesh types can be extended by adding additional types. This variant-based approach allows the device to wrap a dynamic layer around static objects, favoring, for instance, GUI applications.

A device supports methods to scale the contained mesh and to assign roles to segments. This identification allows the device to derive further meta information automatically, such as the ability to assign doping profiles or permittivities appropriately. Also, a device enables to assign material names to segments. This mechanism is coupled with the material database, for instance, to access the relative permittivity (ε_r) of a particular material, multiply it with the absolute permittivity constant (ε_0), and assign the result to the appropriate mesh elements. These auxiliary methods allow to attach meta information to a bare mesh, elevating it from a physical meaningless entity to a device, providing a plethora of properties via the device's interface methods. For instance, the problem classes (Section 4.3.6) uses these informations to assign boundary conditions to the contact segments.

4.3.4 Configuration

The configuration enables to collect parameters relevant for the simulation process (Figure 4.11). For instance, the maximum number of iterations and the break tolerance values for the linear and nonlinear solvers are stored. The individual simulator components use these configuration parameters to drive the simulation process and the problem classes (Section 4.3.6) forward the solver parameters to the solver kernels, such as ViennaCL's conjugate-gradient solver.

Replacing individual simulation components might alter the set of configuration parameters. A typical example would be that by using a different linear solver, other preconditioners are supported changing the simulator's configuration options. This fact introduces the need for each component to provide its configurable parameters in a unified manner to the frameworks configuration mechanism, such as via an associative container, mapping configuration options to corresponding default values which can be updated.

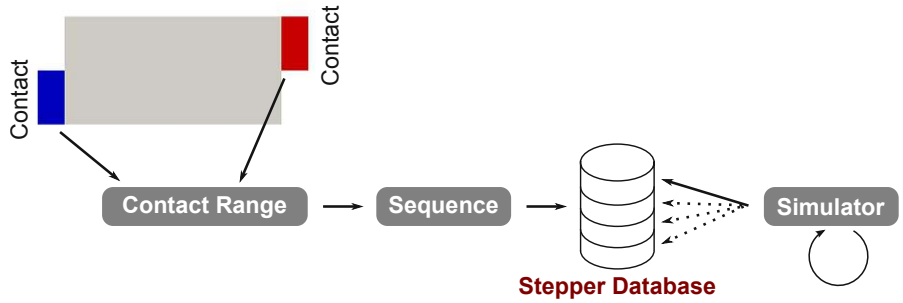


Figure 4.12: A voltage stepper allows to assign terminal contact potential ranges to contact segments. Each range consists of a start, end, and delta value. Based on a range, a sequence of contact values is computed and stored in the database.

4.3.5 Stepper

The stepper component enables to setup a set of simulation configurations according to varying step values. A typical example would be a voltage stepper, which is used to setup a set of simulation configurations according to varying contact values, being the most fundamental stepping mechanism required to determine device characteristics.

In particular, for each contact a sequence of values has to be supported (Figure 4.12). The simulator (Section 4.3.9) uses this mechanism to update the boundary conditions after each simulation, followed by a new simulation, until the stepper database has been completely processed.

Aside from voltage stepping, other stepping targets are of interest. Most importantly current stepping, deriving a set of simulation setups for a sequence of current values at specific Dirichlet contact segments. Also, stepping device geometries and doping profiles are relevant, triggering a new simulation for each element of a set of geometry configurations or doping profiles, respectively. Where the first usually requires a remeshing step - the original doping profile has to be redistributed to the altered mesh - to incorporate the changes in the device's geometry for each new simulation step, the latter applies different doping profiles to the device for the individual simulation steps.

4.3.6 Problem Classes

A problem class enables to group all relevant implementations required for a specific simulation setup into a single entity. This approach is considered the core of the simulator, which implements its expandable modeling facility via a virtual class hierarchy. This hierarchy enables to specialize the assembly and solution steps according to problem-specific requirements. Each problem specialization ensures that the required boundary conditions and initial guesses are assigned, the PDE objects are prepared including optional additional models, the linear solver backend is set up, the discretization method is selected, and ultimately the problem is assembled and solved (Figure 4.13). The simplest problems are the Laplace problem and the DD problem, which are discussed briefly in the following.

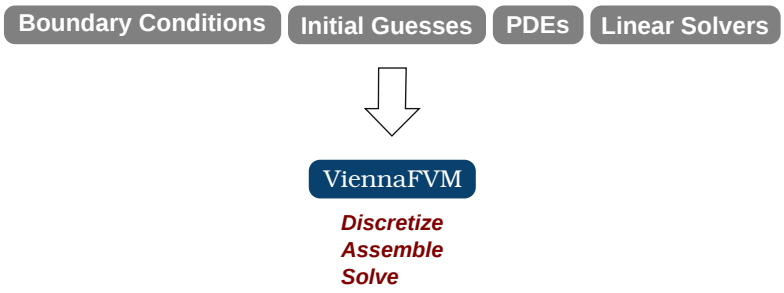


Figure 4.13: Each ViennaMini problem specialization is responsible for assigning the corresponding boundary conditions and initial guesses. Also, the PDEs must be defined and forwarded together with a linear solver backend to ViennaFVM, responsible for discretizing, assembling, and solving.

The Laplace Problem

The Laplace problem solves Laplace's equation for the electrostatic potential via the FVM. This case is implemented by assuming that the simulation domain is charge free, thus Poisson's equation is simplified to:

$$\nabla \cdot (\varepsilon \cdot \nabla \varphi) = 0 \tag{4.6}$$

Note that we keep the permittivity on the left-hand side to investigate jumps in the solution due to material transitions. Overall, such a Laplace problem is relevant for, e.g., capacitance simulations.

Due to the utilization of a symbolic math kernel provided by ViennaMath, the Laplace equation object is implemented in a straightforward manner.

```
1 equation laplace = make_equation(div(eps*grad(phi)),/* = */0);
```

The first parameter of `make_equation` resembles the left-hand side, whereas the second parameter holds the right-hand side of the equation. In this case, the differential operators `div` and `grad` are used, as well as two quantity objects holding data values for each mesh element. Note the strong resemblance of the original mathematical formulation (Equation 4.6) and the implemented equation object.

The mesh element-associated permittivity values (`eps`) are defined according to material parameters assigned to each segment of the mesh. This requires access to material-based parameters, as the device merely provides the material name, the actual relative permittivity has to be extracted from the material database, though. Accessing the permittivity for a given material name is based on the material database, via predefined queries, customized according to the actual material.

The boundary conditions are assigned using the segment roles provided by the device's auxiliary methods. More concretely, segments tagged as contact, are assigned Dirichlet boundary conditions according to the contact potentials provided by the stepper class (Section 4.3.5). Non-Dirichlet contacts are implicitly treated as Neumann boundaries by the utilized ViennaFVM assembly routines. The linear solver kernel is defined via ViennaFVM and configured according to the configuration parameters (Section 4.3.4).

The Drift-Diffusion Problem

The DD problem implements the previously introduced basic semiconductor equations (Section 4.1), consisting of three coupled PDEs in a straightforward and expressive manner for the stationary case.

```
1 equation poisson = make_equation(
2   div(eps*grad(phi)), /* = */ q*(n-p-ND+NA);
3 equation continuity_n = make_equation(
4   div(mu_n*VT*grad(n)-mu_n*grad(phi)*n), /* = */ R);
5 equation continuity_p = make_equation(
6   div(mu_p*VT*grad(p)+mu_p*grad(phi)*p), /* = */ R);
```

The equations follow the original formulations as shown in Equations 4.1-4.5. This approach is reasonable for a small set of models, however, with growing model numbers, such an approach quickly becomes confusing and thus prone to errors. As an alternative each model could alter the equation objects accordingly via a pipelined process, meaning that an initial PDE enters a pipeline of model-based adaptations, according to the end user's model selection. Such an approach would be more expandable and maintainable, as each model implementation performs local adaptions to the equation objects. Concerning the mobility and the recombination, the corresponding variables (`mu_n`, `mu_p`, and `R`) can be associated with models evaluating mesh element-specific values.

The current implementation of the nonlinear solver required for the DD problem utilizes the simple yet effective Gummel's method [127] as provided by ViennaFVM. Among the advantages of this particular nonlinear solver is the fact that it is easier to implement as compared to Newton's method, e.g., no Jacobian matrix has to be assembled. However, a Gummel solver offers stability issues for certain problems, such as high bias scenarios [1]. Overall, Gummel's method is a reasonable nonlinear solver technique for the investigations presented in this work, due to the focus on software design and especially as this particular solver is suitable for a considerable number of problems.

4.3.7 Mesh Generation

ViennaMini utilizes the mesh generation and adaptation facilities of ViennaMesh, providing a unified interface to several meshing kernels. For instance, in the following a mesh is generated from an input geometry using the Netgen kernel.

```
1 algorithm_handle mesher=algorithm_handle(new netgen::algorithm());
2 mesher->set_input("cell_size", 0.1);
3 mesher->set_input("delaunay" , true);
4 mesher->set_input("default"  , input_geometry);
5 mesher->run();
```

The Netgen kernel is instantiated (Line 1), meshing parameters are set in a unified manner (Lines 2-4). The `cell_size` parameter indicates the maximum *size* of a mesh element, which's meaning depends on the meshing backend. For example, CGAL relates the size parameter to the circumradius of a tetrahedron, Tetgen maps it to the volume of a tetrahedron, whereas Triangle interprets it as the area of a triangle.

Consequently, the `cell_size` parameter stores a value without an associated physical unit due to the polymorphic nature. In this case the mesh element is abstractly referred to as a cell[3]. The size of a mesh element is a typical basic parameter of mesh generation tools, as it provides a convenient method to guide the general resolution of the mesh. The `delaunay` parameter indicates a quality criterion, i.e., whether the mesh conforms to the Delaunay property [54]. This is a most common flag which is usually supported by all mesh generation tools for the field of CSE. Obviously not all mesh generation tools provide the same set of meshing properties, as, for instance, some tools provide different meshing algorithms, such as Triangle [62]. However, as already indicated most fundamental parameters, such as the cell size and the Delaunay property are potentially supported by the majority of tools and can thus be set in a unified manner, as depicted in the previous code sample. The `default` property indicates the input geometry object which must be discretized by the meshing tool. Finally, the meshing algorithm is executed (Line 5). Note that changing the meshing kernel to, for instance, Tetgen, solely requires to change the corresponding algorithm in Line 1 accordingly.

Generating meshes is used in the device template mechanism, where a customized device geometry has to be meshed.

4.3.8 Device Templates

A device template mechanism is implemented, enabling devices with customizable geometries to be simulated without requiring an externally provided mesh or a device holding a doping a profile generated by a process simulator. The implemented approach interfaces with the device, configuration, and problem component to prepare a ready-to-simulate device. Each template implementation holds a device object, which is being prepared for simulation by the device template's generation method. The device is specialized according to the template type, for instance, a device holding a two-dimensional triangular mesh. Also, the device is used to assign segment roles and additional meta information, such as manually assigned segment-based doping values. The configuration component is specialized according to the nature of the device at hand. For instance, a three-dimensional DD-based device might need specialized solver parameters, such as a specific preconditioner. Such specialized solver parameters can either be derived from time-consuming trial and error investigations or via automatic approaches, such as heuristical methods where the simulator determines optimal properties based on the evaluation of a set of possible property combinations. Overall, a device template must preset the configuration parameters accordingly, to provide a reasonable simulation configuration to ultimately enable the subsequent simulation step to converge.

[3] A cell is a mesh element with the maximum topological dimension within the respective mesh. Its geometrical equivalent depends on the cell topology and the dimension of the topological space [77]. For instance, in a two-dimensional simplicial mesh, a cell refers to a triangle.

With respect to defining the device geometry, each template provides an associative container holding a set of device-specific geometry parameters, such as the individual point vectors describing the geometry or - more conveniently - the oxide thickness, as required by for instance MOSFET devices. Obviously the set of parameters supporting customization is template specific, each implementation can offer different options. This flexibility is vital as the plethora of potentially available devices requires entirely different geometry properties. For instance, a bipolar transistor does usually not have an oxide, thus providing an oxide thickness parameter is not just wasted but indeed would be wrong. The fact that the parameters may be customized prior to the actual generation, gives rise to the design rationale to decouple the generation step from the setup step. Therefore, the parameters can be customized prior to the generation step, allowing to non-intrusively change the device geometry of the generated device.

Figure 4.14 depicts the essential steps of a device template implementation. With respect to the implementation, a virtual class hierarchy is utilized, allowing to extend the set of supported device templates by adding additional device template specializations. Due to the generic interface of this class hierarchy, devices of arbitrary nature are supported.

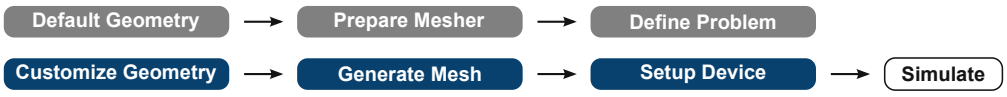


Figure 4.14: The essential steps of a device template implementation are shown. Grey components indicate preparation steps, whereas blue components relate to tasks triggered by the generation of the device. When the device is generated, it can be immediately simulated.

4.3.9 Simulator

As already indicated, the central simulator orchestrates the overall simulation, as is shown in Figure 4.15. Device objects are generated according to imported meshes, by performing optional mesh adaptation and doping interpolation steps provided by ViennaMesh and ViennaGrid. In case a device template is used, the ready-to-simulate device is merely accessed and forwarded to the simulation routine. This particular routine uses the stepper component to iteratively update boundary conditions and perform the problem-specific simulations.

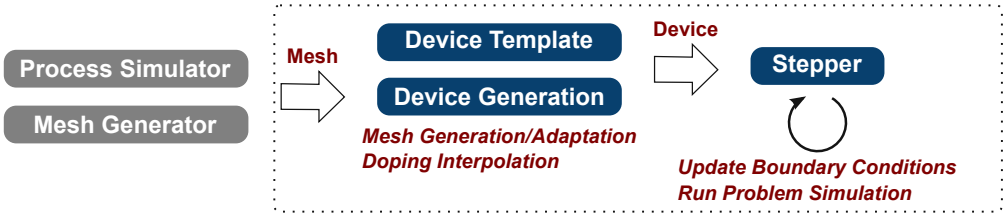


Figure 4.15: The key steps (blue) of the simulator (dotted box) component are shown. Grey components are external tools. Externally provided meshes and doping profiles are processed and corresponding devices are generated. In case a device template is used, the device is merely generated and forwarded to the simulation routine, using the stepper component to iteratively update the boundary conditions and perform the problem-specific simulation.

4.3.10 Examples

This section depicts several examples to show the usability of the introduced device simulator approach. The examples are chosen to depict the multi-dimensional support as well as the device template mechanism, the different simulation problems, and the stepping facility. Implementation details as well as simulation results are shown.

One-Dimensional Capacitor

This section discusses a one-dimensional capacitor device, solving the Laplace problem (Section 4.3.6). This particular case has been chosen to depict the support for one-dimensional devices, usually required for developing and debugging more advanced models. Therefore, this rather trivial device is required to be supported by every device simulator, before delving into more complicated models.

The device consists of five segments; two metal contact segments are attached to either side of a silicon dioxide-silicon-silicon dioxide (SiO_2-Si-SiO_2) structure, both assigned as Dirichlet contacts. As the implementation of the Laplace problem[4] keeps the permittivity on the left side of the equation (Section 4.3.6), the potential reflects the transition between the materials, as shown in Figure 4.16. The potential drops more significantly in the oxide segments than in the middle semiconductor segment.

Figure 4.16: The potential (V) over the spatial dimension (m^{-6}) of a one-dimensional capacitor is depicted. Dirichlet boundary conditions have been applied to the left (1.0V) and the right (0.0V) metal contact. Note the potential transitions at the material interfaces due to the different relative permittivities of SiO_2 and Si.

[4] In case the semiconductor material is doped, the thus arising space charge requires the solution of the Poisson equation (Equation 4.1).

Two-Dimensional PN Diode

This section shows the simulation of a two-dimensional pn-junction diode. The DD problem (Section 4.3.6) is solved for a set of contact potentials. This particular example has been chosen to depict the support for two-dimensional devices as well as the evaluation of device characteristics.

The device consists of four segments, where two metal contact segments are attached to either side of a p-Si-n-Si structure (Figure 4.17). The p-Si offers a constant donor and acceptor doping of 10^5cm^{-3} and 10^{15}cm^{-3}, respectively. The n-Si offers a constant donor and acceptor doping of 10^{15}cm^{-3} and 10^5cm^{-3}, respectively.

The device characteristics is computed by applying a constant cathode contact potential by simultaneously varying the anode contact potential, ranging from -1.0V to 1.0V, with a stepsize of 0.05V (Figure 4.18). In forward and reverse operation a maximum current of 2.3A and 1nA is computed, respectively. Figure 4.19 depicts the computed potential distributions for the reverse, equilibrium, and forward case. In the forward case, the polarity of the anode contact is switched.
Figure 4.20 depicts the computed electron concentration distributions for the reverse, equilibrium, and forward case. Where in the reverse case, the electrons retract toward the cathode contact, in the forward case the electrons are distributed over the entire device. Figure 4.21 depicts the computed hole concentration distributions for the reverse, equilibrium, and forward case. Where in the reverse case, the holes retract toward the anode contact, in the forward case the holes are distributed over the entire device.

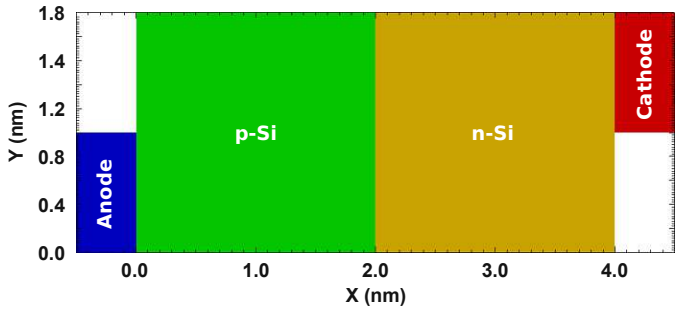


Figure 4.17: The setup of the two-dimensional pn-junction diode; Each color denotes a different device segment. The p-Si and n-Si offer a constant acceptor and donor doping of 10^{15}cm^{-3}, respectively.

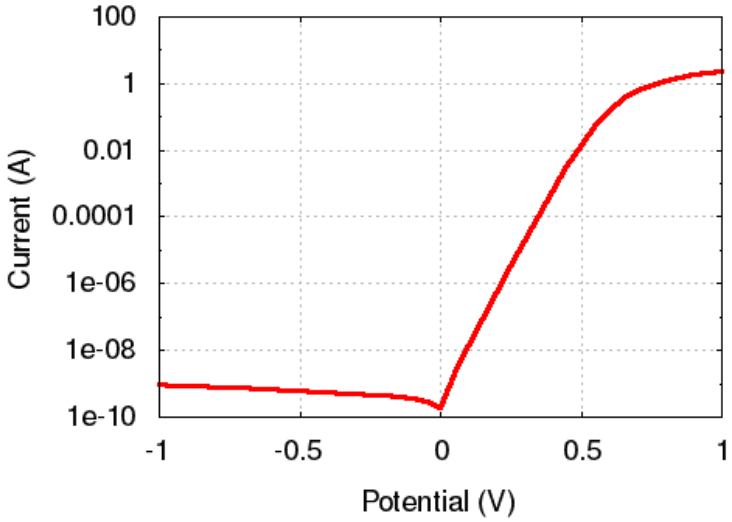
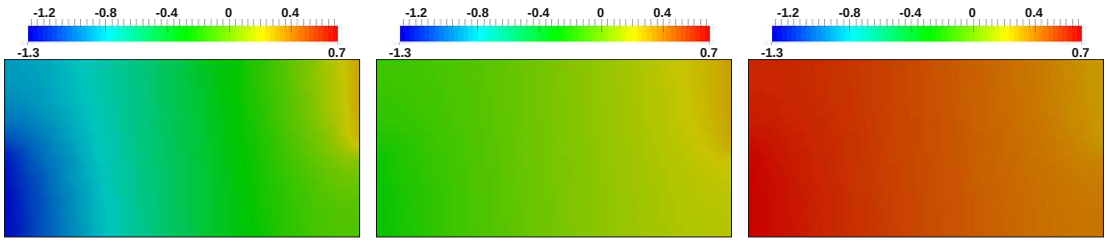
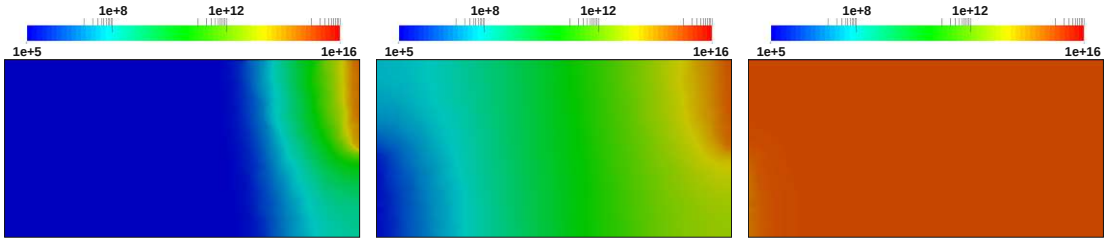


Figure 4.18: The IV-characteristics of a two-dimensional pn-junction diode; the contact potential of the anode contact has been gradually increased from negative to positive voltages relative to a constant cathode potential (0V). For forward bias (positive potential values) the diode is conductive, whereas for negative bias (negative potential values) the diode is non-conductive. Note the current saturation (> 0.6V) induced by high injection effects.



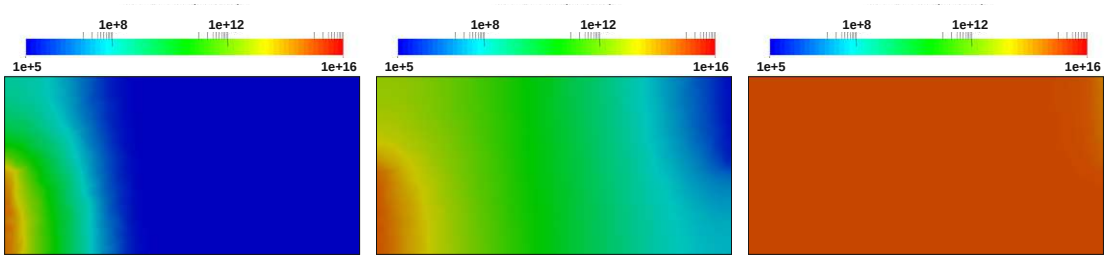
(a) Reverse (b) Equilibrium (c) Forward

Figure 4.19: The potential distributions (V) in reverse (**left**), equilibrium (**middle**), and forward (**right**) mode of a two-dimensional pn-junction diode are shown. The contact segments have been removed to ensure proper color mapping. Due to the builtin potential the potential distribution is shifted. In forward mode, the anode contact switches polarity.



(a) Reverse (b) Equilibrium (c) Forward

Figure 4.20: The electron distributions (cm^{-3}) in reverse (**left**), equilibrium (**middle**), and forward (**right**) mode of a two-dimensional pn-junction diode are shown. The contact segments have been removed to ensure proper color mapping. Where in reverse mode the electrons retreat towards the cathode, in forward mode the electrons populate the entire device.



(a) Reverse (b) Equilibrium (c) Forward

Figure 4.21: The hole distributions (cm^{-3}) in reverse (**left**), equilibrium (**middle**), and forward (**right**) mode of a two-dimensional pn-junction diode are shown. The contact segments have been removed to ensure proper color mapping. Where in reverse mode the holes retreat towards the anode, in forward mode the holes populate the entire device.

Three-Dimensional FinFET

This section shows the simulation of a three-dimensional symmetrically sliced Si-based FinFET device, based on solving the DD problem (Section 4.3.6). This particular example has been chosen to depict the support for three-dimensional devices.

Figure 4.22 depicts the device setup. The source and drain region are set at a constant donor doping of 10^{18}cm^{-3}, whereas the bulk region is set at a constant acceptor doping of 10^{14}cm^{-3}.

The device has been simulated in its active state, by setting the gate and drain contact potential to 0.5V as well as the source and bulk contact potential to 0.0V (Figure 4.23). As can be seen from the results, the electrons gather primarily under the gate contact, forming a conducting channel from the source to the drain contact.

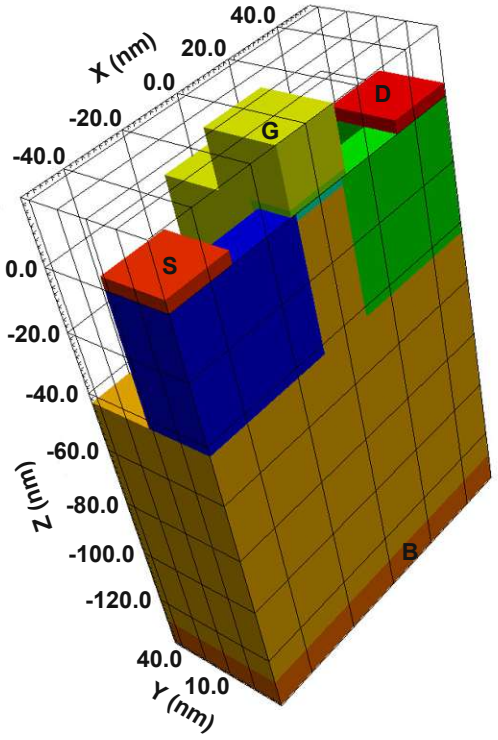
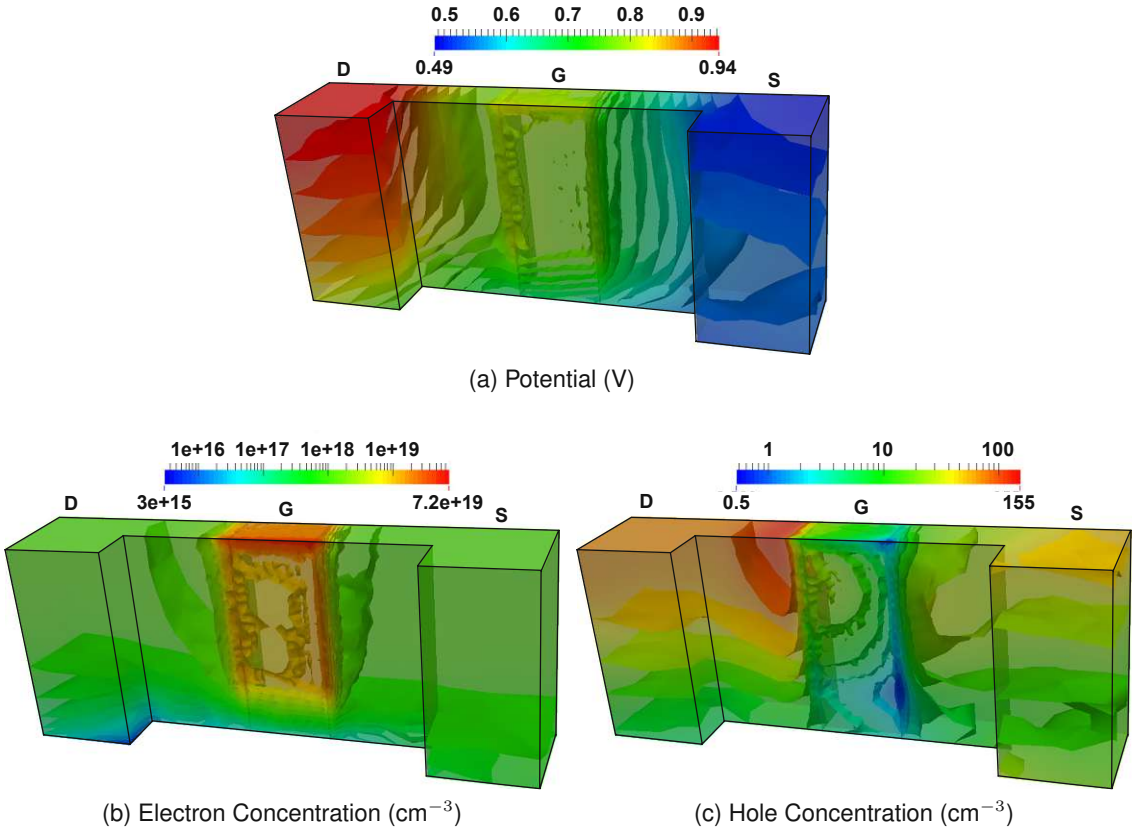


Figure 4.22: The setup of the three-dimensional FinFET device; Each color denotes a different device segment. **S**, **D**, **B**, and **G** refer to the source, drain, bulk, and gate contacts. The source (blue) and drain (green) region offer a constant donor doping of 10^{18}cm^{-3}. The bulk (brown) region offers a constant acceptor doping of 10^{14}cm^{-3}.



(a) Potential (V)

(b) Electron Concentration (cm^{-3})

(c) Hole Concentration (cm^{-3})

Figure 4.23: The potential, electron, and hole distributions of an active FinFET device; The gate and drain contact potential is set to 0.5V, whereas the source and bulk contact potential is set to 0.0V. The contact, oxide, and bulk segments have been removed for the sake of improved visualization. Iso-surfaces have been added to depict the behavior in the interior of the device. A conducting channel is formed under the gate as can be seen from the increased electron concentrations.

Chapter 5

Component Execution Framework

A component execution framework applies CBSE concepts, allowing to setup an application based on reusable components by connecting them accordingly. For instance, a FEM-based assembly component forwards the assembled equation system $Ax = b$ to a linear solver component which computes the solution vector x. As already indicated, the strength of such an approach is based on the growing potential for reusability - as more and more components are added to the set of available components - and on the significantly increased level of flexibility - as components can be exchanged in a straightforward manner. This fact is especially of interest to HPC applications, as a plethora of, for instance, linear solver libraries or finite element analysis libraries are available, consequently introducing the urge to exchange individual tools. Therefore, component execution frameworks merit special consideration for implementing simulation tools as compared to monolithic application approaches. Such frameworks thus provide an alternative to simulator software design. Most importantly, though, such frameworks can wrap - and thus reuse - already available functionality. Therefore, component execution frameworks further benefit from the LCSD approach similarly to the previously discussed device simulation framework (Figure 5.1).

In the following, an overview of today's HPC platforms (Section 5.1) is provided. Based on the established basis for computing targets the requirements and challenges of implementing a component execution framework for the general application in the area of CSE are discussed (Section 5.2). These discussed aspects give rise to the in the course of this work developed component execution framework which is introduced in detail (Section 5.3).

5.1 High Performance Computing

HPC combines all aspects of computationally intensive tasks required for conducting simulations in the area of CSE. This typically refers to utilizing more than one computing system to achieve this goal. In HPC hardware and software aspects with respect to conducting performance critical applications are considered. This fact introduces a highly diverse ecosystem of research areas. Typical HPC areas are parallel computing, network architecture, supercomputers, compilers, debuggers, operating systems, visualization, and data storage. In the following, hardware and software aspects of parallel computing are investigated, which we consider to be the basis for HPC.

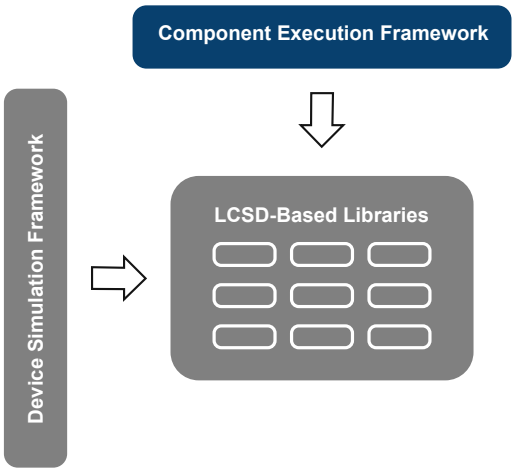


Figure 5.1: A component execution framework - similar to a device simulation framework - benefits from LCSD-based libraries as the available functionality is wrapped as components and can thus be used within the framework.

Historically the CPU's sole core was continually scaled down to increase the processing power by adding more and more transistors. Consequently, each new generation of CPUs automatically improved application performance. However, around the mid-2000s this approach gave way to introducing additional cores on the CPU die but with stagnating - or even decreased - clock frequencies to handle the thermal budget [100][128][129]. Clock frequencies for high-end consumer-level workstation CPUs saturated at approximately 3-4 GHz. Overall, this conceptual change was forced by physical limitations imposed by scaling limits, such as issues with heat dissipation, power consumption, and leakage-currents. Today's computing platforms entirely use multi-core CPUs. This development forces the software - driving the actual computation - to harness the parallel computational power to achieve an appropriate computational speed-up with new generations of hardware [130].

The ongoing scaling of semiconductor devices towards the low nanometer regime introduced the ability to provide multi-core CPUs as well as accelerators, such as graphics adaptors, to single workstations and even mobile devices. Therefore, parallel computing environments are no longer confined to highly expensive supercomputers, but are already available in consumer products. This introduces a dire need for software developers to have a solid grasp on parallel computing and the supported ways of harnessing the computational capabilities.

This section provides a detailed overview of parallel computing, more specifically:

- Shared-memory systems (Section 5.1.1)
- Distributed-memory systems (Section 5.1.2)
- Hierarchical (hybrid) systems (Section 5.1.3)
- Accelerators (Section 5.1.4)

5.1.1 Shared-Memory Systems

A shared-memory system consists of at least one multi-core CPU[1], sharing the memory available on the system, i.e., all CPUs can access the same physical address space. Examples for such a system are modern single- or multi-socket multi-core workstations. Although, these targets are shared-memory systems, and as such provide a single memory address space to the programmer, the supported memory access patterns vary fundamentally. The most commonly utilized patterns are uniform memory access (UMA) [100] and cache-coherent (cc) non-uniform memory access (NUMA)[2] [131]. UMA is typically used by single multi-core workstations, whereas (cc)NUMA is widely utilized by multi-socket systems. Basically, UMA allows to offer the same access performance to all participating cores and memory locations via a single bus (Figure 5.2).

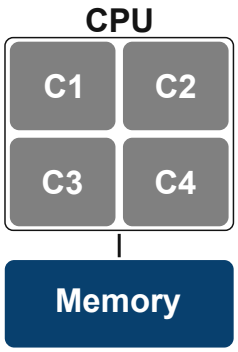


Figure 5.2: A UMA architecture provides each CPU core (C1-C4) the same memory access.

Therefore, UMA systems are also frequently referred to as symmetric multiprocessing systems. Such a symmetric approach does not reasonably scale with the introduction of additional CPUs, as the central bus is quickly saturated. This fact gave rise to the NUMA architecture (Figure 5.3). On NUMA systems memory is physically distributed, but logically shared. A NUMA system consists of several so-called *NUMA nodes*, typically representing a CPU and its local memory. The NUMA nodes are connected with interconnect *links*, such as AMD's HyperTransport or Intel's QuickPath technology. Consequently, latency and possibly bandwidth between the cores vary depending on the physical location (also called NUMA effects).

From a software point of view, usually - but not exclusively - thread-based approaches are utilized on shared-memory systems. Several approaches are available, such as OpenMP [132], Intel Cilk Plus [133], and Pthreads [134]. Although thread-based programming is considered to be rather intuitive and easy to get used to, it is exactly this feature which makes it hard to achieve reasonable scaling for high core numbers. One of the challenges is the shared-memory approach, i.e., all threads can access the same memory address space.

[1] Single-core CPUs are not mentioned, as they practically vanished from the desktop and HPC market, triggered by single-core scaling issues which forced the evasion towards multiple cores.

[2] The cc prefix refers to a mechanism which is responsible for keeping a consistent memory image in the individual distributed caches, meaning that a processor accessing a memory location receives the most up-to-date version of the data. For the remainder of this work NUMA refers to the cc versions, as these are predominantly applied due to significantly higher programmability, when compared to non-cc platforms.

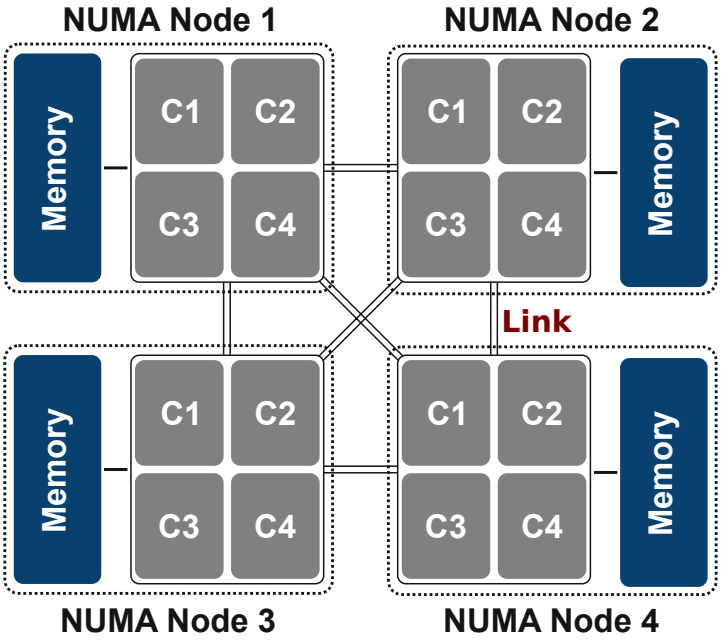


Figure 5.3: A NUMA architecture connects different NUMA nodes (Nodes 1-4) - typically multi-core CPUs (C1-C4) - via interconnect links (Link), to enable a single logically shared global memory. However, memory access times - and possibly throughput as well - are reduced when, for instance, Node 1 accesses memory associated with Node 3, due to overhead introduced by the link. This effect is typically referred to as NUMA effect.

Although convenient, it does not inherently force the developer to handle memory locality, thus NUMA effects easily arise, significantly reducing the scalability. Also, a typical problem with shared-memory approaches is over-utilization, meaning that significantly more threads conducting computations are executed concurrently than CPU cores are available. Obviously severe over-utilization, for instance, more than three times, leads to considerably reduced execution speeds, thus needs to be avoided.

5.1.2 Distributed-Memory Systems

A distributed-memory model connects different processors via a communication network (Figure 5.4). This model originates from a time where a CPU contained a single processing core. Such a setup is typically not found anymore in today's cluster systems due to the advent of multi-core CPUs. However, the model still serves well as an introduction to distributed computing, as the concepts are still applicable.

The defacto standard following the distributed-programming model is the MPI [135]. However, other less popular distributed parallel programming models are available, such as High Performance Fortran [136][137] and partitioned global address space [138] models like Co-Array Fortran [139] and Unified Parallel C [140]. MPI is implemented by various commercial and open source institutions, which make their individual implementation available as a library to be linked to the respective target application. It is important to note that using MPI is not restricted to distributed-memory systems, MPI can also be used on shared-memory systems (Section 5.1.1).

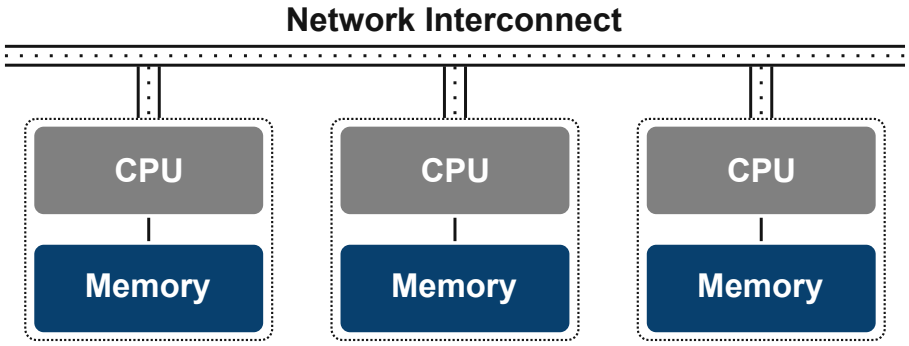


Figure 5.4: In a distributed-memory model, the memory is physically and logically distributed among the individual processing units (P1-P3), e.g., CPUs. Accessing data from remote memory locations requires to initiate a data transfer protocol, such as point-to-point communication provided by the MPI.

The fundamental difference between the shared and the distributed-memory programming model is the fact that in the distributed case processes are used instead of threads. As processes have only access to their own memory address space, accessing non-local data requires inter-process communication. The inter-process communication is implemented in MPI by so-called *messages*, hence the name message passing interface.

The overall performance of such a distributed system is not only determined by the compute capabilities of the individual processing units, but also by the network interconnect technology's speed, layout, and switching capabilities. Collecting, for instance, the partial result data from all processes imposes a considerable challenge for the network interconnect, as concurrent transmissions easily saturate the supported throughput. If communication or computation becomes predominant with respect to run-time performance, the parallel application is said to be communication bound or computation bound, respectively.

5.1.3 Hierarchical (Hybrid) Systems

Today's supercomputers and clusters are primarily hierarchical systems, also called hybrids. Such systems denote a mixture of systems based on different parallel memory models (Figure 5.5). The most prominent example are supercomputers with shared-memory nodes[3] interconnected via a network. Therefore, hybrids are neither purely shared nor are they purely distributed, further contributing to the challenge of parallel software engineering.

In fact, hybrids may be utilized by solely applying a distributed programming model, but hybrid models are supported as well, such as MPI in tandem with OpenMP. A typical approach is to utilize MPI for inter-node communication, and a shared-memory approach, e.g., OpenMP, on the individual nodes.

[3] A node is considered to be a self-sufficient computer which works in the collective, forming the supercomputer.

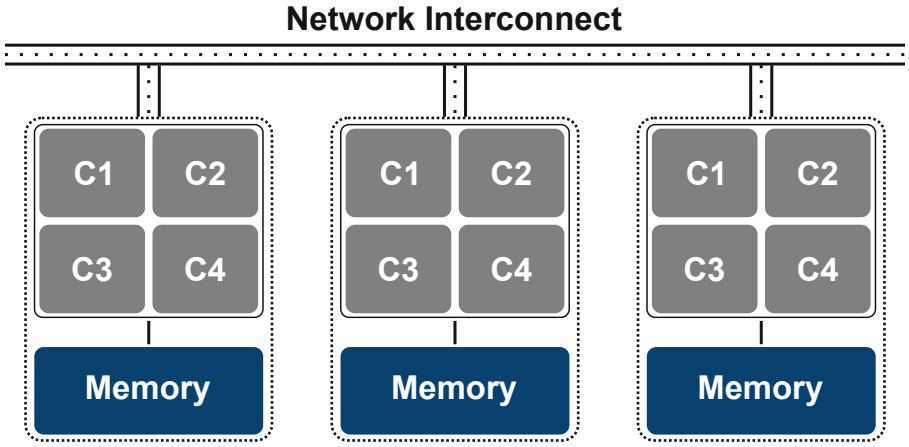


Figure 5.5: In hierarchical systems the distributed- and shared-memory model are combined. In the depicted case three UMA-based shared-memory nodes are connected via a network connection, however, NUMA systems can be used in a similar manner.

If the node's infrastructure offers a NUMA architecture, it may become a viable option to assign several MPI processes to a node and pin one, for instance, to each CPU socket[4]. For example, for each socket an MPI process can be assigned rather than using one MPI process for the whole node. On the right platform, such an approach reduces NUMA effects. Another considerable aspect is the node's memory-core ratio, where the sweet-spot[5] in today's systems is typically around 1-2 GB/core. Therefore, using one MPI process for each core might become challenging with memory intensive applications, as a node might not offer enough memory to accommodate a given problem. This becomes evident when considering a typical node setup used in current cutting-edge supercomputers, such as *Titan* [141], which is based on single-socket nodes offering a 16-core CPU and 32GB of shared memory. If 16 MPI processes are executed on such a node, each MPI process has access to merely 2GB of memory. A ratio of one is provided by the *Sequoia* supercomputer, offering 16 compute cores and 16GB of system memory on each node [142]. The observed node setups also depict an interesting fact based on the continued microprocessor scaling: a single CPU can now hold up to 16 compute cores, allowing to build cheaper single-socket nodes rather than the significantly more expensive multi-socket systems. This enables to remain cost efficient by simultaneously providing high core numbers. This trend towards higher core numbers per CPU is likely to continue [143]. At this point it becomes evident that due to the high degree of anisotropy of the available parallel target platforms, developing reasonably scaling software for such platforms is one of the major challenges today and especially in the future [144].

[4] Assigning an MPI process to a specific CPU core enables to distribute the MPI processes evenly throughout the available CPUs of a node.
[5] The term sweet-spot refers to the optimal ratio between computational capabilities and costs.

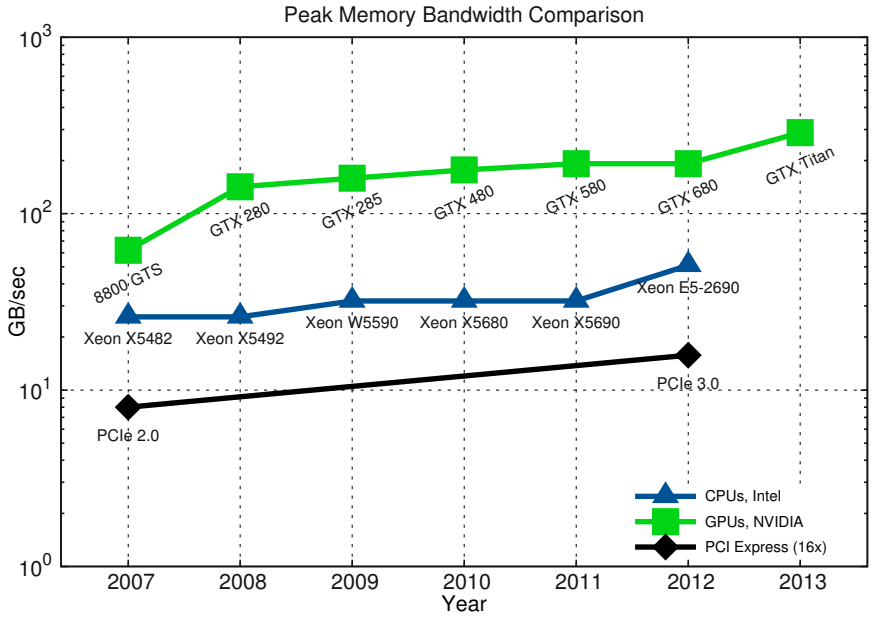


Figure 5.6: Memory-bandwidth between graphics-adaptors, CPUs, and PCI Express buses. Note that other accelerator cards - such as Intel Xeon Phi and AMD graphics adaptors - attached to PCI Express offer similar memory-bandwidths, thus for the sake of simplicity only NVIDIA GPUs are shown. Between 2007 and 2013 memory-bandwidth doubled for CPUs, whereas it tripled for GPUs primarily due to additional memory channels and broader buses. In 2012 the memory-bandwidths from CPUs and GPUs are roughly twelve times and more than three times faster compared to PCI Express, respectively.

5.1.4 Accelerators

At the time of writing, it is common practice to augment nodes of supercomputers with accelerators, such as graphics adaptors and so-called coprocessors. For instance, *Titan* utilizes NVIDIA Tesla K20 graphics processing units (GPUs) [141], where *Stampede* instead uses Intel Xeon Phi SE10P coprocessors [145]. In general, accelerators offer high core numbers and an excellent peak performance/watt ratio when compared to common shared-memory clusters [146]. However, one of the fundamental limitations is the connection with the host system via a connecting bus. The bandwidth and latency of such a bus system, e.g., PCI Express, are inferior compared to, for instance, the host's memory bus, resulting in a significant communication bottleneck. Figure 5.6 compares the memory bandwidths of typical CPUs with accelerators and the PCI Express bus [147]. To overcome this problem, processor vendors aim to integrate the accelerators on the CPU die, thus eliminating the need for a slow bus system [148]. However, if the applications can be tuned to minimize such communication in the first place, accelerators are capable of providing outstanding performance with simultaneously reduced energy consumption, heat dissipation, and costs.

From a software point of view, different approaches to program accelerators are available, which are typically also specific to the utilized accelerator. Prominent examples are CUDA and OpenCL. Where the first solely supports NVIDIA-based boards, the latter is not restricted to a specific board vendor, nor is it restricted to GPUs, as it can be utilized with CPUs as well. The computational capabilities of Intel accelerators - so-called coprocessors - can be harnessed with, for instance, Intel Cilk Plus, offering an *offloading* feature to perform computations on the coprocessor. Recently the OpenACC standard has been released, aiming for an increased level of convenience for programming highly-parallel compute targets. OpenACC focuses on a directive-based programming approach, similar to OpenMP.

The broad availability of programming models and different processor architectures of accelerators shows that this field is currently in flux. It is not clear at the moment which hardware and software approach will prevail. This, however, has considerable ramifications for software developers with respect to long-term support. Relying on a specific accelerator technology for boosting an application introduces a significant external dependence. Programming accelerators is very challenging and thereby allocates considerable resources. However, due to the rather unstable accelerator environment, it is not foreseeable whether a chosen accelerator platform will be supported in the future, thus bearing the risk of wasted development.

5.2 Requirements and Challenges

The availability of highly diverse parallel computing platforms promises significant speedups for HPC-focused applications, among the potential beneficiaries are component execution frameworks. A component execution framework involves several essential aspects (Figure 5.7). The component system enables to implement reusable components via defined interfaces. Data communication enables components to receive and forward data from and to other components, thus representing dependencies. Scheduler routines - supporting parallel execution - are responsible for determining the appropriate component execution order according to the dependencies of the utilized components. Finally, a configuration mechanism allows an appropriate configuration of the framework and the modules. Each of these essential parts are discussed in detail in the following.

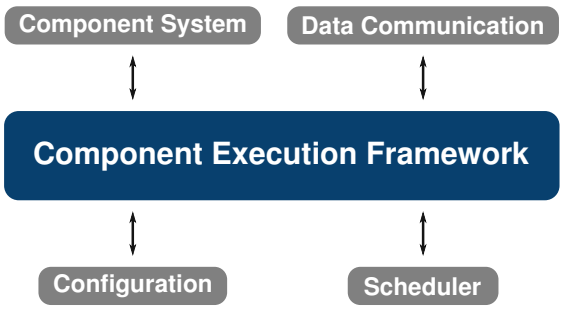


Figure 5.7: A component execution framework involves four essential aspects, those being a component system, a data communication layer, a configuration mechanism, and schedulers, driving the overall execution.

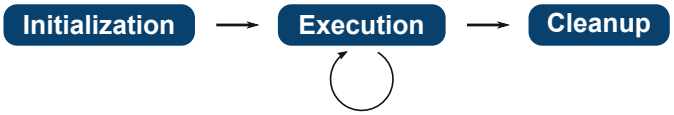


Figure 5.8: A component must provide a decoupled initialization, execution, and cleanup state. Where the initialization and cleanup state is only executed once by the framework, the component might be executed several times if, for instance, it is part of a loop.

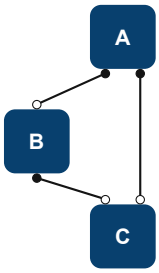


Figure 5.9: A component data communication mechanism must support an arbitrary number of input and output dependencies for each component (A, B, C).

5.2.1 Component System

A component system enables to implement reusable software entities, offering a defined interface. The interface must support an initialization, execution, and cleanup step (Figure 5.8). Where the first is essential to, for instance, allocate data structures for the subsequent step of execution, the latter enables to implement appropriate cleanup methods, such as releasing memory. The execution step contains the actual computation. In general, such a three-step approach allows for multiple executions of a plugin within a simulation introduced by, for instance, loops. Utilizing a three-step interface is a common approach in scientific frameworks, such as the ESMF [29]. Furthermore, the system must support the addition of new components as well as the convenient exchange of components, i.e., the exchange must not require recompilation.

It is particularly important to provide a high level or usability. Developers and advanced users will implement additional components and thus the interface has to be intuitive to facilitate straightforward implementations without requiring significant development efforts.

5.2.2 Data Communication

Strongly related to the component system is the ability of components to share data. The already discussed example of an assembly component forwarding A, b of an equation system $Ax = b$ to a linear solver sketches this requirement. As can be seen from this example, the communication mechanism has to support an arbitrary number of input and output dependencies for each component (Figure 5.9).

An important aspect is the support of arbitrary datatypes for the communication layer, to avoid confining the use of a fixed pre-defined set of types. This would otherwise limit the applicability - as potentially utilized external libraries typically offer their own data types - and also triggers costly data transfer operations to move data from unsupported datatypes to new objects based on supported types.

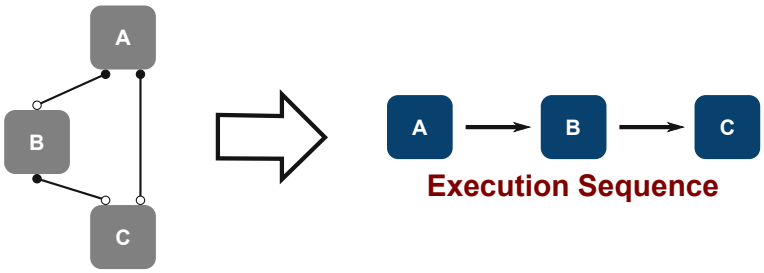


Figure 5.10: A component execution scheduler determines the execution sequence based on the dependencies.

Furthermore, it is required to associate the individual dependencies with an additional dynamic identifier, as using merely type-based identifiers might not offer a unique identification. For instance, a component might provide two matrices, both of the same type but holding different data. Merely using the type as identifier would thus not allow to distinguish the data and would consequently require manual connection.

The challenge of a data communication layer is primarily to provide a type-safe approach offering a convenient setup of input and output data communication among the components. In essence a generic mechanism has to be implemented, capable of linking input and output data communications of each component. This link in essence can be implemented by a generic object which needs to be aware of the data type transmitted via the link as C++ is a statically typed language. Implementing such a generic object capable of representing data of arbitrary type requires so-called *type-erasure* techniques, effectively removing the type information from an object to transform it into a *typeless* object. This transformation allows to handle objects of arbitrary types in a unified manner, for instance, a homogeneously-typed container, such as a `vector` can hold such typeless objects which originally differed in the type and thus would not have been possible to store them together in such a container. As a consequence arbitrary yet unforeseen data types can be processed. However, usually the challenge is not to lift the actual type from an object - basically removing the type information from the object - but to access the object from a type-erased state again appropriately. Therefore, the access mechanisms need to be as safe as possible, i.e., allowing for correct data retrieval. Other projects, like the ESMF [29] approach this problem by supporting a predefined set of datatypes for data communication. However, as already indicated, such an approach limits usability.

5.2.3 Scheduler

Ultimately the components must be executed. The sequence of execution, however, depends on the input and output data communication requirements of each utilized component. Scheduler mechanisms are required to determine the execution order in such a way that upon a component's execution all the required input data is available, i.e., the input data has been generated by other components (Figure 5.10).

Essential for usability is an automatic dependence resolution process. The end user should merely list the components to be utilized by the framework, but should not be burdened with the responsibility of establishing the connections manually. The framework must automatically - based on the data communication layer and the thus provided dependencies of each component - resolve the dependencies and compute an appropriate execution sequence.

Also the scheduler should support different serial and parallel execution methods. Most important is data parallelism based on the MPI, where each component works on a subset of the data. This particular parallelization mode is especially important to the field of CSE, due to the typically large simulation domains where a specific physical model is evaluated. Methods like domain decomposition are used to partition the data set into data chunks of comparable computational effort which are then processed in an MPI-based computing environment [100], such as hybrid computing clusters. Aside from data parallelism also task parallelism merits special consideration. In this particular execution the components are executed in parallel, meaning that, for instance, two components might be executed by two different processes. Among the application areas are wave front simulations [149]. However, a serial mode is also essential, as not all applications inherently support or favor a distributed memory model.

With respect to task parallelism, a mechanism has to be provided allowing to guide the automatic task scheduling. For instance, an advanced user and a developer has knowledge about the computational effort of a task and thus by extension about the estimated execution times. If the scheduler has knowledge about the expected computational load of each task, the scheduling can be adapted, for example, the task with the long run-time is processed on a separate process than the rest of the tasks, offering short run-times and ultimately improving parallel execution efficiency.

The challenges of scheduler implementations are primarily related to automatically determining a correct execution sequence especially in a parallel setting. In particular, the usability should be maximized whereas the code base should be minimized to favor maintainability. Also the task parallel scheduling has to be augmented with an optional task-specific *weight* factor, reflecting the individual execution run-time. In this regard, the challenge is to incorporate the advanced user or developer-provided task weights into the scheduling, either by hard-coding it into the component implementations or by using the non-intrusive configuration mechanism, which is discussed in the subsequent section.

5.2.4 Configuration

The component execution framework must be informed from the end users which of the available components to utilize for an execution. Also components might offer customizable parameters which can be adjusted prior to execution. Hardcoding the parameters into the source code would be inconvenient, as changes to the parameters would require recompilation, quickly becoming unbearable especially with growing component numbers.

Therefore, a run-time configuration mechanism is required providing both, the set of components to be utilized as well as forwarding component-specific parameters to individual components where they are accessed and utilized. Obviously, the framework must process this data to setup the components and use their individual dependencies to derive an appropriate execution order via the previously discussed schedulers (Section 5.2.3).

The primary challenge of such a run-time configuration mechanism is to provide an intuitive and easy-to-use approach, capable of handling both small and large sets of components. For instance, command-line arguments are unfeasible, as with growing component numbers the parameter list would become irritating and thus prone to errors.

For configuring applications usually either an input configuration file or a GUI is utilized. In this particular case, a configuration file approach merits special consideration as computing clusters typically only support command-line and job submission-based access. However, such an approach can also be paired with a GUI, allowing to generate configuration files in a convenient and decoupled manner, thus favoring framework utilization by end users.

5.3 The ViennaX Project

The FLOSS-based ViennaX framework facilitates the setup of flexible scientific applications by applying a CBSE (Section 3.3) approach, based on providing an execution framework for plugins [10][150][151]. In particular, ViennaX tackles the previously introduced requirements and challenges for a component execution framework (Section 5.2). The decoupling of simulations into separate components is facilitated by the framework's plugin system. Functionality is implemented in plugins, supporting data dependencies. Most importantly, the plugin system enables a high degree of flexibility, as exchanging individual components of a simulation is reduced to switching plugins by altering the framework's configuration data. Consequently, no changes in the simulation's implementation must be performed, thus avoiding recompilation and knowledge of the code base. Furthermore, decoupling simulation components into plugins also increases the reusability significantly. For example, a file reader plugin for a specific file format can be utilized in different simulations. Ultimately, the effort of changing parts of the simulation is greatly reduced, strongly favoring long-term flexibility and reusability.

5.3.1 General

ViennaX can be seen as a plugin execution framework. Available simulation tools or components can be wrapped by plugins and therefore reused. An application is thus constructed by executing a set of plugins. The input configuration file based on the XML contains information indicating the plugins to be utilized during the course of the execution. Additionally, parameters can be provided by this configuration file, which are forwarded to the respective plugins by the framework.

Plugins can have data dependencies, which are internally represented by a task graph and handled by the so-called socket system. Different scheduler kernels are available, focusing on different execution approaches, being serial, task parallelism, and data parallelism, respectively. These applications can be used to execute the graphs generated from the input XML file.

Figure 5.11 schematically depicts the general execution flow of the framework. Plugins are implemented and compiled as dynamic shared objects (DSOs), which are forwarded to the framework's application. In addition to the plugins, the input configuration file is passed to the application. The schedulers automatically generate and execute the task graph according to the data dependencies. The intended target platforms are workstations or clusters, which are supported by different distributed scheduler kernels based on the MPI. More specifically, the Boost MPI Library [49] is utilized to support distributed-memory parallelization (Section 5.1.2). Our approach does not wrap the parallel execution layer of the target platform like MPICH [152].

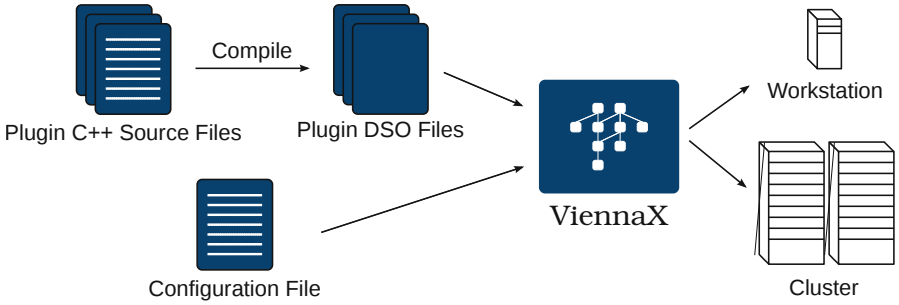


Figure 5.11: Schematic utilization of the component execution framework. C++ source files modeling the plugin concept are compiled into DSOs. The DSOs as well as the configuration file are loaded into the framework. The plugins are loaded during run-time and based on the dependencies a task graph is generated. The plugins are executed according to the dependencies, until the graph has been processed.

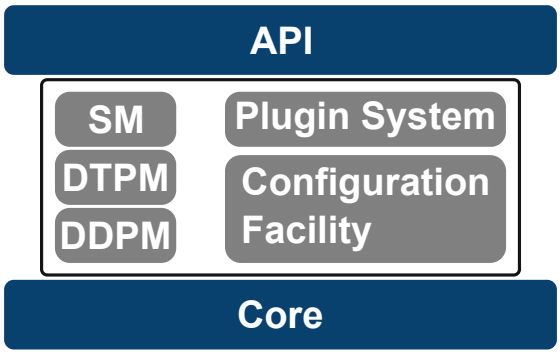


Figure 5.12: Design of ViennaX. An API provides developers access to the supported different scheduler kernels, being SM, DTPM, and DDPM. Additionally, the plugin system, and the configuration facility can be accessed externally. The core part provides fundamental functionality utilized throughout the framework, such as a task graph implementation.

As such, the framework is executed as a typical application utilizing the respective parallelization library. For instance, to execute an MPI capable scheduler application, the following expression is used.

```
mpiexec -np 4 ./vxscheduler config.xml plugins/
```

In this case the `mpiexec` command spawns the execution of four instances. `vxscheduler` relates to the application, whereas `configuration.xml` refers to the XML input file holding the required information to build the task graph. The final parameter `plugins` refers to the directory path, containing the plugins to be utilized during the execution.

Different scheduler kernels, those being the serial mode (SM), distributed task parallel mode (DTPM), and distributed data parallel mode (DDPM) scheduler, as well as the plugin system and the configuration facility are accessible via an API (Figure 5.12). The API enables software developers amongst others to implement or adapt schedulers. The design of the framework allows for different task execution modes implemented by the respective scheduler kernels to support, for instance, different parallel task graph execution strategies.

	Graph Execution	Plugin Execution
SM	serial	serial/shared-memory
DTPM	distributed	serial/shared-memory
DDPM	serial	distributed

Table 5.1: Overview of graph and plugin execution modes supported by the component execution framework.

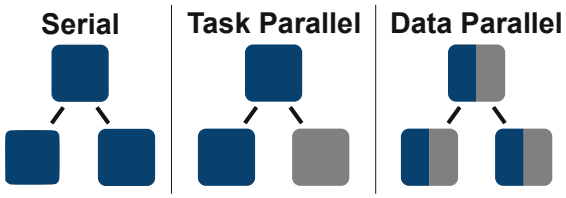


Figure 5.13: Different task graph execution models, where each vertex of the graph represents a plugin. Grey and white shaded plugins denote different compute-units, e.g., MPI processes. In serial mode, one compute-unit executes all the tasks but only one task at a time. In task parallel mode, different compute-units are responsible for subsets of the task graph. In data parallel mode, each task is executed by every available compute-unit, where each compute-unit processes only a subset of the data.

Table 5.1 discusses the available scheduler kernels. The SM-based kernel processes one plugin at a time, where the individual plugins run either serial and/or parallel shared-memory-parallelized implementations restricted to a single process, such as OpenMP. The DTPM kernel models the task parallel concept in an MPI context, where plugins are executed in parallel by different MPI processes, if the respective dependencies are satisfied. Consequently, applications with parallel paths in the graph can benefit from such a scheduling approach, such as the already indicated wave front simulations [149]. Finally, the DDPM kernel allows for a data parallel approach, where, although each plugin is processed consecutively, the plugins' implementation follows an MPI-based parallelization approach. Such an approach allows, for instance, to utilize an MPI-based linear solver component within a plugin, such as PETSc [122]. Figure 5.13 schematically compares the principles of the different execution modes, by mapping components to vertices[6] of a graph.

The currently implemented parallel scheduler focus on the distributed MPI. To better support the ongoing development of continually increasing core numbers per computing target, scheduler kernels utilizing shared-memory parallelization approaches are planned for future extensions. These future extensions are supported by the introduced naming scheme for the scheduler kernels as well as by the applied modular kernel approach.

[6] A vertex is a topological element of dimension 0. Its geometrical equivalent is a point.

5.3.2 Plugin System

This section discusses the design and implementation of ViennaX's plugin system. Figure 5.14 depicts the setup and exchange of a plugin. If the process of interchanging plugins is compared to the one of conventional simulation tools, it becomes clear that the conventional approach requires actual coding, and as such in-depth knowledge of the implementation at hand. For obvious reasons, this fact impedes the implementation of changing functionality. With the plugin-based approach, the exchange can be realized conveniently by only adjusting the input configuration data accordingly.

In the following, a factory [153] implementation is discussed, allowing to register and to load plugins in an automatic manner. Also the utilized plugin interface is introduced as well as the communication layer.

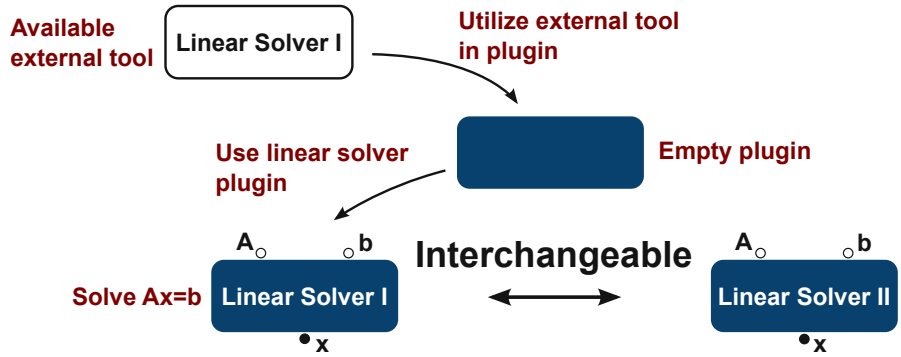


Figure 5.14: A plugin can be used to wrap available functionality, like linear solver implementations. Due to the abstraction mechanism provided by the socket input/output dependencies, plugins can be exchanged by other plugins. In this case, a linear solver implementation provided by ViennaCL as depicted in Section 5.3.3 is interchanged with an implementations from PETSc.

Factory

The factory implementation enables to discover, load, and execute plugins. The applied approach is based on the so called self-registering technique, enabling the plugins to register themselves in a global plugin database upon loading the DSOs by the Portable Operating System Interface (POSIX) `dlopen` command. The implementation is based on the so-called template factory design pattern [24][154], which can be seen as an extension of the abstract factory design pattern with C++ templates.

Figure 5.15 depicts a simplified class diagram of the registration mechanism. The `Base` and `Concrete` template parameters refer to a base and a derived class of a class hierarchy, respectively. This hierarchy in turn relates to the base and derived classed of a plugin system, holding the actual functionality. Due to the increased genericity introduced by the template factory design pattern, class hierarchies of arbitrary type can be stored. However, the derived class has to satisfy a so-called registrable concept. This concept requires the derived class to provide a static function named `ID` returning an identification (ID) string and to offer a member type named `Base` holding the type of the base class. The need for the registrable concept is discussed in the following.

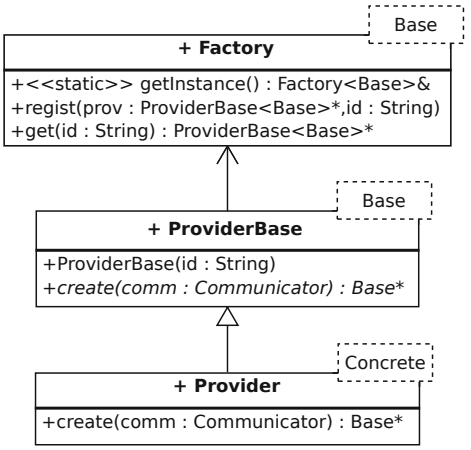


Figure 5.15: Class diagram of the implemented template factory design pattern. The constructor of the `ProviderBase` class registers instances of itself into the singleton `Factory` class.

Each plugin source file holds aside of the implementation of the derived plugin (`ViennaCLLinSol`) a static object of the type `Provider<ViennaCLLinSol>`. The `Provider` class is part of the factory mechanism and provides automatic registration within the factory's database. This automatism is based on the fact that static objects are generated during the start-up phase of the application, thus the registration related code provided by the `Provider` class is automatically executed before the main application is executed.

The constructor of `Provider<ViennaCLLinSol>` utilizes the registrable concept induced interface to access the base class type and the ID string. This information is forwarded to the `ProviderBase<Base>` constructor which in turn registers itself in the instance of the singleton pattern-based factory class. Using the factory's `get` method, a specific plugin's `Provider` class can be retrieved and created with the respective `create` method.

Interface

This section discusses the plugin interface which has to be modeled by a ViennaX plugin. Additionally, the general class hierarchy and the access for the ViennaX scheduler kernels is introduced.

ViennaX offers a three stage interface model, enabling an initialization, execution, and finalize step realized by the `init()`, `execute()`, and `finalize()` functions, respectively. Although such a three-stage interface is known to handle most application scenarios, more sophisticated needs cannot be covered by such an approach, for instance, additional communication between the individual components. Therefore, improving the interface for more intricate cases is part of future extensions.

The scheduler kernels use a `load` method to initialize the plugin with the plugin specific configuration data and with a unique plugin ID integer. The constructors are used by the factory mechanism to instantiate the plugins as well as providing the plugins with a `Communicator` object.

If ViennaX is compiled with MPI support, the communicator refers to a Boost MPI communicator, otherwise it maps to an integer value, enabling to compile ViennaX on non-MPI targets without any changes.

With respect to the implementation, a straightforward dynamic polymorphism approach via virtual functions is used to specialize the functionality for each plugin. Boilerplate code[7], required to implement, for instance, the appropriate plugin's constructor, is automatically generated by macros to increase the level of convenience.

Sockets

Aside of loading and executing plugin implementations via the interface, data communication between the plugins is a vital task in the field of CSE. For instance, a scalar field representing the result of a simulation conducted in a plugin might be used as an initial guess for another simulation performed by a subsequent plugin.

The approach for the plugin communication layer is based on previously conducted research for the COOLFluiD framework [24]. We refer to the communication access points in plugins as sockets. The socket system supports input and output data ports, called sink and source sockets, respectively.

In general, the data associated with the sockets can either be already available, thus no copying is required, or it can be generated automatically during the course of the socket creation. The following code snippet creates a source socket, generating the associated data object automatically.

```
create_source<Vector>("x");
```

The data of the socket can be accessed by the following.

```
Vector& x = access_source<Vector>("x");
```

If a data object is already available, the socket can be linked to it.

```
Vector x;
link_source(x, "x");
```

Similar implementations for socket creation and access are available for sink sockets.

Figure 5.16 gives an overview of the socket implementation via a class diagram. In general, the socket hierarchy utilizes a socket ID class and a database class to store the data associated with the sockets (Figure 5.17). Sockets can be compared to enable matching validation tests. The remainder of this section discusses the database implementation and the socket class hierarchy.

The `DataBase` class provides a centralized, generic storage facility for the data associated with the sockets. This storage additionally provides access and lookup mechanisms for retrieving and deleting the data objects of a given socket. The storage internally uses an associative container, mapping a string ID value to a `void`-pointer, thus being able to hold pointers of arbitrary type.

[7] Boilerplate code indicates code that occurs often in the implementation with almost no changes. Therefore it is cumbersome to implement as well as to maintain and is thus often the target of automated code generation using macros.

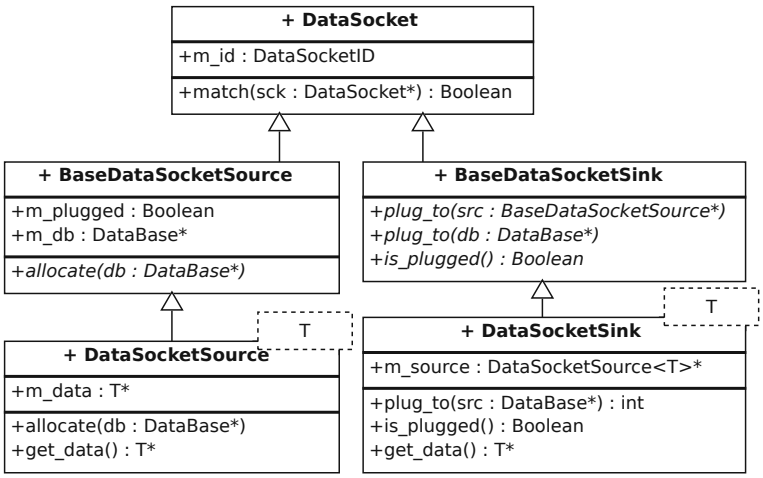


Figure 5.16: Class diagram of the socket system.

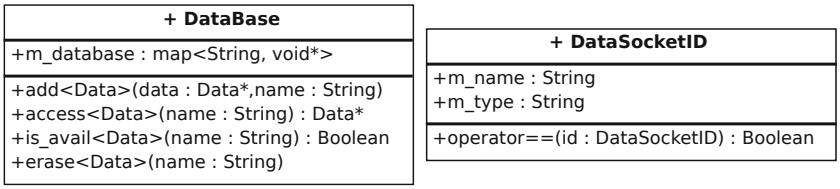


Figure 5.17: Class diagram of the socket database and ID class.

The ID string is generated from the name of the socket and the type string, thus as long as the names are unique, the data can be clearly identified even if the types are the same. This access mechanism represents the key of the entire socket-based data communication layer. The applied socket data storage approach decouples the actual storage related tasks from the actual socket implementations, thus improving maintainability and expandability as, for instance, possible future extensions to the socket storage layer can be conducted without interfering with the socket implementations.

To enable storing source and sink sockets and holding data of arbitrary types in a homogeneously typed data structure, a virtual inheritance approach is applied. As such, source and sink sockets are generalized by the `BaseDataSocketSource` and `BaseDataSocketSink` classes, respectively. The derived, type-aware socket class specializations `DataSocketSource/Sink`, provide access to the associated data object via the `get_data` function. In general, a source socket holds the actual data pointer (`m_data`), whereas the sink socket merely points to the corresponding source socket (`m_source`). A sink socket has thus to be linked to a source socket via the `plug_to` method, which is explained in the following.

Before working with the socket data, the source sockets have to be allocated and the sink sockets have to be linked to their respective source counterparts. This step can be implemented using the `allocate` and `plug_to` methods. The `allocate` function requires a pointer to an already available socket database object, which is then used for the allocation implementation, as depicted in the following.

```
1 db->add(m_data, m_name);
2 m_db = db;
```

In Line 1 the data pointer (`m_data`) is added to the socket database (`db`), whereas in Line 2 the externally provided database pointer is stored locally for future references.

The socket linking step, required for accessing the data of sink sockets, is implemented by the `plug_to` method, which prior to updating the internal source socket pointer verifies socket compatibility.

```
1 if(match(src)) m_source =
2   static_cast<DataSocketSourceT*>(src);
```

Therefore, a suitable external source socket has to be provided by the calling instance utilizing the `DataSocketID` information.

Aside of the exchange of data between plugins, the data communication layer inherently supports an approach to handle physical units in a straightforward manner. As already discussed, units are a major concern in CSE, as mixing the units between functions obviously results in a major corruption of the computational result [111]. As such it is of utmost interest to introduce automatic layers of protection to ensure that required data is given in the expected units. The communication approach enables to tackle this particular challenge by, for instance, coupling the unit information to the string-based ID of the sockets. As the automatic socket plugging mechanism requires the sink and source socket to have not only the same type, but also the same ID string, a sink and a source socket with different ID will not be connected. The string-based approach allows for coupling arbitrary properties to the sockets, making it a highly versatile system to impose correctness on the plugin data connections.

5.3.3 Exemplary Plugin Implementation

For the sake of clarity, an exemplary plugin implementation is provided, which not only depicts the utilization of the developed framework with respect to using already available implementations in plugins, but is also a reference for the subsequent investigation of the implementation details.

The plugin wraps a ViennaCL [68] iterative linear solver implementation, a high-performance reusable linear solver component supporting shared-memory computing platforms as well as accelerators (Section 5.1). Utilizing ViennaCL in this example additionally underlines the straightforward applicability of our framework with respect to utilizing already available implementations.

In the following, the full implementation of a simplistic ViennaCL-powered iterative solver plugin is given.

```
1  // Plugin Name
2  #define PLUGIN_NAME ViennaCLLinSol
3  // Plugin Class Implementation
4  struct PLUGIN_NAME : public plugin {
5  INIT_VIENNAX_PLUGIN
6  // Initialization: Setup Sockets
7  void init() {
8    create_sink  <Matrix>("A");
9    create_sink  <Vector>("b");
10   create_source<Vector>("x"); }
11 // Execution: Perform computation
12 bool execute(std::size_t call) {
13   // Access socket data
14   Matrix& A = access_sink  <Matrix>("A");
15   Vector& b = access_sink  <Vector>("b");
16   Vector& x = access_source<Vector>("x");
17   // Solve the system
18   x = solve(A, b, bicgstab_tag());
19   return true; } };
20 FINALIZE_VIENNAX_PLUGIN
```

The plugin's name (Line 2), class definition (Lines 4), and required macros - automatically generating boilerplate code required for the plugin mechanism - are implemented (Lines 5, 20). The data dependencies are set up in the initialization part (Lines 7-10). Two input sockets (`A`, `b`) and one output socket (`x`) are created in ViennaX's central socket database, relating to the linear system $Ax = b$. The data associated with the sockets are accessed (Lines 14-16) and the system is solved by using ViennaCL's biconjugate gradient stabilized linear solver [68] (Line 18). The result vector `x` is automatically available to other plugins via the outgoing data connection.

5.3.4 Configuration

Supporting a run-time configuration allows to define and drive the execution of the framework without the need for recompilation. As already indicated, ViennaX utilizes an input file - potentially provided by advanced users - approach based on XML, which's path is provided to the application via a command-line argument. Such a file-based method also enables possible future pairing with a GUI application generating configuration files upon an end user's behest in a straightforward manner, therefore eliminating the need for users to generate the configuration files by hand, further improving the support for end users. This input file contains information regarding the components to be utilized for the execution and also parameters for each component. In a first step the framework processes this input configuration and extracts the list of components to be utilized for the execution. According to this list, the corresponding components are created. In a second step, the component-specific parameters are extracted and forwarded to each individual component, allowing the component to change the setup of its input and output dependencies accordingly. In the following step, the execution order is determined by the schedulers, as the input and output dependencies are - at this point - final.

Using an XML approach for the input configuration data offers similar advantages as the previously discussed material database approach introduced in the context of the device simulation framework (Section 4.3.2). Most essential is the support for an arbitrary number of components, by using a dedicated XML tag. Also, the component-specific properties offer similar characteristics to material parameters, for instance, physical units. Furthermore, XML-based data are advantageous because of a rich ecosystem of libraries and tools, such as GUI based XML editors easing the interaction of XML files, especially important for large files. Additionally, having the parameters available in XML format enables the utilization of XPath queries to conveniently access the data from within the component framework and the actual components.

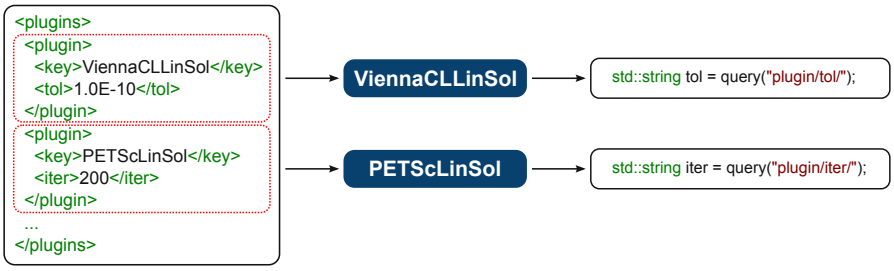


Figure 5.18: Each plugin possesses its own configuration region within the input configuration file. In this representative case default values for a break-tolerance of 10^{-10} for a ViennaCL linear solver component and an iteration limit of 200 iterations for a PETSc linear solver component are used. The values can be accessed from within the plugin by querying the configuration object.

In the following, we shall adapt the already introduced approach for a general parameter database in the context of the configuration of a component framework. A basic configuration file providing default values - potentially being generated by a GUI-based application to better support end users- with the sole purpose of executing the plugin named `ViennaCLLinSol` is depicted in the following.

```
1 <plugins>
2    <plugin>
3       <key>ViennaCLLinSol</key>
4       <tol>1.0E-10</tol>
5    </plugin>
6 </plugins>
```

The general `plugins` region contains the set of all plugins, which should be utilized during the execution (Lines 1-6). Each plugin is defined within its own region (Lines 2-5), which enables to pass parameters to the plugin instance (Figure 5.18). The name of the plugin has to be mentioned within the key region (Line 3), and must match the name as provided by the static ID method provided by the respective plugin (Section 5.3.2).

5.3.5 Scheduler Kernels

This section discusses the design and implementation of different scheduler kernels. In particular, a serial scheduler is discussed as well as a task parallel scheduler - components can be executed by different MPI processes if possible - and a data parallel scheduler - all components execute the same instructions, but solely on a specific subset of the data.

Serial Mode

The SM kernel is used for serial task graph execution on a shared-memory machine. Although the task graph is processed in a serial manner, the individual plugins can indeed utilize shared-memory parallelization approaches, such as OpenMP. The task graph implementation is based on the Boost Graph Library [48][155], which not only provides the data structure but also graph algorithms, such as topological sort [156].

The serial scheduler is based on the list scheduling technique [157]. Informally, this technique uses a prioritized sequence of tasks, which is then processed consecutively. Figure 5.19 depicts the major steps of execution flow.

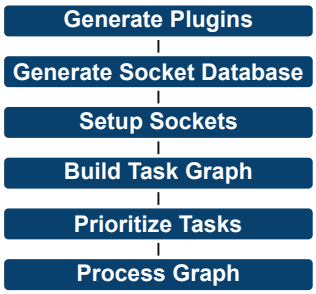


Figure 5.19: Flow diagram of the SM scheduler kernel.

As previously discussed in the context of the framework's configuration mechanism, the plugins are loaded according to the input configuration file (Section 5.3.4) by the factory mechanism (Section 5.3.2). Each plugin is configured based on the parameters listed in the input file. With these parameters the input and output dependencies are defined.

A task graph meshing algorithm connects the various plugins based on their dependencies. The meshing procedure is based on plugging the sink sockets of the plugins to valid source sockets of other plugins (Section 5.3.2). Validity is ensured by comparing the socket IDs, incorporating the socket key as well as the data associated with the socket. The generated task graph is used for building the prioritized sequence, generated by the Boost Graph's topological sort graph algorithm, as depicted in the following.

```
1 std::list<Vertex>   plist;
2 boost::topological_sort(graph,
3   std::front_inserter(plist));
```

The `graph` object is a directed graph data structure, which is used to hold the entire task graph. A directed graph is a graph consisting solely of directed edges, i.e., edges pointing from a source to a target vertex. In addition, the graph must not contain cycles[8], also known as loops. If these conditions are satisfied, the topological sort algorithm can be utilized to generate a linear sequence of vertices - representing the ViennaX components - which upon execution guarantees that the input dependencies of each component are met. More concretely, the prioritized sequence of tasks is processed consecutively by traversing the result container `plist` and executing the individual plugins via the plugin's `execute` interface method (Section 5.3.2). Note that the linear solver plugin introduced in Section 5.3.3 can be utilized with this scheduler.

[8] A graph which does not contain loops is typically referred to as *acyclic*.

Despite of the acyclic requirement, the SM scheduler supports loops by identifying the loop entry vertex and the loop exit vertex. These particular vertices are identified by a loop-detection algorithm based on Boost Graph's implementation of Tiernan's approach to detect cycles in an acyclic graph [158]. The loop is then broken up to satisfy the acyclic condition, followed by executing the topological sort algorithm to determine the linear execution sequence. ViennaX manually triggers a re-execution of the subgraph representing the loop part via the previously identified loop vertices.

To implement a loop, the components representing the previously mentioned loop entry and loop exit vertices, need to provide a sink-source socket connection representing the loop. To this end, the loop exit component implements a source socket whereas the loop entry component provides a corresponding sink socket, thus closing the loop via a backwards connection. As this connection is merely required for establishing a dependency in the task graph, the associated data is irrelevant, and can thus be chosen arbitrarily. The loop exit component is responsible for triggering a loop continuation or a loop exit by either returning *false* or *true* at the end of the execution method, respectively. The framework evaluates the boolean return value and acts accordingly.

Distributed Task Parallel Mode

The DTPM scheduler kernel enables applications focusing on a task parallel approach. In general, the scheduler follows a static scheduling approach, based on load balancing indicated by optional plugin weights. Similar workload distribution approaches are available, focusing on dynamic scheduling implementations based on, for instance, work-stealing [159][160]. The execution of the individual plugins is distributed among the available MPI processes. Therefore, a considerable speedup of the task execution can be achieved, if the task graph offers parallel paths. Figure 5.20a depicts the flow diagram of the scheduler.

The DTPM scheduler has two peculiarities: First, the global task graph is partitioned and ultimately the individual subgraphs are processed by different MPI processes. Second, as the plugins sharing a data connection might be executed on different MPI processes, an extension to the socket data communication layer incorporating the distributed memory environment is required.

The distribution of the workload is based on the METIS graph partitioning library [161], to automatically improve the efficiency of the parallel execution of the task graph. A weighting approach is implemented enabling an advanced user or a developer to assign a weight to the plugin implementation via a corresponding method, indicating the computational load of the respective component. This load value is used by METIS, aiming to equalize the computational effort over the generated partitions, thus improving parallel execution efficiency. In general, the larger the assigned value the larger the associated computational load of the respective component. Considering four components (each offering a load of one) and two processes, each process will be assigned two components as each process is responsible for a computational load of two. However, if one component has a load of three instead, one process will be responsible for three components, whereas the other will be dealing with one component; in this case both processes will handle a computational load of three.

With respect to the implementation of the METIS-based graph partitioning, internally, the developed framework approach converts the Boost Graph-based graph data structure to a compressed sparse row format [120], which is required by the METIS API. The computational weights are transferred to the METIS backend and the corresponding partitioning algorithm is executed. The algorithm aims to minimize the so-called *edge-cut*, aiming to minimize the number of edges which straddle partitions by simultaneously balancing vertex weights across partitions. Upon completion, each vertex is assigned a partition number which is used by ViennaX to distribute the workload among the MPI processes accordingly.

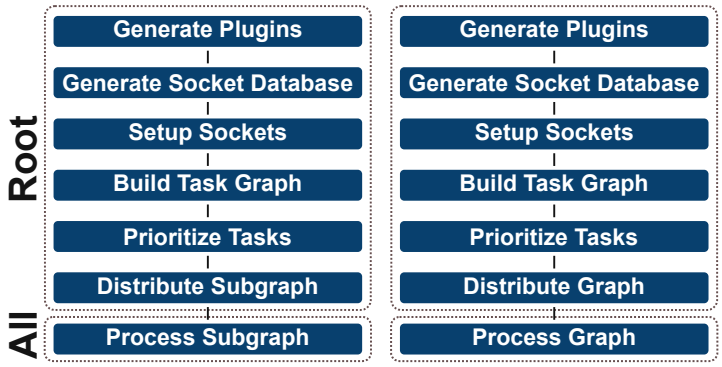
The second peculiarity of the DTPM scheduler, being the incorporation of a distributed memory environment into the socket data communication layer, is based primarily on the non-blocking point-to-point communication capabilities of the MPI layer. The graph partitioning step yields, aside of the MPI process assignments of the plugins, a lookup table for the socket communication. Each MPI process holds its own socket database, and utilizes the communication lookup table to determine the corresponding transmission sources and sinks. This mechanism is utilized after a plugin has been executed on an MPI process, where its source sockets requiring outbound inter-process communication are traversed and the transmission is initiated.

In general, the non-blocking point-to-point methods are utilized to increase execution performance. This is crucial, as an MPI process should in the optimal case not wait for an outgoing transmission to be finished before it executes another plugin. Such an approach is typically referred to as overlapping communication with computation. However, using a pure MPI approach and therefore non-blocking communication methods, such an overlap is rarely achieved. In fact, specialized hardware and software is required to achieve a reasonable overlap, for instance, Cray's XE6 with Gemini interconnects is capable of delivering such an overlap [100]. A possible future extension would be a hybrid approach, utilizing MPI and threads to implement a true asynchronous approach, thus introducing a much more improved overlap of communication and computation. The DTPM scheduler currently does not support loops, being especially challenging due to the distributed nature of the components requiring additional communication to orchestrate a looped-execution.

The linear solver plugin introduced in Section 5.3.3 can be utilized with this scheduler, as each plugin is executed by one process. Therefore, one MPI process accesses the available computational resources via the parallel accelerator layer.

Distributed Data Parallel Mode

The DDPM scheduler kernel enables simulations based on the data parallel approach. Figure 5.20b depicts the flow diagram of the scheduler implementation. Contrary to the DTPM scheduler, the graph is not partitioned as all plugins are processed by all MPI processes in the same sequence. The root process prepares the task graph and generates a prioritized list of plugins. This list is distributed to all MPI processes each processing the graph in its entirety. As with the DTPM scheduler, each MPI process holds its own socket database responsible for storing the data associated with the sockets on the local process.



(a) DTPM (b) DDPM

Figure 5.20: **(a)** Flow diagram of the DTPM scheduler kernel. The root MPI process is responsible for preparing and distributing the workload evenly between the compute units. All available compute units process their distinct parts of the graph. The fact that this scheduler assigns parts of the graph to the compute units is indicated by the respective *Subgraph* nodes. **(b)** Flow diagram of the DDPM scheduler kernel. Similar to the DTPM scheduler, the root MPI process prepares the entire task graph. However, the entire workload is distributed to all MPI processes, as each process executes the entire task set represented in the task graph.

A peculiarity of the DDPM scheduler kernel is the fact that each plugin has access to an MPI communicator object via the `comm` method, providing access to the entirety of the MPI environment. The following code snippet depicts an exemplary utilization in a plugin's implementation to evaluate the rank of the current MPI process.

```
1 if (comm().rank() == 0) {
2   // Root code
3 }
```

A Boost MPI communicator object offers implicit conversion to a raw MPI communicator, ensuring interoperability with non-Boost MPI implementations.

Figure 5.21 shows the execution behavior of the scheduler. Each plugin is processed by all MPI processes and has access to an MPI communicator. Inter-plugin communication is provided by the socket data layer, whereas inter-process communication is supported by the MPI library.

Note that the utilization of the linear solver plugin introduced in Section 5.3.3 is not reasonable here, as in this case each process would perform the computation, thus massively overburdening the compute unit beyond reasoning. For the scheduler at hand, an MPI-powered linear solver implementation is the proper choice, as is provided by, for instance, the PETSc library.

5.3.6 Examples

This section presents application results for each scheduler provided by ViennaX. Due to the nature of ViennaX, the framework can be applied to different fields of CSE. To reflect this fact the discussed examples have been selected accordingly by investigating application scenarios outside of the field of MNDS.

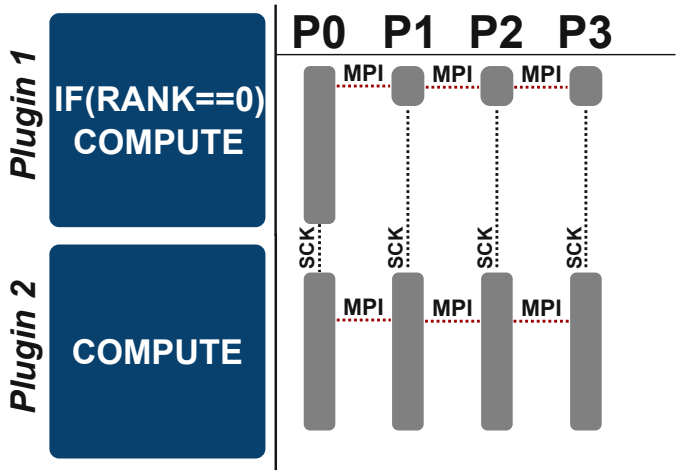


Figure 5.21: Exemplary execution behavior of the DDPM scheduler based on two plugins and four MPI processes. The bars in the right part of the figure indicate the computational load. Each MPI process executes the individual plugin. Additionally, each plugin has access to an MPI communicator object, enabling not only classical data parallel execution modes but also plugin inter-process communication. Inter-plugin communication is realized by the socket mechanism (SCK).

Serial Mode

This section discusses adaptive mesh refinement (AMR), a typical requirement for FE simulations. The general idea is to improve the solution process by locally adapting the mesh during the course of the simulation. Usually, a first solution is computed based on an initial mesh. Based on this solution, the mesh is locally adapted. Typically, a refinement is conducted in regions with large gradients or large curvatures in the solution [162][163]. Increasing the resolution, i.e., adding mesh elements in areas of interest, tends to increase the overall accuracy. The adaptation and solution steps are repeated, until a certain threshold level is reached as indicated by an error estimator [164].

In the following, an example application provided by the deal.II library is investigated [92]. This discussion is followed by depicting a decoupled implementation including simulation results based on ViennaX. Concretely, the `step-8` example, dealing with an elasticity problem coupled with an iterative AMR scheme is analyzed and ViennaX's loop mechanism for modeling the iterative nature of the AMR algorithm is investigated. We analyze the initial implementation, followed by a detailed description of a decoupling scheme.

The initial implementation is based on a single C++ source file, containing a class implementation (`ElasticProblem`), which provides methods to utilize the deal.II library for solving the elasticity problem.

The subsequent code discussion is based on the class interface instead of the full implementation.

```
1 template <int dim>
2 class ElasticProblem { // ..
3  void run ();
4  void setup_system ();
5  void assemble_system ();
6  void solve ();
7  void refine_grid ();
8  void output_results (const unsigned int cycle) const; };
```

The structure of the class implementation provides an intuitive description of the entire simulation process. The class itself is parameterized via the compile-time template parameter `dim` (Line 1), indicating the dimensionality of the problem. The `run()` method (Line 3) drives the overall simulation by performing the iteration process, in turn calling the individual methods required for the computation (Lines 4-8). As the focus of this particular application example is on implementing the AMR algorithm via the ViennaX loop mechanism, the implementation of the `run()` method deserves closer investigation:

```
1 template <int dim>
2 void ElasticProblem<dim>::run () {
3  for (unsigned int cycle=0; cycle<6; ++cycle) {
4   if (cycle == 0) {     // Initial mesh generation
5    hyper_cube (triangulation, -1, 1);
6    triangulation.refine_global (2); }
7   else refine_grid (); // AMR
8   setup_system ();
9   assemble_system ();
10  solve (); } }
```

The AMR loop is implemented via the for-loop and the `refine_grid` method (Lines 3-7). In the first iteration an initial mesh is generated (Lines 4-6). During each loop iteration all steps required for this particular FE simulation are conducted, being the assembly of the linear system of equations and the linear solution process (Lines 8-10).

Concerning the implementation of a decoupled version by using ViennaX, an exemplary decoupling of this particular simulation is schematically depicted in Figure 5.22.

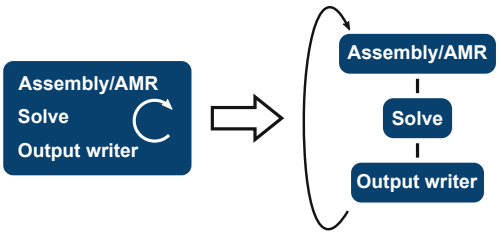


Figure 5.22: The schematic overview of the initial deal.II example (**left**) and the proposed decoupling into a loop-based execution graph (**right**) is shown. For the sake of simplicity, the sockets are abstracted by single edges.

Note that this particular `ElasticProblem` class implementation had to be slightly altered to allow external access to the data structures and methods. In the following, we focus on the implementation aspects of the `Assembly/AMR` and `Output writer` component.

The initialization phase of the `Assembly/AMR` component is implemented as follows.

```
1 // Component state
2 ElasticProblem<2>     sim; /* .. */
3 void init() { // Initialization method /* .. */
4 link_source(sim, "sim");
5 create_sink<bool> ("loopback");  }
```

We omit the instantiation of the data structures for the mesh, the system matrix, and the load vector. These data structures - kept in the state of the component - are forwarded to the constructor of the `sim` object. An additional sink socket is used for the loop mechanism (Line 5). A corresponding source socket is used in the last component in the loop sequence, being the `Output writer` component.

The implementation of the execution part is similar to the original version.

```
1 bool execute(std::size_t call) {
2  if (call == 0)     {
3   hyper_cube (triangulation, -1, 1);
4   triangulation.refine_global (4);  }
5  else sim.refine_grid ();
6  sim.setup_system ();
7  sim.assemble_system ();
8  return true; }
```

No explicit loop mechanism is required, such as a `for`-loop, as ViennaX automatically issues a re-execution of the components within the loop. However, to keep track of the loop iteration within the component, the `call` variable is used (Lines 1,2), replacing the originally utilized `cycle` loop parameter. The `sim` object is used to call the respective methods for assembling the linear system of equations (Lines 5-7).

The `Output writer` component is - additionally to generating the simulation output files - responsible for notifying ViennaX whether to continue the loop. Aside of the obvious sink sockets required to access the result data to be written to a file, this particular component has to provide a complementary source socket to close the loop defined in the initialization phase:

```
1  create_source<bool>    ("loopback");
```

This socket is not required for actual data communication, it is merely required to inform ViennaX that there is a backward-dependence.

The execution phase utilizes the boolean return value to indicate whether the loop has to be continued. In the following example, after six executions, the loop is exited.

```
1 bool execute(std::size_t call) { /* .. */
2   if(call == 6) return true;
3   else return false; }
```

In Lines 2,3 ViennaX's loop break condition is implemented by comparing the `call` parameter with the intended number of six loop iterations. In this exemplary case, the value is hard coded, however, it can also be implemented in a non-intrusive manner via the data query mechanism provided by ViennaX's configuration facility (Section 5.3.4). By convention returning `true` forces ViennaX to exit the loop, whereas returning `false` triggers a new iteration.

Figure 5.23 depicts exemplary simulation results for a three-dimensional simulation based on the introduced decoupled implementation.

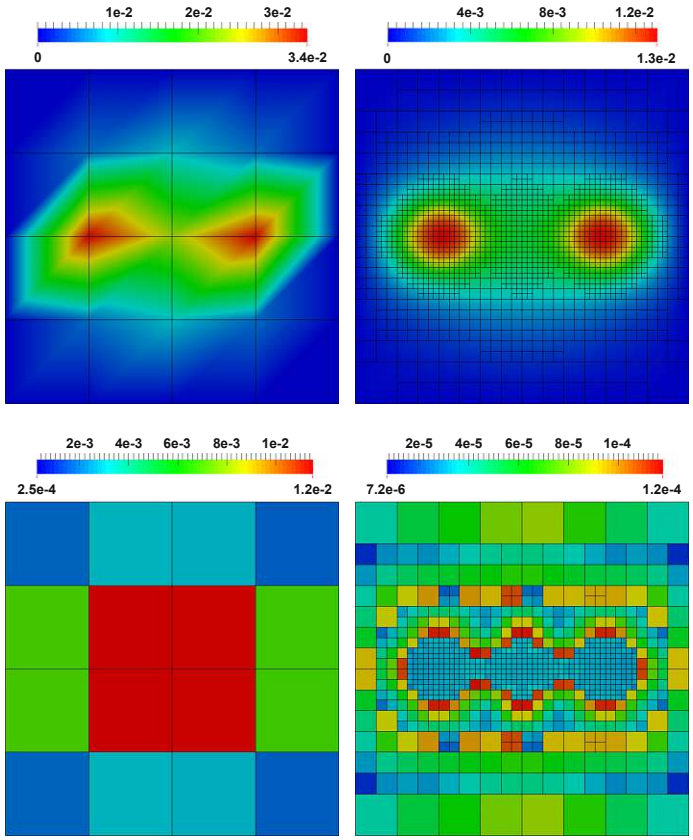


Figure 5.23: **Top:** The x-displacement for the deal.II example for the initial (left) and the six-times adapted mesh (right). **Bottom:** The maximum error (infinity norm) of the computed error estimates drops from 0.012 to $3.9 \cdot 10^{-5}$. The depicted error estimates for the sixth iteration would trigger a refinement in the corners in the subsequent iteration step, as due to the continuing reduction in the error the relative small solution gradients become relevant.

Distributed Task Parallel Mode

This section investigates the scalability of the DTPM scheduler by a Mandelbrot benchmark as implemented by the FastFlow framework [165][166]. Implementation details of the benchmark are provided as well as performance results. This example has been benchmarked on HECToR [167], a Cray XE6 supercomputer with a Gemini interconnect located at the University of Edinburgh, Scotland, UK.

The DTPM scheduler is used to compute the Mandelbrot set for a partitioned application domain by using different instances of a Mandelbrot plugin responsible for the individual parts. This particular example has been chosen, as the computational effort of computing the Mandelbrot set is inhomogeneously distributed over the application domain. Therefore, the computational load of the individual parts have to be weighted accordingly to improve the scalability.

The partial results are gathered at the end of the simulation, which provides insight into the communication overhead (Figure 5.24). The implementation of the Mandelbrot plugin performs a partitioning of the simulation domain. Thus, each instance is responsible for a subdomain, which is identified by the plugin index. In its essence, this approach follows a data parallel approach, which also shows that the DTPM can be used for such kind of tasks, despite of its inherent focus on task parallelism.

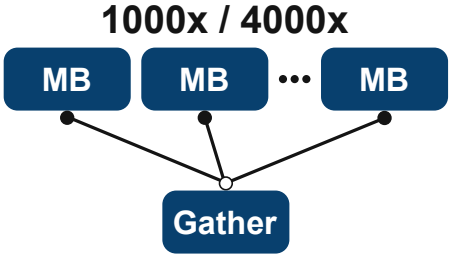


Figure 5.24: The graph for the Mandelbrot benchmark. 1000 and 4000 instances of the Mandelbrot plugin (MB) have been used. The partial results are gathered, which forces MPI communication.

Two different simulation domains have been investigated, being a grid of 1000x1000 and 4000x4000 to demonstrate the influence of varying computational load on the scaling behavior. Each plugin processes one line of the grid, consequently 1000 instances for the smaller and 4000 instances for the larger case have been used.

The following depicts the configuration file for the smaller benchmark.

```
<plugins>
  <plugin>
    <key>Mandelbrot</key>
    <clones>1000</clones>
    <dim>1000</dim>
    <niters>10000000</niters>
  </plugin>
  <plugin>
    <key>MandelbrotGather</key>
  </plugin>
</plugins>
```

The `clones` entry triggers an automatic duplication of the plugin in ViennaX, thus in this case 1000 instances of the Mandelbrot plugin are generated. The parameters `dim` and `niters` are forwarded to the plugins and relate to the number of grid points in one dimension and the number of iterations (in this case 10^7 iterations are conducted to increase the computational effort) used in the Mandelbrot algorithm, respectively. The configuration of the larger benchmark is similar, however, the number of clones and dimensions is increased to 4000, respectively.

During the Mandelbrot plugin's initialization phase, the output socket has to be generated as well as the computational level of the plugin has to be set to balance the computational load over the computational resources. A peculiarity of computing the Mandelbrot set is the fact that the computational load in the center is larger than on the boundary. Therefore, a weighting scheme has been applied to increase the load balancing over the MPI processes. The plugin instances in the center are assigned a higher computational weight (Section 5.3.5) as depicted in the following.

```
1 void init() {
2   double current = pid();
3   if( (current >= dim*0.45) &&
4       (current <= dim*0.55) )
5     computation_level() = 100;
6   // ..
7 }
```

In Line 2 the current plugin ID is extracted, which has been provided by the scheduler during the preparation phase (Section 5.3.2). As each plugin is responsible for a single line of the simulation grid, its ID is tested whether it is responsible for the central part. In this case, the central part is identified by evaluating whether the component is responsible for the centered 10% of the grid line, i.e., to be larger equal than 45% (0.45) and smaller equal than 55% (0.55) of the grid line dimension. If so, the computational level is set to a high value, such as 100 in this case, to ensure that these center components are equally spread over the computational resources by the scheduler's graph partitioner.

As each plugin computes a subset of the simulation result, the output port has to be localized with respect to the plugin instance to generate a unique socket. As already stated, the socket setup has to be placed in the initialization step.

```
create_source_local<Vector>("vector");
```

This socket generation method automatically generates sockets of the type `Vector` and the name `vector` including an attached string representing the plugin ID, retrieved from the `pid` method. This approach ensures the generation of a unique source socket for each Mandelbrot plugin instance.

The execution part accesses the data associated with the source socket and uses the data structure to store the computational result. The plugin ID is used as an offset indicating the responsible matrix line which should be processed.

```
bool execute(std::size_t call) {
  Vector & vector =
    access_source_local<Vector>("vector");
  i = pid();
  for (j = 0; j < dim; j++) {
    // compute k as a function of i and j
    vector[j] = k;
  }}
```

On the receiving side, the Gather plugin's initialization method generates the required sink sockets using a convenience function.

```
1 void init() {
2   create_sink_set<Vector>("vector");
3 }
```

The above method automatically generates one sink socket for $0 \ldots n-1$ predecessor plugins, where n represents the current plugin ID. Therefore, all previous output ports generated by the Mandelbrot plugins can be plugged to the respective sink sockets of the Gather plugin.

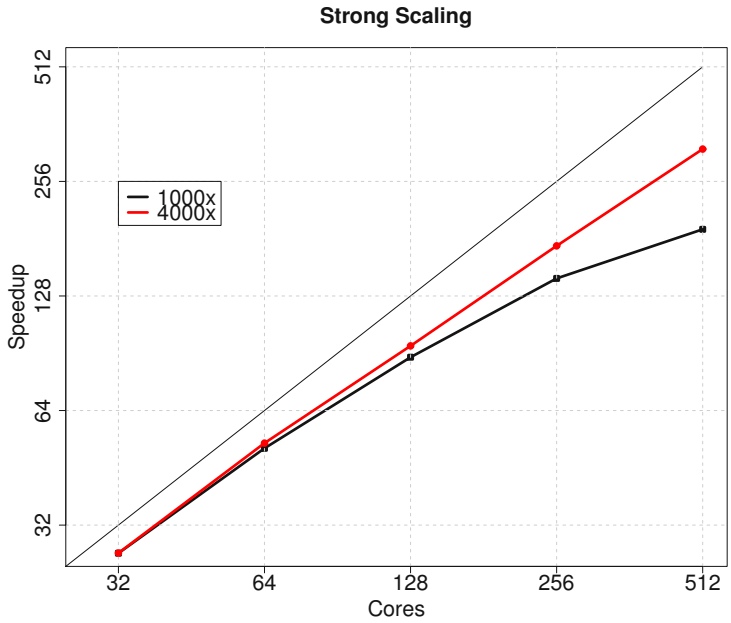


Figure 5.25: The strong scaling behavior of the Mandelbrot benchmark. Increasing the computational load on the plugins shifts the scaling saturation towards higher core numbers. The smaller problem scales well for up to 256 cores, however, for higher core numbers the communication overhead outpaces the computational load on the plugins. Both benchmarks depict a non-optimal speedup for small core numbers, which is due to communication overhead and insufficient load balancing.

The execution phase is dominated by a gather method, similar to the MPI counterpart.

```
bool execute(std::size_t call) {
  std::vector<Vector> matrix;
  gather<Vector>("matrix", matrix);
}
```

The partial results of the $0 \dots n-1$ predecessor plugins are stored consecutively in a `vector` container, thus forming the result matrix where each entry corresponds to a point on the two-dimensional simulation grid.

Figure 5.25 depicts the strong scaling results, i.e., a fixed problem size is investigated for different core numbers. Reasonable scaling is achieved, although communication overhead and load balancing problems are identifiable already for small core numbers. However, increasing the computational load on each plugin and the number of plugins to be processed by the MPI processes further shifts the scaling saturation towards higher core numbers. For the smaller problem an efficiency of 38% and for the larger problem an efficiency of 60% for 512 cores is achieved. Improving the load balancing via the plugin weighting approach as well as introducing a hybrid scheduler to improve communication and computation overhead will further improve the scaling behavior.

Distributed Data Parallel Mode

The DDPM scheduler is investigated by comparing the execution performance to a reference implementation provided by the deal.II library [91][92]. This benchmark not only shows that an available implementation using external high-performance libraries can be transferred to the ViennaX framework in a straightforward manner, but also that the execution penalty of using the framework is negligible. Similar to the previous example, HECToR [167] was used as benchmark target.

A large-scale two-dimensional Laplace test case named `step-40` of the deal.II library offering 67 million degrees of freedom is considered, which utilizes components representing important aspects of large-scale high performance computing applications [168]. For instance, the data structure holding the mesh and the linear system is fully distributed by using data structures provided by the PETSc library [122] and the p4est library [169][170]. The equations are discretized using biquadratic finite elements and solved using the conjugate gradient method preconditioned by an algebraic multigrid method provided by the Hypre [171] package and accessed via PETSc.

The reference implementation is split into two functional parts, being the assembly by the deal.II library and the solution of the linear system via the PETSc library. Therefore, two plugins have been implemented, which also underlines the reusability feature (Figure 5.26). For instance, the linear solver plugin can be replaced with a different solver without changing the implementations.

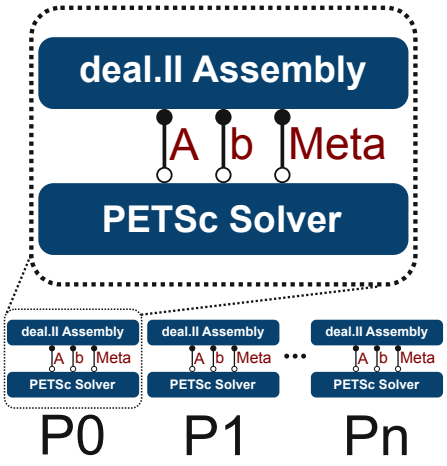


Figure 5.26: The graph of the deal.II benchmark, which is executed by all MPI processes (P0-Pn). The system matrix (A) and the right-hand side (b) as well as meta information (Meta) is forwarded to the solver plugin via the socket communication layer.

The following depicts the configuration file for the benchmark.

```
<plugins>
   <plugin>
      <key>deal.II.Assemble</key>
   </plugin>
   <plugin>
      <key>PETSc.Solve</key>
   </plugin>
</plugins>
```

The assembly plugin's initialization phase prepares the distributed data structures as well as provides the source sockets. The data structures required for the computation are generated and kept in the plugin's state. Therefore, the source sockets are linked to the already available objects, instead of created from scratch during the socket creation. The execute method of the assembler plugin forwards the simulation data structures to the implementation provided by the deal.II example, which performs the actual distributed assembly. This fact underlines the straightforward utilization of already available implementations by ViennaX plugins. The following code snippet gives a basic overview of the assembly plugin's implementation.

```
1  // Plugin state
2  Matrix     matrix;
3  Vector     rhs;
4  Mesh       triangulation;
5  Simulation sim;
6
7  void init() {
8    link_source(matrix, "matrix");
9    link_source(rhs,    "rhs");
10   // ..
11 }
12
13 bool execute(std::size_t call) {
14   GridGenerator::hyper_cube (triangulation);
15   triangulation.refine_global (10);
16   sim.assemble_system ();
17 }
```

The plugin state holds various data objects (Lines 2-5). The `sim` objects holds references of the other objects, thus the internals of the simulation class can access the data structures. The source sockets link to already available data objects (Lines 8,9). The simulation grid is generated (Lines 14,15) and the linear system (`matrix`, `rhs`) is assembled (Lines 16).

The solver plugin generates the corresponding sink sockets in the initialization part and utilizes the PETSc solver environment in the execution method. As each MPI process holds its own instance of the socket database, the pointers of the distributed data structures are forwarded from the assembler to the solver plugin. The PETSc internals are therefore able to work transparently with the distributed data structures without the need for additional copying operations. The following depicts the crucial parts of the solver plugin's implementation.

```
1  void init() {
2    create_sink<Matrix>   ("matrix");
3    create_sink<Vector>   ("rhs");
4  }
5  bool execute(std::size_t call) {
6    Matrix &matrix=access_sink<Matrix>("matrix");
7    Vector &rhs   =access_sink<Vector>("rhs");
8    // compute solution vector x }
```

Figure 5.27 compares the execution performance of the ViennaX implementation with the deal.II reference implementation. A system of 67 million degrees of freedom is investigated, which due to it's memory requirements does not fit on one compute node. Generally, excellent performance is achieved, however, an overall constant performance hit of about one second for all core numbers is identified. This performance hit is due to the run-time overhead introduced by the plugin framework, such as the time required to load the plugins, generate the task graph, and perform virtual function calls. The relative difference for 1024 cores is around 8%, however, it is reduced to 1.5% for 64 cores, underlying the fact that the framework's overhead becomes more and more negligible for larger run-times. This overhead is acceptable, as the simulations we are aiming for have run-times way beyond 50 seconds, as is the case for 64 cores.

Although the relative difference is significant for short run-times, a delay of one second hardly matters in real world, day-to-day applications. On the other hand, accepting this performance hit introduces a significant increase in flexibility to the simulation setup due to the increased reusability of ViennaX's component approach.

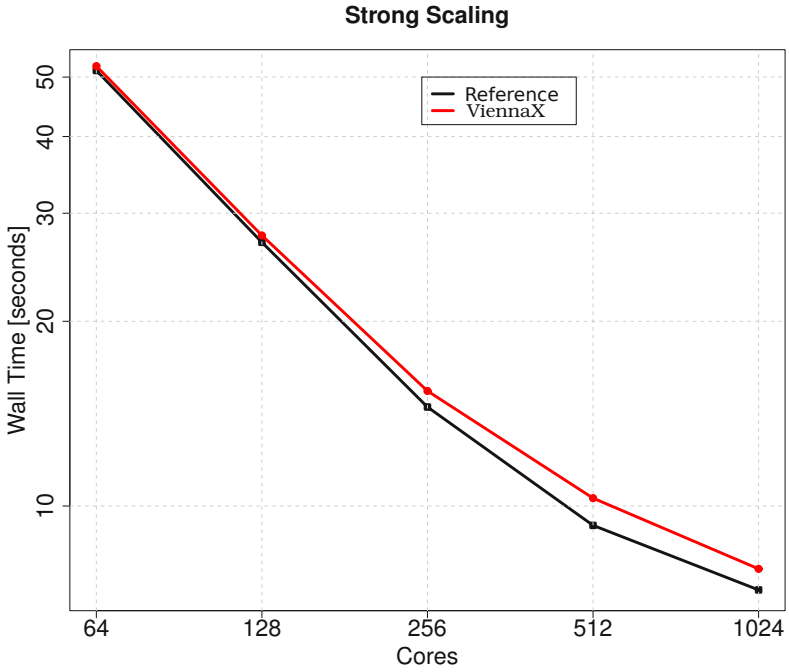


Figure 5.27: The execution performance of the Laplace benchmark is compared to the reference implementation provided by the deal.II library. ViennaX is approximately one second slower than the reference implementation throughout the core spectrum.

Chapter 6

Interactive Simulation Framework

Although the previously discussed component execution framework increases the flexibility of setting up simulations, the approach supports primarily advanced users. To extend support specifically to end users, this chapter presents an interactive simulation framework, based on a CBSE approach - similar to the previously discussed component execution framework - to enable a flexible and modular setup of simulation-focused CSE applications. As the name suggests, the focus is on end user interaction and thus on usability. Each component of the CBSE-based framework shall provide access to a full-fledged application, such as a device simulator, providing the end user the ability to, for instance, utilize different simulation tools on the same software platform. To underline the component's containment of a full-fledged application, the term *module* is used instead of component. Overall, an interactive simulation framework promises improved end user experience as different applications can be utilized in a unified manner, as, for instance, they share the same visualization backend. The key for supporting this is based on enforcing a unified interface upon the individual modules wrapping the external tools, as is introduced by a CBSE approach.

An interactive simulation framework is characterized by extending the previously established requirements for a component execution framework by incorporating GUIs (Section 6.1). Interactive simulation frameworks further extend the set of tools which benefit from LCSD-based libraries (Figure 6.1), as due to the CBSE-based approach available functionality provided by, for instance, libraries can be reused. Most importantly, though, due to the rigorously applied decoupled software designs presented in this work, reusability is not restricted to LCSD-based libraries, but also frameworks themselves can be reused. Section 6.2 introduces an approach for an interactive simulation framework. The presented investigations focus on utilizing the previously introduced device simulation framework ViennaMini (Section 4.3) instead of the component execution framework ViennaX, as it reflects the more relevant application scenario in the field of MNDS, being the focus of this work.

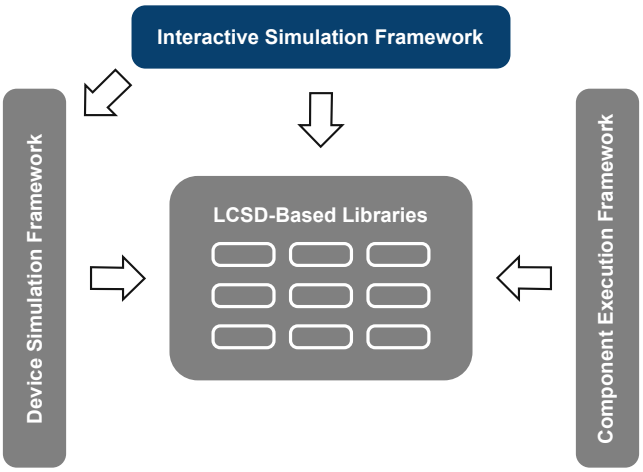


Figure 6.1: An interactive simulation framework further extends the previously introduced set of tools, those being the device simulation framework and the component execution framework, which benefit from a decoupled software design; in this case the device simulation framework as well as the libraries are wrapped as or used in modules and are thus reused within the interactive simulation framework.

6.1 Requirements and Challenges

Considering a typical simulation workflow, such as performing a device simulation of a transistor device, generates concrete requirements regarding an interactive simulation platform. In a first step the simulation domain representing the electronic device must be discretized - yielding a mesh - and visualized to the end user in a simplified form. The visualization of the simulation domain aids the end user to setup the simulation, by providing a visual target for the simulation setup. Device regions are not abstracted by numbers or names but actually rendered on the screen, allowing to assign segment roles in a natural manner. Therefore, it is not required to visualize the mesh elements by default but rather the so-called surface representation, enabling end users to grasp the shape and size of the simulation domain. However, advanced users with an additional background in mesh generation can choose to inspect the mesh visually supported by computational aids, e.g., algorithms which identify badly shaped mesh elements, especially required for three-dimensional meshes.

Based on the discretization, device-specific meta information such as material data and doping information has to be assigned to the mesh, elevating the mesh to a device. As already discussed in Section 4.2.1, the doping information is either based on the result of a numerical process simulation or according to an analytic approach where the latter is of particular importance for day-to-day use as no time-consuming numerical simulation is required. The thus generated device acts as an input for the actual device simulation step, where the physical problem is defined by appropriate models, such as transport and scattering models, and boundary conditions. The simulation is conducted and the results are visualized allowing the end user to analyze the device properties. To this end, not only three-dimensional quantity rendering is required, allowing to show, for instance, electron concentration distributions on top of a mesh, but also charts are required to visualize, for example, current-voltage characteristics.

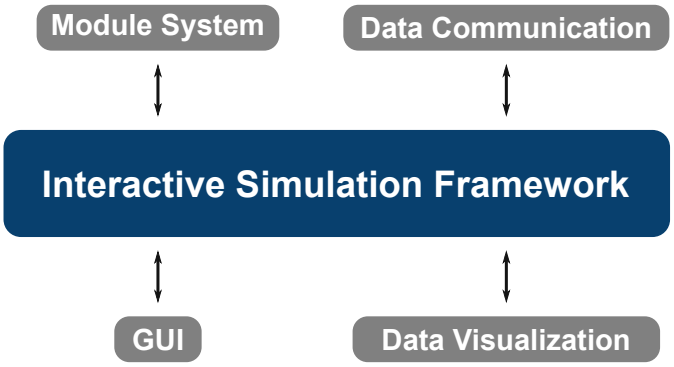


Figure 6.2: An interactive simulation framework consists of four essential parts, those being a module system, a data communication layer, a GUI, and a data visualization facility.

Based on these requirements, the essential key aspects of an interactive simulation framework are extracted (Figure 6.2), those being a module system, a data communication layer, a GUI, and a data visualization facility. A module system enables to separate different functionalities in standalone entities - thus enabling reusability - whereas a data communication mechanism allows the modules to exchange data, such as a device. A visualization facility enables to render meshes and quantities, such as simulation results, therefore enabling the end user to visually perceive the data, thus significantly aiding the end user in the various evaluation steps. A GUI provides a general high-usability platform to the end user, enabling to conveniently interact with the simulation platform. Note that the visualization mechanism is embedded into the GUI. However, as the visualization mechanism has rather specialized requirements, such as rendering data fields on top of meshes, as compared to general interaction elements of a GUI, like buttons and dialogs, they are considered separately.

Where the requirements and challenges of the module system and the data communication are similar to the already discussed approaches in Section 5.2, schedulers are typically not required with an interactive simulation framework. Instead of executing a set of modules in a single execution run - which indeed would require scheduling - only one module is usually executed at the end user's behest. This peculiarity stems from the primary area of application as a platform for providing individual simulation tools. Consequently, the results of each simulation are likely to be investigated by the end user, thus a single-module execution mode merits special consideration.

Contrary to the previously discussed component execution framework (Chapter 5), a key aspect of an interactive simulation framework is the ability to interact with the end user, represented by a GUI and a data visualization mechanism, such as three-dimensional rendering. Therefore, the GUI takes the place of the configuration mechanism as introduced in the context of a component execution framework. Each of these four introduced key aspects is discussed in detail in the following.

6.1.1 Module System

The module system extends the requirements introduced with the component execution framework (Section 5). Aside from imposing a unified interface upon the individual modules to enable exchangeability and expandability, each module is required to provide its own specific GUI. This ability is vital as simulation tools offer different parameters, hence requiring specific GUI-elements. The challenge is to extend the module interface - used to hold the module implementations - to further support module-specific GUIs.

Coupling a GUI with intensive computations by sharing a thread freezes the GUI, which is typically the case with numerical simulations. The thread is utilized for the computation, for instance, to solve a linear equation system, and is by itself[1] not able to process GUI-related events. This behavior must be avoided to allow interaction with the GUI during the computation. For instance, whilst a simulation tool performs the computations, the rendering facility should still allow interactions such as zooming or panning. The challenge is thus to implement a non-intrusive mechanism, allowing for GUI interactions during heavy computations, by, for instance, outsourcing the computation to a worker thread (Figure 6.3). In this context, non-intrusive refers to the fact that no changes in the wrapped simulation code are required.

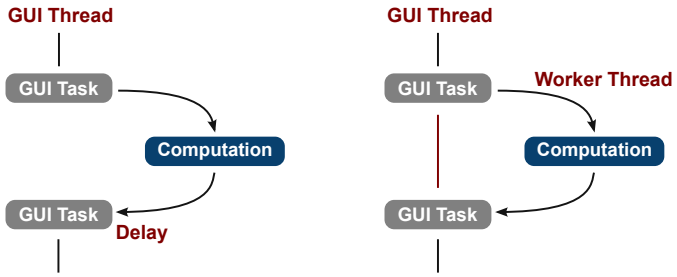


Figure 6.3: The concept of outsourcing computationally intensive implementations to a separate thread is shown. **Left:** GUI tasks are delayed by the computation, if the GUI thread is used for the computation. **Right:** GUI tasks can be processed during the computation, if the computation is outsourced to its own worker thread.

6.1.2 Data Communication

Similar to the component execution framework, a data communication layer is required to enable modules to share their data. Otherwise, the framework would be robbed of the fundamental ability to combine the specialized functionalities of the modules, significantly reducing reusability as, for instance, data generated by a module can not be accessed by another one. However, where in the other case communication was restricted between the components (modules), an interactive simulation framework also requires the framework itself to have access to the data for visualization purposes. This adds an additional indirection to the data handling, thus requires a different approach than the introduced socket-based communication layer (Section 5.3.2).

[1] In this case, *by itself* relates to the case where a thread is manually tasked to stop processing the computation and process required GUI tasks. However, such a mechanism is intrusive, clearly being a drawback for modules wrapping external application, as it requires knowledge of the code base which cannot be assumed.

Also, modules must be given control to handle the data destined to be used by other modules differently than the data used for visualization. Transferring data to the visualization backend typically requires additional copy operations, which become more significant with larger problem sizes. Furthermore, potentially not all data of interest to other modules makes sense to visualize, for instance, auxiliary data required for enabling the simulation in the first place. Overall, to minimize data transfer operations, the data communication among modules as well as between the modules and the visualization backend has to be separated.

Also, the raw data has to be coupled with additional meta information to allow for specialized visualization and to enable modules to handle input data accordingly. For example, data can be represented by tensor fields of different levels, such as scalar or vector fields and can be associated with different mesh elements such as tetrahedra, triangles, or points. This requires the meta information to contain parameters indicating the nature of the data, allowing the visualization mechanism to use the appropriate rendering method.

6.1.3 Graphical User Interface

Modules of an interactive simulation framework can provide their own specific GUI, allowing end users to conveniently interact with the module. More explicitly, providing a GUI is optional meaning that a module must not be forced to provide a frontend[2]. This would allow, for instance, module developers to test module backends before devoting efforts into the development of a frontend. Overall, these specialized GUIs must be presented to the end user in a consistent manner, meaning that the module-specific frontend is embedded into the main framework's GUI upon the end user's behest.

The framework must also support the utilization of several modules and the monitoring of the modules' readiness state. In this context readiness relates to the availability of the potentially required input data of a module. More concretely, only modules which have their input dependencies satisfied can be made active, meaning that the end user is allowed to interact with the corresponding module. The framework is required to automatically evaluate the readiness state of each module and activate/deactivate them accordingly. In turn, each module must be given the ability to investigate whether the available data provided by other modules is suitable to satisfy its specific requirements and, based on this evaluation reports, its state of readiness to the framework.

Also, simulation tools usually generate output messages during the execution, such as the convergence information during a solver process. The framework must provide a centralized message output element to provide the end user with a centralized spot of output messages, further increasing usability. In addition, the ability to reroute output streams, such as C++'s `std::cout`, to such an output window is required. Such a mechanism allows to non-intrusively transfer output data from a simulation module to the framework's centralized message element.

6.1.4 Data Visualization

Data visualization for CSE-based simulations is a fundamental capability as it enables to analyze the simulation results and thus allows to ultimately derive conclusions, further driving the research.

[2] In the context of an interactive simulation framework, a frontend refers to a GUI.

Therefore, data visualization capabilities are a central aspect of an interactive simulation framework. The diversity of simulations requires flexible visualization tools, such as three-dimensional rendering methods supporting, for instance, scalar- or vector-field visualization on top of a mesh. Especially important is to give the individual framework modules full access to the rendering backend, as the visualization demands for all future modules are not foreseeable. For instance, simulation tools might require a simulation-specific visualization especially for three-dimensional data, such as a combination of rendering algorithms to reveal the behavior inside of the simulation object.

Also, convenient selection of a specific quantity to be rendered is required as well as different mesh representations, e.g., wireframe[3]. The latter can be coupled with automatic mesh evaluation algorithms aiding an advanced user, referring to a user with an additional background in mesh generation, in the task of evaluating the mesh quality. These additional investigation methods further underline the importance of a modular simulation environment, capable of attaching non-simulation modules, such as mesh generation and evaluation modules, to the simulation platform.

Aside from supporting three-dimensional rendering, support for chart visualization is essential. Chart visualizations enable to investigate a set of quantities relative to a reference quantity. For example, a current-voltage characteristics enables to judge a device's performance. Vital to such a mechanism is the ability to access data generated from several simulation runs to, for instance, compare a set of current-voltage characteristics for the same device but for different doping profiles. The data from the individual simulation runs should be visualized in the same chart, enabling the end user to relate the quantities to each other.

In general, essential to visualization is the ability for comparing results. For instance, two different charts are compared or two different render views of the same mesh are displayed simultaneously. This requires the visualization backend to support an arbitrary number of visualization windows, in an arbitrary combination. Exemplary combinations would be three rendering windows and two chart windows as well as two rendering windows and four chart windows. Related to this mechanism is the ability to analyze data in a multi-monitor environment. Today's typical workstation setups are increasingly equipped with more than one monitor, introducing the ability to utilize this setup for visualization needs. For instance, where two chart views are displayed on one monitor, one rendering view is displayed on the second. While this requirement seems trivial, it introduces substantial design considerations concerning the general GUI platform of an interactive simulation framework and is thus vital to be explicitly stated as a requirement.

Simulation tools may generate a series of simulation results which has to be supported by the visualization backend. For instance, a device simulator performs a range of simulations according to a range of contact values, allowing to determine the current-voltage characteristics of a device. In such a case, the visualization backend must support manual or automatic stepping through the simulation results. Where the first allows to specifically analyze results, the latter enables a visualization in a movie-like manner, further improving the perception of the simulated physical processes.

[3] The term wireframe relates to visualizing the mesh's elements without coloring the interior of the elements, such as areas and volumes. The thus remaining vertices and their individual connections - visualized as lines - represent the wireframe.

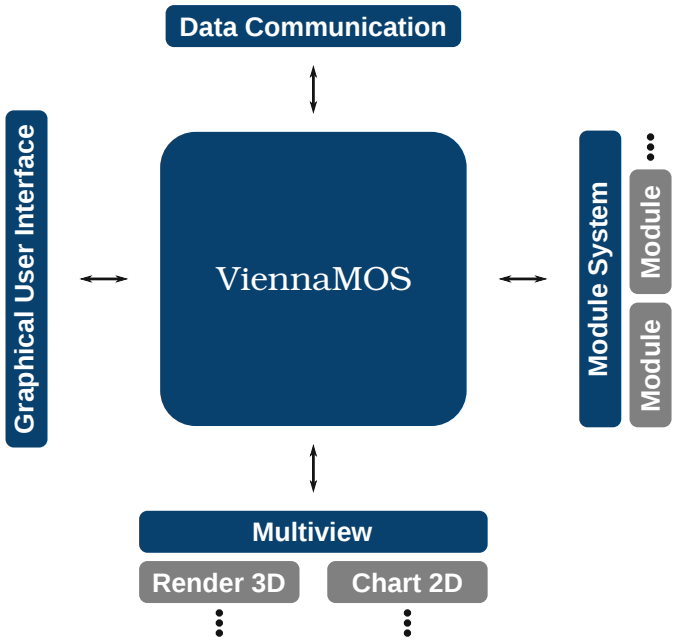


Figure 6.4: The ViennaMOS general software design; The module system allows interfacing with reusable, exchangeable, and expandable modules, using a unified interface. Data communication between modules is realized with a central data storage. A flexible and modular GUI provides access to the individual modules as well as to the visualization mechanism. The latter is based on the so-called multiview facility, enabling access to an arbitrary combination of three-dimensional rendering and two-dimensional chart visualization backends.

6.2 The ViennaMOS Project

The FLOSS-based ViennaMOS project [11] provides an interactive simulation platform applying CBSE (Section 3.3) as well as utilizing LCSD-based libraries (Section 3.4) in combination with a modular GUI and modern visualization methods. The modular concepts enabled by the CBSE approach are extended to the GUI as each module offers its own specific end user interface. By using these approaches ViennaMOS tackles the discussed challenges for an interactive simulation framework (Section 6.1).

The framework's code base benefits from the extensive functionality provided by the VTK library [76] for the visualization backend and the Qt library [64] for the frontend. The design - especially with respect to the visualization capabilities and the modular GUI - is significantly influenced by the FLOSS-based visualization software ParaView [63].

ViennaMOS is composed of four essential parts, those being the data communication layer, the GUI, the module system, and the visualization facility represented by the so-called multiview mechanism, providing access to three-dimensional rendering and two-dimensional chart visualization backends (Figure 6.4). Overall, the four key parts tackle the requirements and challenges introduced by implementing an interactive simulation framework (Section 6.1).

The following sections give a detailed overview of ViennaMOS.

Section 6.2.1 discusses the data communication layer based on a central data storage. Section 6.2.2 introduces the three-dimensional rendering facility. Section 6.2.3 depicts the two-dimensional chart visualization ability. Section 6.2.4 describes the multi-view mechanism, providing arbitrary combinations of visualization windows. Section 6.2.5 discusses the module system, in particular the interface. Section 6.2.6 characterizes the main GUI of ViennaMOS. Section 6.2.7 gives two application examples, by discussing a device generator and a device simulator module and their interplay.

6.2.1 Data Communication

The single-execution mode and the fact that the set of utilized simulation modules used in an active ViennaMOS simulation is significantly smaller than the usual scenarios of large-scale component execution frameworks simplifies the data communication requirements considerably. Therefore, to keep the implementation simple - yet effective - and also highly maintainable due to a simplified code base, a straightforward centralized *database* is utilized by ViennaMOS. Each module stores and accesses data in this database, which is in turn governed by the framework. This storage is solely used to exchange data between the individual modules and is not related to the visualization backend.

To ensure flexibility with respect to the supported storage types, a storage setup based on an associative container mapping a string-based key with a generic object pointer is utilized. The generic object pointer is implemented via a `void` smart pointer, provided by the Boost Smart Pointers library [51]. The use of smart pointers ensures that upon destruction the destructor of the stored, type-erased object is called appropriately, thus eliminating memory leaks.

6.2.2 Three-Dimensional Render Visualization

An essential ability of simulations in the general field of CSE is to visualize simulation results. Therefore, special consideration has been given to provide a flexible three-dimensional rendering backend[4]. To this end the FLOSS-based VTK library is utilized as a visualization backend of a dedicated rendering class(Section 2.3), directly utilizable in Qt-based GUI applications. The rendering class provides an interface for ViennaMOS and its modules to the VTK-based visualization backend. Also, a render editor GUI is available, providing the advanced users the ability to customize the rendering (Figure 6.5). Note that the central VTK-based mesh storage is not related to the previously introduced ViennaMOS communication database. The mesh storage is solely used for the VTK-based visualization and is thus enforced to be a VTK object. Therefore, no generic storage - as provided by the ViennaMOS' communication database - is required, otherwise unnecessarily complicating the mesh storage access.

As already indicated, the support for multi-segment meshes is essential for a simulation framework, as different areas of a simulation domain might require a different handling. For instance, different materials might be assigned to the individual segments.

[4] Although the name of the rendering facility indicates sole support for three-dimensional objects, one- and two-dimensional objects are also supported.

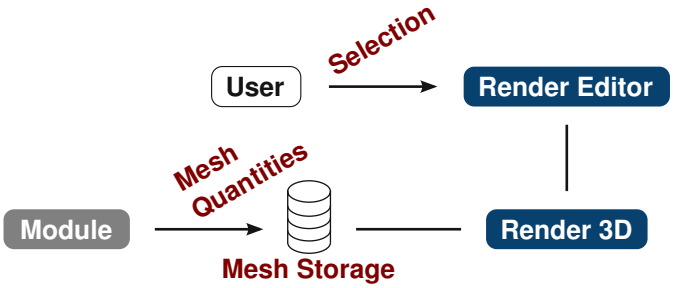


Figure 6.5: The schematic process of the three-dimensional rendering mechanism; A ViennaMOS module stores data - mesh and/or quantities - in a central VTK mesh storage, dedicated to store mesh objects via a `vtkMultiBlockDataSet` object. An end user interacts primarily with an editor to select and tune the visualization. Upon user input, the renderer accesses the required data from the mesh storage and performs the rendering in the GUI.

Although using a multi-segment mesh for visualization is not required - as a multi-segment mesh can also be mapped to a single-segment mesh - it allows to provide segment-specific visualization options, such as segment-wise coloring or visibility adjustments. These features further improve usability, as, for instance, the framework can aid the end user in identifying individual segments of a mesh, especially important for intricate simulation devices. To this end we use VTK's `vtkMultiBlockDataSet` data structure, enabling to hold an arbitrary set of VTK meshes, such as `vtkUnstructuredGrid` and `vtkStructuredGrid` (Figure 6.6). For each mesh segment, a `vtkActor` and a `vtkDataSetMapper` is stored, allowing to perform segment-specific visualization tasks, such as coloring according to vertex-based scalar fields.

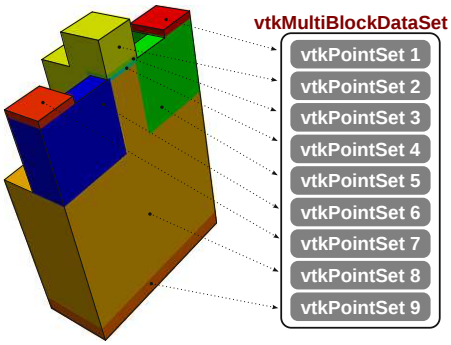
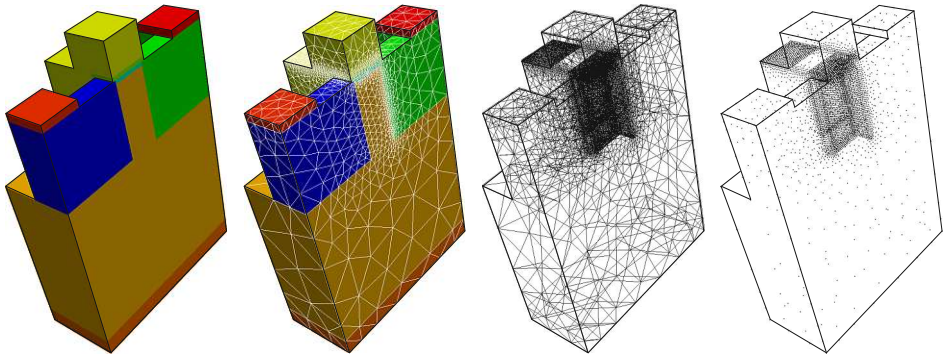


Figure 6.6: ViennaMOS' render class stores the individual segments of a mesh - indicated by colors - as `vtkPointSet` objects, capable of holding, for instance, unstructured and structured grids. A `vtkMultiBlockDataSet` is used to store the set of `vtkPointSet` objects.

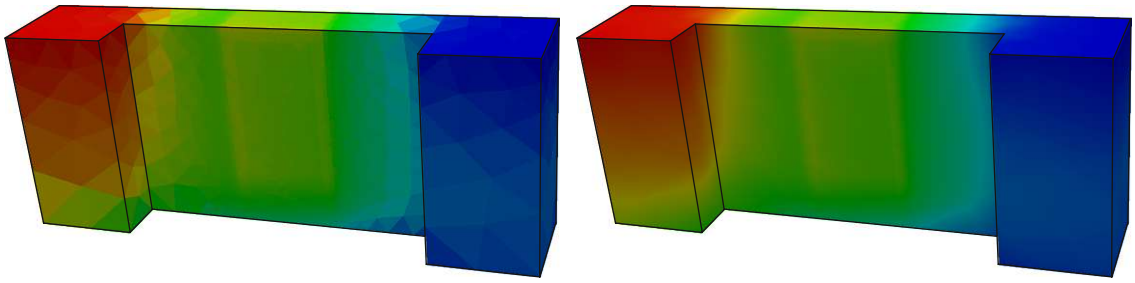
Different mesh representations are supported natively by the VTK library, in particular surface, surface with edges, wireframe, and point representations, enabled by using the corresponding `vtkActor` methods (Figure 6.7).



(a) Surface (b) Surface with Edges (c) Wireframe (d) Points

Figure 6.7: ViennaMOS supports different mesh representations, those being surface, surface with edges, wireframe, and points.

Visualizing data fields on top of meshes requires the data to be stored on the respective `vtkPointSet` objects - representing mesh segments - collectively governed in the central `vtkMultiBlockDataSet` object. In particular, the data fields have to be stored as `vtkDataArray` objects and linked to the corresponding `vtkPointSet`'s point or cell data container. Note that VTK natively only supports point or cell data. Due to the support for mapping data fields on different mesh elements, specific meta information is required by the render class indicating, for instance, the mesh element association of a particular data set, i.e., whether the data to be visualized has to be mapped on vertices or on cells. Overall, ViennaMOS supports cell-based and vertex-based quantity visualization of scalar fields (Figure 6.8).



(a) Cell Quantity (b) Vertex Quantity

Figure 6.8: ViennaMOS supports cell and vertex quantities. A cell-based (**left**) and vertex-based (**right**) quantity distribution is shown. Where with the first the individual mesh elements can be clearly identified, the latter offers an automatic color smoothing mechanism provided by VTK's vertex-based coloring backend.

Simulation tools might generate a series of simulation results. For instance, simulating the characteristics of a device yields for each applied bias not only the current at the terminal, but also the actual quantity distributions, like potential, electron carrier, and hole carrier distributions. Therefore a playback mechanism has been implemented allowing to step through the individual simulation results.

More concretely, all results are stored on the multi-block data structures associated with a unique ID relating to the sequence position of the result. ViennaMOS can thus order the renderer to show a specific result out of a sequence via generating the corresponding ID. This mechanism not only allows to step forward and backward, but also enables ViennaMOS to provide an automatic playback mechanism enslaved to a time-delay used between rendering the individual simulation result.

6.2.3 Two-Dimensional Chart Visualization

ViennaMOS provides a two-dimensional chart plotting facility based on the VTK library. The key mechanisms are provided by the `vtkChartXY` and the `vtkTable` classes. The end user is exposed to the chart mechanism via a chart editor, allowing to customize the visualization of data generated by modules. Figure 6.9 depicts the utilization of the plotting tool.

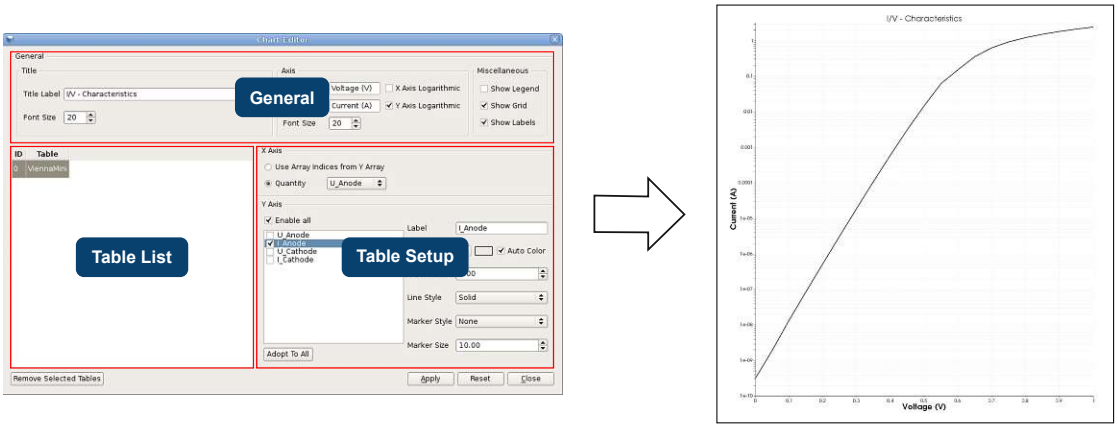


Figure 6.9: ViennaMOS' two-dimensional chart plotting ability; **Left:** The editor provides general properties (`General`) such as the chart title as well as support for visualizing a set of results in a single chart by providing the set of stored tables (`Table List`). Selecting a specific table in the `Table List` raises the respective `Table Setup` section. **Right:** Based on the editor properties, a chart is rendered.

With respect to the implementation, a simulation result is added to the facility by inserting a new `vtkTable` into the storage along with meta information, such as the quantity name. More concretely, `vtkDataArrays` are used to store the individual data sequences, for instance, the computed current values and the applied voltage values of a device contact. These arrays are used to populate a `vtkTable` which is in turn forwarded to the framework, which provides access to all available chart rendering instances (Figure 6.10). Similar to the previously discussed three-dimensional rendering mechanism, the central VTK-based table storage is not related to ViennaMOS' communication database. The requirement of supporting the comparison of different simulation runs is implemented in a straightforward manner, by using a set of tables, one for each simulation run. This set is accessed by the chart editor, and in turn used to update the GUI's `Table List`.

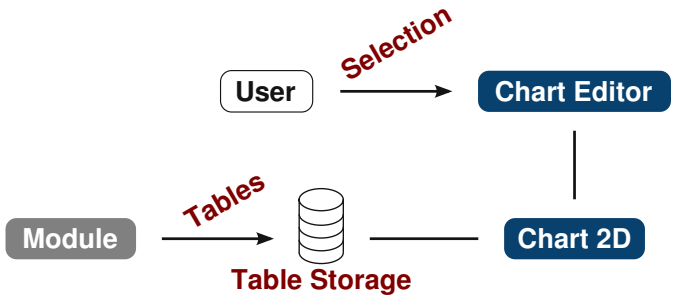


Figure 6.10: The schematic process of the two-dimensional chart visualization mechanism; A ViennaMOS module stores data - tables - in a central VTK table storage, based on a container of `vtkTable` objects. An end user interacts primarily with an editor to select and tune the visualization. Upon user input, the chart visualization backend accesses the required data from the central table storage and performs the rendering in the GUI.

6.2.4 Multiview

ViennaMOS allows to use several instances of the previously introduced rendering and chart visualization simultaneously via the so-called *multiview* mechanism (Figure 6.11).

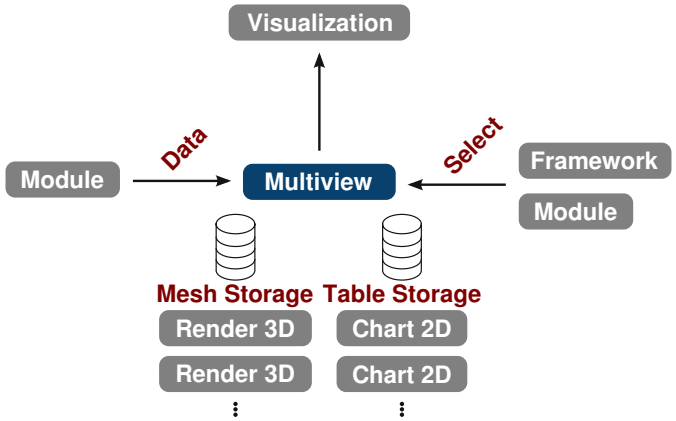


Figure 6.11: The multiview facility centrally governs the data received by the individual modules and distributes it to each of the utilized render and chart views. The framework and the modules interact with multiview's interface, in turn orchestrating the corresponding updates to the visualization.

The key mechanism is based on a polymorph `view` entity, which - upon input from an end user - can be transformed into either a three-dimensional render view or into a two-dimensional chart view. The essence of the provided comparative views is based on splitting available views in a horizontal or a vertical manner at the behest of the end user. This splitting is natively supported by Qt by embedding each view into a `QDockWidget`. The following code snippet depicts the simplicity of the splitting mechanism.

```
1 QDockWidget* dock = new QDockWidget;
2 splitDockWidget(multi_view->getCurrentDock(),dock,Qt::Horizontal);
3 multi_view->addNewView(dock);
```

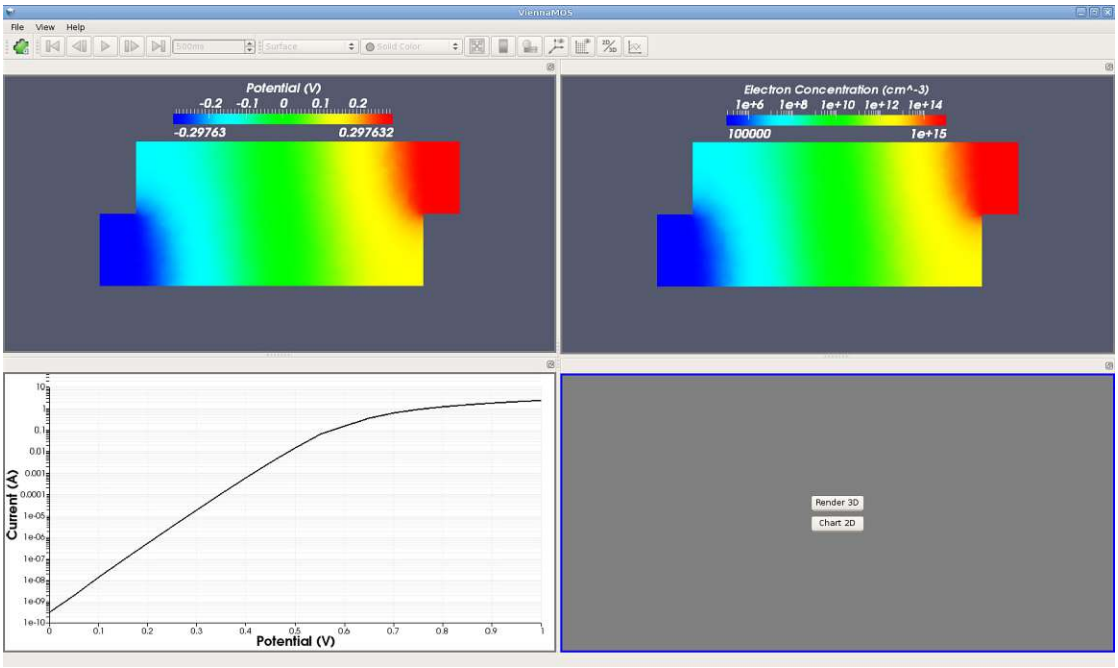


Figure 6.12: An exemplary multiview setup is shown consisting of two render views (**top**), one chart view (**bottom left**), and one polymorph view (**bottom right**). Representative simulation results based on a two-dimensional pn-diode - already discussed in Section 4.3.10 - are used to depict the multiview's GUI. In particular, the potential (**top left**) and electron distribution (**top right**) as well as a current-voltage characteristics (**bottom left**) are visualized in a comparative manner. The polymorph view can be transformed into a render view or into a chart view by the end user.

A new `dock` widget[5] is generated (Line 1). The `multi_view` object provides the currently selected `QDockWidget`, which is split horizontally - vertical splitting is similarly available - initializing the newly created `dock` object (Line 2). The new dock object is forwarded to the `multi_view`, being preloaded with a polymorph view entity. This particular view provides the end user with button widgets enabling to transform the view either into a three-dimensional render view or into a two-dimensional chart view (Figure 6.12). As a convenient side-effect, a `QDockWidget` - as the name suggests - can be detached from its parent widget, effectively allowing to natively provide the required support for multi-monitor setups.

The multiview facility centrally stores the data received by the individual modules, such as meshes and associated quantities (in Figure 6.5 referred to as mesh storage) as well as data tables (in Figure 6.10 referred to as table storage). When a new view - render or chart - is created, the data objects are copied to the new view states. Although this introduces memory and copy transfer overhead, it ensures that no *crosstalk* between the individual views occurs. Concerning crosstalk, quantity visualization of the VTK meshes is based on activating a single specific data array previously stored on the mesh.

[5] A widget refers to a GUI element, such as a button. In Qt all GUI elements are widgets, due to the applied object-oriented approach (Section 3.1), supporting dynamic polymorphism essential for adapting a GUI application during run-time.

Consequently, if several views operate on the same centrally stored mesh, each attempting to visualize different quantities on top of the mesh, the views turn each other's selections on and off, thus introducing the aforementioned crosstalk.

6.2.5 Module System

The core of ViennaMOS' module system is based on Qt's plugin mechanism, providing convenient facilities to extend an application with additional functionality. Therefore, each module is implemented by a plugin, similar to the plugin system discussed with the component execution framework ViennaX (Section 5.3). Each module has access to the multiview facility, enabling direct access to the rendering backends. Although this approach introduces visualization responsibilities with a module, it enables a module to setup its own specific visualization needs by, for instance, applying a series of VTK algorithms to a quantity dataset prior to rendering the result.

Each module implementation has to adhere to a specific class interface, induced by the applied virtual plugin class hierarchy. The interface enables the individual module implementations to, for instance, provide general information on each module, such as the name, description, and version of the module. Also, a readiness method is used by the framework to evaluate whether all input dependencies of a module are satisfied. Each module is thus required to perform the required check routines defined by the module developer, for instance, testing whether the available data stored in the central database (Section 6.2.1) can be processed by the respective module. An update method enables to access a possibly updated state of the database triggered by the execution of other modules. For instance, another module stored new data into the database, therefore the update method allows for reevaluating whether the new data is suitable.

A reset method gives the module the opportunity to fallback into an initial, defined state. Such a mechanism is required to clear the module without reloading it. Furthermore, each module's GUI provides an execution button which executes the module by calling the corresponding execution method. Usually, external simulation code is wrapped and executed in this method, thus this particular method contains implementations with potentially significant run times.

A quantity meta-information retrieval method enables the framework to access the information on the module's visualization quantities, i.e., quantities which are copied by the module to the visualization backend as implemented by a developer and are thus available to the end user - via a drop-down widget containing the individual quantities - for selection. In this context, each module is responsible to transfer its result data, i.e., result quantities such as a potential distribution, into ViennaMOS' visualization data storages - implemented by VTK objects - centrally governed by the multiview facility (Section 6.2.4). More concretely, this step requires copying data from module-specific data structures to VTK objects, such as a `vtkTable`. This is required as it is to be expected that the module wraps an external simulation tool, offering its own specialized data structure. Therefore, only the module has the *knowledge* - implemented by the module developer - about the utilized data structure and is thus in the unique position to transfer the data appropriately.

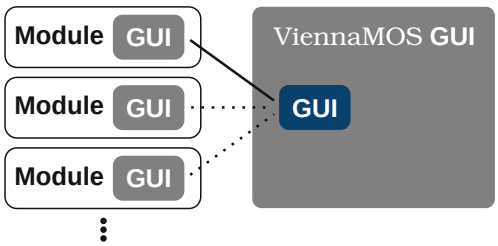


Figure 6.13: Each module provides its own specific GUI being presented to the end user as part of the central ViennaMOS GUI.

The fact that the data transfer to the visualization backend is explicitly coded with respect to VTK, is an acceptable limitation, as the visualization backend is based on VTK. However, utilizing a different rendering backend, such as the Visualization and Computer Graphics library [172] would require a transfer to the corresponding native data structures. Providing access to different rendering backends would trigger further investigations regarding a generic visualization backend data transfer mechanism to further decouple the implementations and thus by extension the maintainability and extendability of ViennaMOS.

Coupled to this data transfer is a registration mechanism, meaning that each module registers the transferred quantity by providing the quantity name, the unit, the cell-level (cell-based or vertex-based data), the tensor-level, e.g., scalar field, and the module name. The latter enables the framework - or other modules - to associate the data with a specific module at a later point of the simulation process. In essence, the registration is based on a set of quantity registration objects, each holding the required meta-information for a specific quantity. The framework has access to this set via the module interface and can thus perform data-agnostic specializations, for instance the main GUI can therefore use a different icon for indicating vertex- or cell-based data. Also, this registration mechanism allows the rendering backend to determine the appropriate visualization technique, for instance, map the quantities on the mesh vertices (Section 6.2.2).

With respect to the module loading mechanism, Qt's `QPluginLoader` class provides the essential mechanisms. Each module implementation is required - aside from sticking to the mentioned virtual class interface - to be a valid Qt object, i.e., a class which not only derives from `QObject` but also uses the `Q_OBJECT` macro in the class's state, automatically generating boilerplate code required to, for instance, enable the class to utilize Qt's signal/slot system. By adhering to this Qt-specific implementation guideline, no manual factory mechanism has to be implemented, contrary to the previously discussed component execution framework (Section 5.3.2).

As previously mentioned, each ViennaMOS module potentially provides its own GUI, being plugged into the framework's main frontend upon the selection of the corresponding module (Figure 6.13). As all Qt GUI elements derive from Qt's `QWidget`, forwarding a module-specific frontend from the module to the main GUI is straightforward. If indeed the module does not provide a frontend, an empty widget is used. Overall, typically each module provides two classes, one implementing the module interface and the other implementing the GUI, providing a `QWidget` object containing the GUI.

A module can outsource computationally intensive parts to a separate thread by using Qt's `QThread` facility. In essence, the computational intensive part is outsourced to a new *worker* object, modeling Qt's `Q_OBJECT` concept. The new class is associated with a new `QThread` instance, execution controls, e.g., which method of the worker object has to be called when the thread starts up, are linked via Qt's signal/slot mechanism, and finally the thread is executed (Figure 6.14). This approach can be further simplified by providing macros, as the required code is to a large extent boilerplate code.

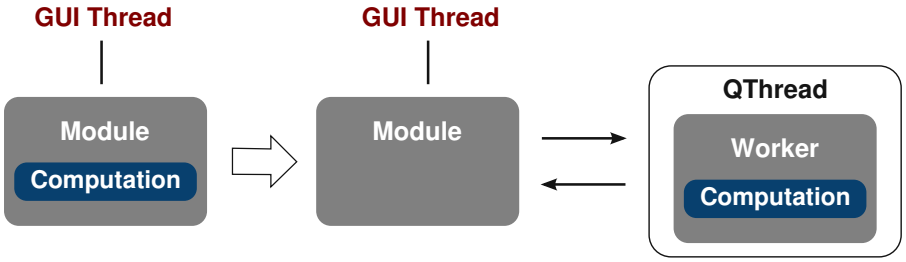


Figure 6.14: Computational intensive parts can be outsourced from the GUI thread to a separate thread, thus keeping the GUI responsive during extended computations. The computation part is moved to a new class (`Worker`), modeling Qt's `Q_OBJECT` concept. The worker is then assigned to a new `QThread` and executed separately from the main GUI thread.

6.2.6 Graphical User Interface

The GUI of ViennaMOS is the central interaction platform, providing the end user access to the rendering facilities and the modules. It makes use of the previously discussed flexible GUI elements, in particular the module system as well as the multiview facility with its three-dimensional rendering and two-dimensional chart views. All main elements of ViennaMOS' GUI are based on `QDockWidget` objects, enabling to decouple the containing widget from the main frontend and move it, for instance, to a secondary monitor (Figure 6.15).

ViennaMOS provides the means to automatically discover valid modules on the local filesystem, to load them, and to provide a list of discovered modules to the end user via the so-called *module manager*. The module manager is in fact a dialog enabling the end user to select individual modules to be used in the current session. Selected modules are then loaded, meaning that the module's factory mechanism is used to instantiate a module, and are finally listed in the *active module list* where they can be selected to show the module-specific GUI (Figure 6.16).

This active module list allows selection-based switching between the individual module GUIs. Only modules the input dependencies of which are resolved, can be interacted with by the end user. As already indicated, each module is asked by the framework to evaluate its readiness state after one module of the active set finished execution, as previously discussed in Section 6.2.5. For instance, two modules are loaded, where the first is used to generate a device representing the simulation domain, the latter actually conducts simulations requiring the presence of a device.

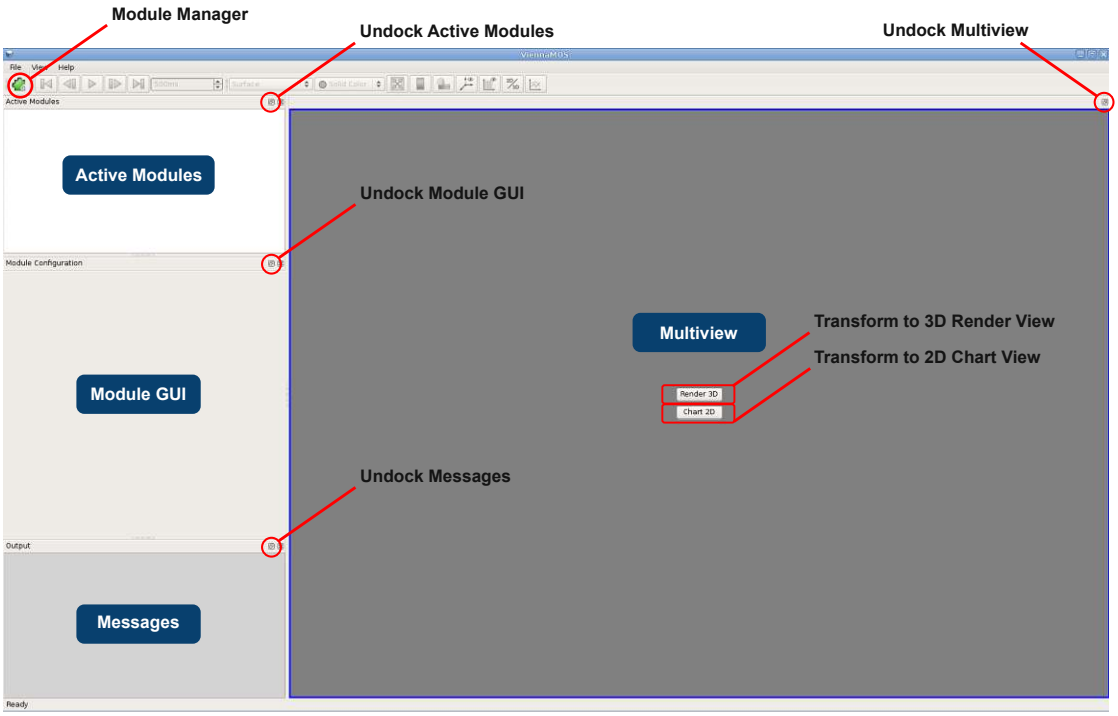


Figure 6.15: The main elements of the default ViennaMOS GUI; **Top left:** The list of activated modules is provided, which is being populated by using the module manager dialog. **Middle left:** The module-specific GUI is shown based on the selected module in the active modules window. **Bottom left:** An output message window gathers all messages, both from the framework and from the modules. **Middle:** A multiview environment holds a set of views, where each view can be specialized to either hold a three-dimensional render view or a two-dimensional chart view. All four main elements can be undocked from the main window, enabling, for instance, to move the respective windows to a secondary monitor.

Therefore, until the first module stored a suitable simulation domain in the central database, the second module remains inactive. Only if a valid domain has been stored, the second module becomes active and thus enables the end user to interact with it. This mechanism ensures that the end user interacts only with modules which can be actually executed, due to the resolved dependencies.

The output messages generated by the framework and by the modules are collectively presented to the end user via the central `Messages` window. The implementation is based on extending Qt's `QPlainTextEdit` class by additional methods to claim and release C++ streams, based on rerouting the respective buffers.

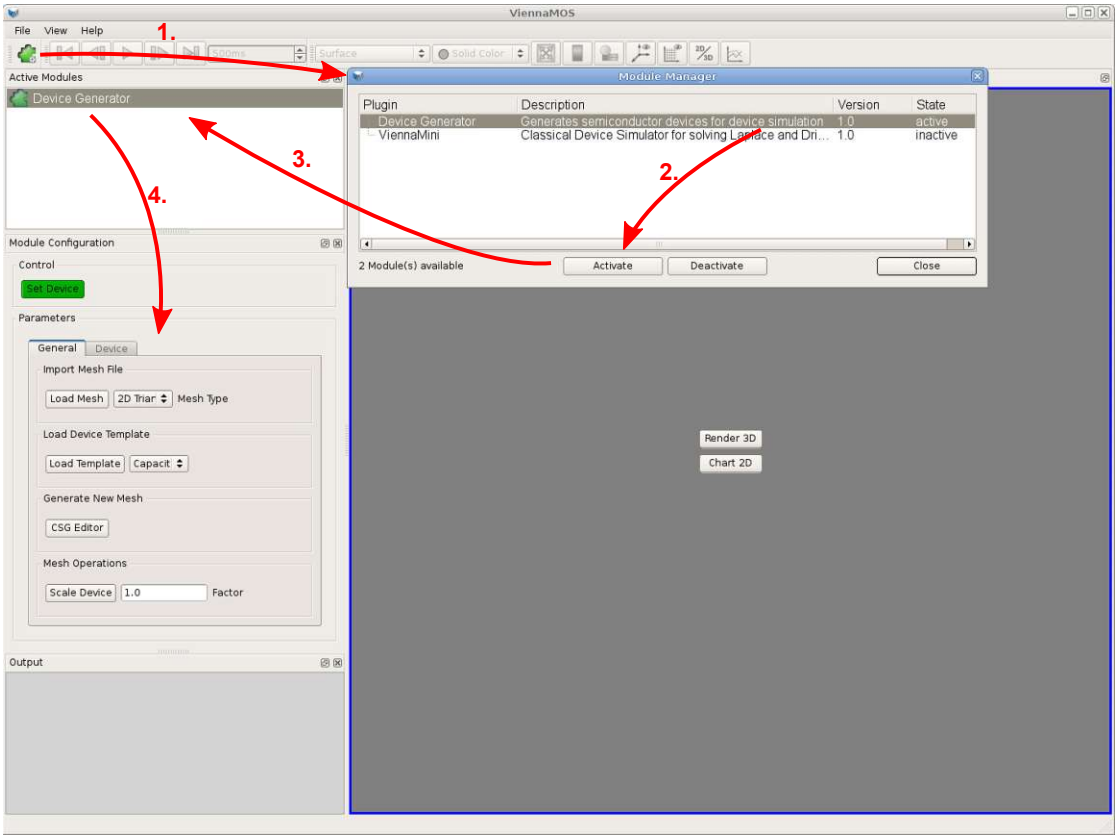


Figure 6.16: Triggering the module manager dialog (Step 1) allows to select modules used for the simulation (Step 2). When a module is selected - in this case the so-called `Device Generator` - it can be activated, which instantiates the module in the background and adds it to the list of active modules (Step 3). An active module can be selected, loading the module-specific GUI into the main frontend (Step 4).

6.2.7 Examples

In this section, two representative application examples are discussed based on two ViennaMOS modules. The general design and usage of the modules is depicted as well as the basic utilization of the ViennaMOS GUI and visualization backends. More concretely, the already mentioned device generation module is introduced which provides different ways to generate a simulation domain. Upon the module's execution, the generated device is stored in the central database. The second module, a device simulation module based on the previously introduced device simulation framework ViennaMini (Section 4.3), utilizes the device generated by the device generation module to conduct a device simulation.

These examples further underline the advantages of the decoupled approach discussed throughout this thesis as depicted in Figure 6.1. Not only libraries are reused by the individual modules, but also another framework - ViennaMini - is utilized via ViennaMOS modules. Therefore, additional development overhead is drastically minimized, as already available implementations, such as mesh generation or device simulation facilities, are reused.

Device Generation Module

In this section a device generation module is discussed providing the means to generate devices for device simulations via the ViennaMini framework (Section 4.3.3). This example also underlines the versatility of ViennaMOS modules, as this particular module is in fact not a simulation module, but a modeling tool required to characterize the simulation domain. To this end, the module requires mesh generation capabilities and the ability to assign a doping profile and segment roles. Overall, the module utilizes the various device generation functionalities enabled by interfacing with external tools, such as ViennaMini, and provides a flexible GUI for the process, further underlining the benefit of rigorously applying a LCSD approach.

The device generation module provides three mechanisms to generate a device. Where the first mode loads an externally generated already meshed structure enabling the end user to assign segment roles as well as a doping profile, the second mode allows to perform an in-place mesh generation for a desired device structure via a constructive solid geometry (CSG) language (primarily of interest to advanced users) also requiring the end user to provide segment roles as well as doping information. The third mode utilizes a device template mechanism, already providing default segment and doping information. Figure 6.17 schematically depicts the individual device generation modes.

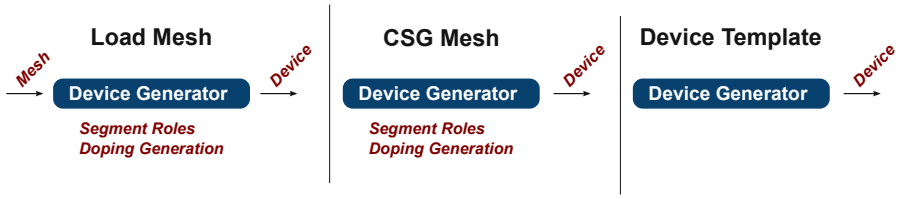


Figure 6.17: The device generation module provides three modes; **Left:** An externally generated meshed structure is loaded via ViennaGrid, requiring the assignment of segment roles and the generation of a doping profile. **Middle:** A CSG feature provided by ViennaMesh's Netgen backend allows to generate meshed structures on the fly via an expressive language. Upon completion, the segment roles and a doping profile must be assigned. **Right:** The device template mechanism of ViennaMini is used to generate ready-to-simulate devices. No further steps are required, as the template feature already assigns segment roles as well as a doping profile.

Figure 6.18 depicts the module's GUI and the basic process flow. The end user selects one of the three device generation mechanisms. As soon as the structure is loaded, the render window visualizes the mesh. Finally the GUI provides the means to assign segment roles including doping levels for semiconductor segments. When the end user is finished, the device can be stored into ViennaMOS' central data storage, so other modules can access it. In this particular case, a FinFET device is prepared, offering the same properties as depicted in Section 4.3.10. This device is used in the subsequent device simulation steps, therefore it is stored in the central database.

Aside from the already discussed structure loading and device template mechanism, a peculiarity of this module is the CSG functionality. ViennaMesh provides an interface to Netgen's three-dimensional CSG backend, enabling on-the-fly generations of meshed structures inside the device generation module. Due to the applied LCSD approach, the module's GUI has to provide merely a text input field - holding the CSG commands - which is forwarded to the CSG backend via ViennaMesh.

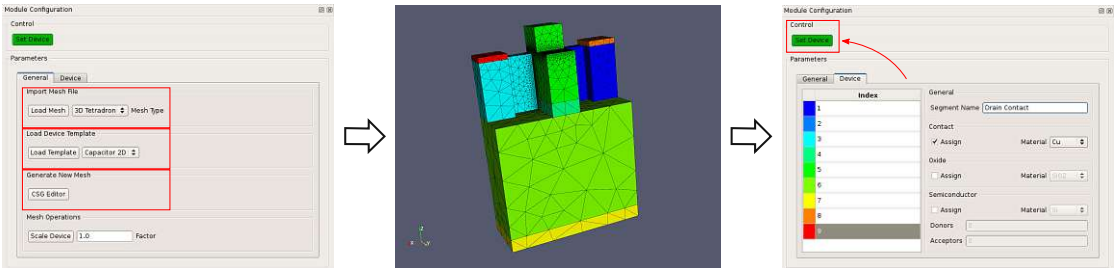


Figure 6.18: General process flow of the device generation module; One of the three generation modes is used to load a meshed structure (**left**) - in this case representing a FinFET device - and visualized in the render window (**middle**). When the structure has been loaded, the segment roles and doping profiles are assigned (**right**). Each segment can either be a contact, an oxide, or a semiconductor, the latter allows to assign an initial acceptor and donor doping value. All segments support the assignment of a specific material. When the end user finished entering the segment roles and doping information, the final device can be stored in ViennaMOS' central database (red arrow), so other modules can access it and further utilize it.

In turn, ViennaMesh takes care of transferring the generated mesh into an appropriate ViennaGrid mesh object. Figure 6.19 depicts the utilization of the CSG mechanism.

The implementation of the device generation module is based on two classes, separating the GUI from the actual module implementation interfacing with external libraries (Figure 6.20). The GUI forwards the input data to the module according to the chosen generation mode. For instance, if the CSG mode is used, a string containing the CSG information is passed. The module makes use of synergy effects in particular by interfacing with ViennaGrid, ViennaMesh, ViennaMini, and ViennaMaterials for loading and generating meshes, accessing material data as well as using the device template mechanism, respectively. The thus received data is used to update the GUI, allowing the end user to customize the device by, for instance, updating the segment roles and doping information. Finally, if the end user is finished, the device is forwarded to the framework, where it is stored in the central database for other modules to access it.

Device Simulation Module

This section introduces a device simulation module providing basic device simulation capabilities. The module is a wrapper for the previously introduced device simulation framework ViennaMini (Section 4.3), further underlining the benefit of decoupling functionality into reusable tools by the LCSD approach. Also, the fact that this module is merely a wrapper puts the focus of this section on the module itself rather than discussions regarding the simulation results.

With respect to the implementation, the device simulator module interfaces with ViennaMini and provides a GUI for the provided API (Figure 6.21). The device simulation module becomes usable, i.e., it can be selected in the active module list of the main ViennaMOS GUI (Section 6.2.6), as soon as a valid device is stored in the ViennaMOS central data storage. The device is automatically imported and the GUI is updated accordingly, enabling the end user to setup contact potentials or currents as well as particular physical models. Upon the module's execution the ViennaMini-powered simulation is conducted.

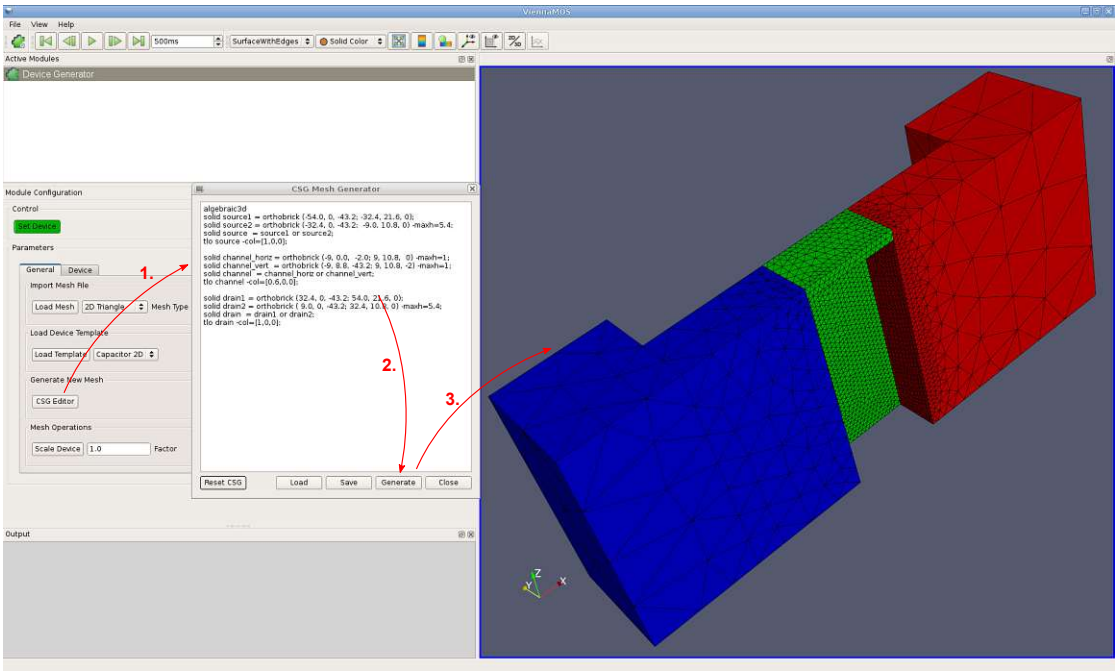


Figure 6.19: The CSG mechanism of the device generation module; Upon raising the editor, the CSG commands can be entered (Step 1). The mesh generation is manually triggered (Step 2) followed by an automatic visualization in the three-dimensional rendering window (Step 3). In this case the drain, source, and channel region of a FinFET device is modeled.

Figure 6.22 shows a basic but representative populated device simulation GUI based on the FinFET device generated by the previously executed device generation module. The GUI provides an overview of the device segments and related meta information, generated by the previously utilized device generation module. The device simulation setup is segment-based, meaning that the end user can assign contact potentials and semiconductor models, like scattering models, for each segment. The actual simulation is triggered via the corresponding execution button. When the simulation is finished, the results can be visualized by ViennaMOS' multiview mechanism (Figure 6.23). In this particular case a similar simulation - an active FinFET device - as introduced in Section 4.3.10 has been conducted. Note that the current limitations of the visualization mechanism do not allow for manual color ranges, therefore if logarithmic color rendering is required - as is the case for the electron concentration and hole concentration distributions - segments offering no quantity values are automatically assigned a value of one to enable the computation of the logarithm. This manifests with the carrier concentration distributions' renderings where the gate is assigned a value of one.

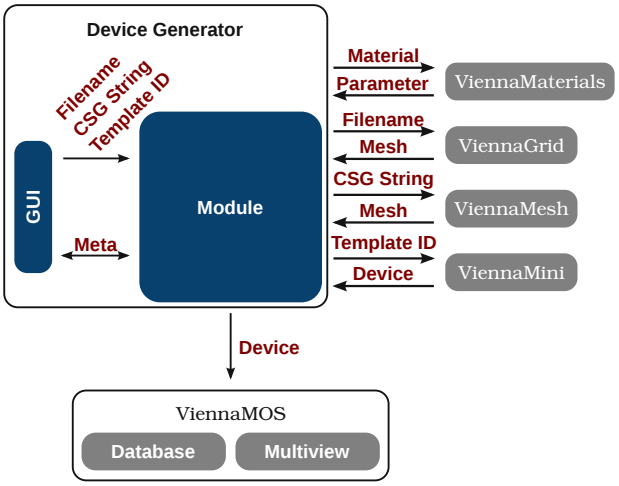


Figure 6.20: The design of the device generation module; The GUI provides the input data to the module, forwarding it to the appropriate external library to generate a mesh. In the case of ViennaMini's device template mechanism a device is already returned not requiring any additional user interaction to finish the device setup, although further customizations are supported. In the other cases - a mesh file is loaded via ViennaGrid or a mesh is generated by ViennaMesh's CSG mechanism - the end user is required to provide additional meta information, such as material information, to elevate the mesh to a device. To this end, ViennaMaterials is used to retrieve material-specific parameters, such as the relative permittivity. Finally, the device is forwarded to the framework, where it is stored and processed for visualization.

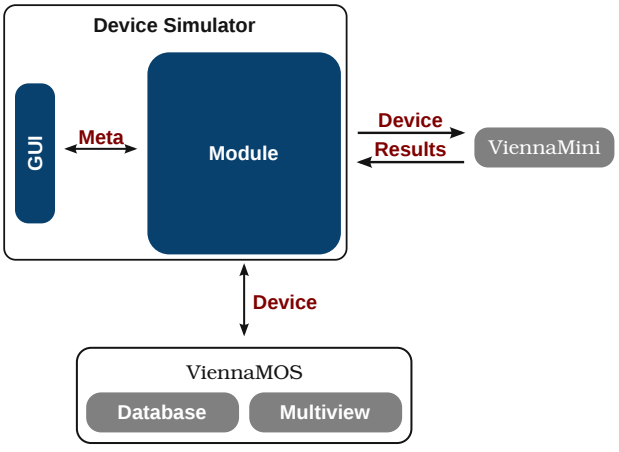


Figure 6.21: The design of the device simulation module; The module loads a suitable device, i.e., a meshed structure with assigned segment roles and a doping profile, from the ViennaMOS data storage and updates the GUI with the meta information, such as segment roles. The end user inputs simulation properties into the GUI, like the physical models to be solved, which are used to configure a ViennaMini simulation instance. After the simulation is finished, the results are being accessed and forwarded to ViennaMOS, in particular to the database and the multiview mechanism to visualize the results.

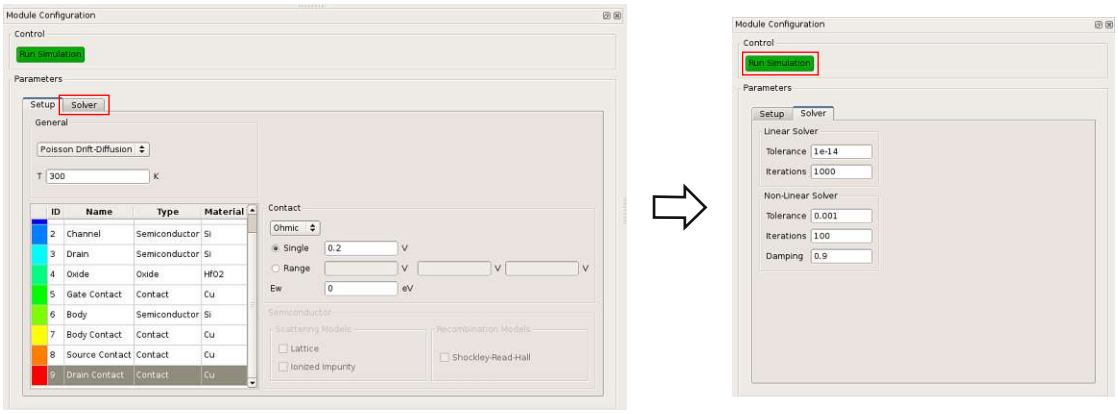


Figure 6.22: The GUI of the device simulation module; **Left:** An overview of the device generated by the device generation module is given, including segment roles, names, and materials. Contact and semiconductor segments can be customized by, for instance, assigning contact potentials and activating particular physical models, respectively. **Right:** Parameters for the solver can be customized if desired, influencing the convergence behavior. The actual simulation is started by pressing the green execution button, calling the module's execution method.

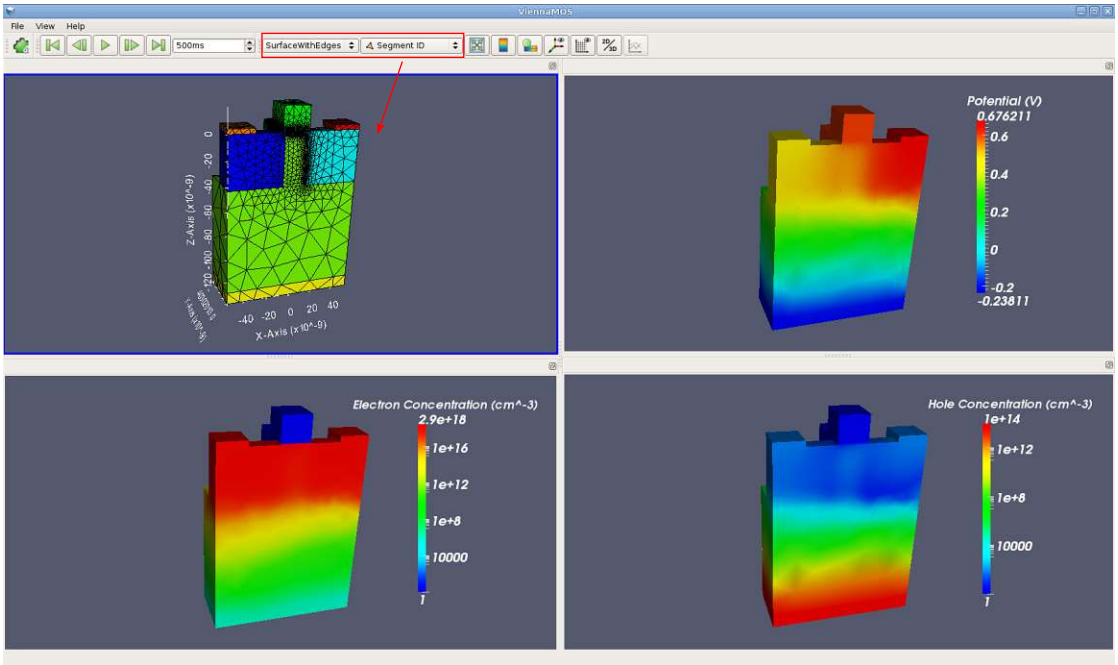


Figure 6.23: ViennaMOS' comparative visualization mechanism is used to investigate the results of a device simulation based on an active FinFET device. By selecting the appropriate render window (blue box) and the corresponding visualization properties (red box), the respective visualization is updated. **Top left:** The individual segments are colored by simultaneously visualizing the surface triangles and coordinate axes. **Top right:** The computed potential distribution is depicted. **Bottom left:** The computed hole concentration distribution is shown using logarithmic colors. **Bottom right:** The computed electron concentration distribution is visualized using logarithmic colors.

Chapter 7

Thesis Evaluation

7.1 Summary

Requirements, challenges, and approaches for frameworks relevant for the field of MNDS have been examined in detail. A discussion has been based on an extensive analysis of the available software tools with an additional focus on CSE. Fundamental methods and tools have been introduced followed by a detailed presentation of three different framework concepts. The device simulation framework ViennaMini developed in the course of this work has been introduced, providing a flexible simulation platform to compute the characteristics of a semiconductor device. Also, the developed component execution framework ViennaX has been discussed, allowing to separate functionality into reusable components and execute them - among others - on large-scale computing clusters. The component approach has been extended in the context of an interactive simulation framework to ViennaMOS augmenting the components with GUIs. The feasibility of the individual frameworks has been underlined by application examples.

7.2 Future Extensions

The presented research in this thesis gives rise to a plethora of future research work.

With respect to the device simulation framework ViennaMini further investigations regarding the simulator-material database interface, in particular support for tensor-based material data have to be conducted. Also, adapting the discussed equation objects - holding the PDEs - to incorporate additional more complex models has to be analyzed. These investigations have to especially consider continually growing model numbers. Furthermore, an interface to Python merits special consideration, enabling to perform simulations from within Python scripts, being of interest to advanced users.

Concerning the component execution framework ViennaX, among the identified future investigations further scheduler extensions are of particular interest, especially with respect to shared-memory platforms. The communication layer has to be extended to incorporate an additional communication facility aside from the socket-based communication, based on, for instance, a messaging mechanism. This would allow the modules to tune the execution part according to load-balancing issues. Also, the DTPM scheduler requires further investigations regarding support for loops as well as concerning the overlap of communication with computation based on an hybrid approach.

Regarding the interactive simulation framework ViennaMOS, possible future work should focus, among others, on the extension of the visualization backend. In particular, support for vector field visualization as well as enabling manual color ranges are important, to further improve the visualization capabilities. Furthermore, a decoupling of the visualization backend has to be analyzed, allowing to exchange the currently utilized VTK backend with other libraries, such as the Visualization and Computer Graphics library. Also, capturing C-based output messages from external C-based tools wrapped by ViennaMOS modules has to be investigated, to enable the rerouting of external debug messages into the central framework message window, further improving the overall usability.

7.3 Conclusion

In essence, this thesis is a testament to the importance of CBSE, FLOSS, and LCSD to software engineering in the field of CSE and MNDS in particular. Overall, the three developed frameworks utilize a total of 19 LCSD-based libraries (Table 7.1), whereas several libraries are utilized more than once. This fact further underlines the benefit of decoupling implementations in general, as implementations can be reused in different contexts.

	ViennaFEM	ViennaFVM	ViennaMesh	**Sum**
Boost	•	•	•	3
Netgen			•	1
Tetgen			•	1
Triangle			•	1
ViennaCL	•	•		2
ViennaGrid	•	•	•	3
ViennaMath	•	•		2
Sum				**13**

(a) Libraries

	ViennaMini	ViennaMOS	ViennaX	**Sum**
Boost	•	•	•	3
deal.II			○	1
Qt		•		1
ViennaCL	•	•		2
ViennaFVM	•	•		2
ViennaGrid	•	•		2
ViennaMaterials	•	•		2
ViennaMath	•	•		2
ViennaMesh	•	•		2
ViennaMini		•		1
VTK		•		1
Sum				**19**

(b) Frameworks

Table 7.1: Alphabetical overview of relevant libraries (a) and frameworks (b) part of the Vienna* collection which utilize synergy effects by using LCSD-based libraries; a bullet (•) denotes a dependence whereas a circle (○) relates to an optional dependence. The Boost libraries and the Vienna* libraries are utilized most. Where ViennaFEM, ViennaFVM, and ViennaMesh utilized in total 13 libraries, the discussed framework approaches altogether make use of 19 libraries.

Furthermore, by utilizing and implementing FLOSS as well as by applying CBSE and LCSD techniques for the introduced frameworks, the initially defined research goals (Section 1.4) are tackled. More concretely, not only has the overall goal of developing flexible simulation tools relevant for the field of MNDS been met via the developed frameworks, but also the five established primary requirements of this work, those being reusability, flexibility, usability, maintainability, and expandability, are inherently supported by utilizing FLOSS-based approaches and by applying LCSD as well as CBSE techniques (Table 7.2).

	FLOSS	LCSD	CBSE
Reusability	•	•	•
Flexibility	•	•	•
Usability		•	•
Maintainability		•	•
Expandability	•	•	•

Table 7.2: The key research goals established for this thesis are supported by applying CBSE, FLOSS, and LCSD approaches. A bullet (•) denotes the inherent support of the particular feature by the respective method.

Reusability is provided by implementing and using FLOSS, as software can be accessed by other developers. By applying an LCSD approach libraries can be reused by other libraries and applications. Components of CBSE-based implementations can be reused in different execution contexts.

Flexibility is supported by FLOSS-based tools, as the access to other tools enables to alter an application by developing and using interfaces. Also, LCSD favors flexibility, as the decoupled nature allows to change parts of an implementation with minimized effort. CBSE inherently enables to change the setup of an application via exchanging components.

Usability is improved by LCSD due to the enforced interfaces introduced by the libraries' APIs, easing the utilization of the functionality provided by the individual libraries, which is particularly relevant for software developers. On the contrary, usability is supported by the applied CBSE approach as is being utilized, for instance, in ViennaMOS, where each module potentially provides its own GUI, allowing a highly usable access to the simulation kernel.

Maintainability is favored by LCSD and CBSE approaches, as the decoupled nature of the software reduces the overall code base of individual tools (Table 7.3) and thus by extension increases the maintainability of each tool.

Finally, expandability is supported by FLOSS in the sense that already available software can be accessed and functionality can be added without reimplementing the code base allowing to add the new functionality in the first place. Also, LCSD and CBSE approaches inherently favor expandability due to their utilized component interface, enabling the addition of further components in a straightforward manner.

These five primary aspects, being reusability, flexibility, usability, maintainability, and expandability, are of particular importance to simulation software in a fast pacing research environment such as MNDS. As the presented frameworks utilize these aspects, they allow to provide modern and long-term simulation platforms to end users and advanced users. Simultaneously, developers can continuously advance the simulation tools with minimum effort, ultimately enabling to stay at the forefront of research.

	Lines of Code
ViennaCL	58 019
ViennaGrid	19 578
ViennaMath	7 562
ViennaMesh	7 280
ViennaFEM	4 672
ViennaFVM	2 880
ViennaData	1 779
ViennaMaterials	399

(a) Libraries

	Lines of Code
ViennaMOS	8 717
ViennaX	4 121
ViennaMini	1 760

(b) Frameworks

Table 7.3: The number of source code lines for the Vienna* libraries (a) and frameworks (b) are listed in descending order, as evaluated via the Count Lines of Code [41] tool. The frameworks, allowing to setup the actual applications, offer a small code base, as a significant part of functionality is provided by external libraries.

Bibliography

[1] S. Selberherr, *Analysis and Simulation of Semiconductor Devices*. Springer, 1984, ISBN: 3211818006.

[2] W. Arden, M. Brillouët, P. Cogez, M. Graef, B. Huizing, and R. Mahnkopf, "More-than-Moore," International Technology Roadmap for Semiconductors (ITRS), White Paper, 2010. URL: http://www.itrs.net/Links/2010ITRS/IRC-ITRS-MtM-v2%203.pdf

[3] I. Sommerville, *Software Engineering*, 9th ed. Addison-Wesley, 2010, ISBN: 0137035152.

[4] J. Weinbub, "Why Isn't There More Open Source in Research?" *Blog of the Software Sustainability Institute*, 2013. URL: http://software.ac.uk/blog/2013-01-21-why-isnt-there-more-open-source-research/

[5] *Archimedes*. URL: http://www.gnu.org/software/archimedes/

[6] The Yale Law School Roundtable on Data and Code Sharing, "Reproducible Research," *Computing in Science and Engineering*, vol. 12, no. 5, pp. 8–13, 2010. DOI: 10.1109/MCSE.2010.113

[7] V. Stodden, "The Legal Framework for Reproducible Scientific Research," *Computing in Science and Engineering*, vol. 11, no. 1, pp. 35–40, 2009. DOI: 10.1109/MCSE.2009.19

[8] M. Fayad and D. C. Schmidt, "Object-Oriented Application Frameworks," *Communications of the ACM*, vol. 40, no. 10, pp. 32–38, Oct. 1997. DOI: 10.1145/262793.262798

[9] *ViennaMini*. URL: https://github.com/viennamini/

[10] *ViennaX*. URL: http://viennax.sourceforge.net/

[11] *ViennaMOS*. URL: http://viennamos.sourceforge.net/

[12] *ANSYS*. URL: http://www.ansys.com/

[13] T. Goodale, G. Allen, J. Lanfermann, G.and Massó, T. Radke, E. Seidel, and J. Shalf, "The Cactus Framework and Toolkit: Design and Applications," in *High Performance Computing for Computational Science - VECPAR 2002*, ser. Lecture Notes in Computer Science, 2003, vol. 2565, pp. 197–227. DOI: 10.1007/3-540-36569-9_13

[14] *Cactus*. URL: http://cactuscode.org/

[15] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, "Toward a Common Component Architecture for High-Performance Scientific Computing," in *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 1999, p. 13, ISBN: 0769502873.

[16] B. A. Allan, R. Armstrong, D. E. Bernholdt, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou, "A Component Architecture for High-Performance Scientific Computing," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 163–202, 2006. DOI: 10.1177/1094342006064488

[17] *The Common Component Architecture Forum*. URL: http://www.cca-forum.org/

[18] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl, "The CCA Core Specification in a Distributed Memory SPMD Framework," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 5, pp. 323–345, 2002. DOI: 10.1002/cpe.651

[19] M. Govindaraju, M. Head, and K. Chiu, "XCAT-C++: Design and performance of a distributed cca framework," in *High Performance Computing - HiPC 2005*, ser. Lecture Notes in Computer Science, 2005, vol. 3769, pp. 270–279. DOI: 10.1007/11602569_30

[20] M. J. Lewis, A. J. Ferrari, M. A. Humphrey, J. F. Karpovich, M. M. Morgan, A. Natrajan, A. Nguyen-Tuong, G. S. Wasson, and A. S. Grimshaw, "Support for Extensibility and Site Autonomy in the Legion Grid System Object Model," *Journal of Parallel and Distributed Computing*, vol. 63, no. 5, pp. 525–538, 2003. DOI: 10.1016/S0743-7315(03)00012-1

[21] K. Zhang, K. Damevski, V. Venkatachalapathy, and S. Parker, "SCIRun2: A CCA framework for high performance computing," in *Proceedings of the International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS*, 2004, pp. 72–79. DOI: 10.1109/HIPS.2004.1299192

[22] *COMSOL Multiphysics*. URL: http://www.comsol.com/

[23] A. Lani, T. Quintino, D. Kimpe, H. Deconinck, S. Vandewalle, and S. Poedts, "The COOLFluiD Framework: Design Solutions for High Performance Object Oriented Scientific Computing Software," in *Computational Science - ICCS 2005*, ser. Lecture Notes in Computer Science, 2005, vol. 3514, pp. 279–286. DOI: 10.1007/11428831_35

[24] T. Quintino, "A Component Environment for High-Performance Scientific Computing," Ph.D. thesis, Katholieke Universiteit Leuven, 2008.

[25] *COOLFluiD*. URL: http://coolfluidsrv.vki.ac.be/trac/coolfluid/

[26] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A Generic Distributed DAG Engine for High Performance Computing," *Parallel Computing*, vol. 38, no. 1-2, pp. 37–51, 2012. DOI: 10.1016/j.parco.2011.10.003

[27] *DAGuE*. URL: http://icl.cs.utk.edu/dague/

[28] C. Hill, C. DeLuca, V. Balaji, M. Suarez, and A. d. Silva, "The Architecture of the Earth System Modeling Framework," *Computing in Science and Engineering*, vol. 6, no. 1, pp. 18–28, 2004. DOI: 10.1109/MCISE.2004.1255817

[29] *Earth System Modeling Framework*. URL: http://www.earthsystemmodeling.org/

[30] M. Berzins, "Status of Release of the Uintah Computational Framework," Scientific Computing and Imaging Institute, University of Utah, Tech. Rep. UUSCI-2012-001, 2012.

[31] J. Davison de St.Germain, J. McCorquodale, S. Parker, and C. Johnson, "Uintah: A massively parallel problem solving environment," in *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2000, pp. 33–41. DOI: 10.1109/HPDC.2000.868632

[32] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley Longman Publishing, 2002, ISBN: 0201745720.

[33] J. Sellier, J. Fonseca, and G. Klimeck, "Archimedes, the free Monte Carlo Simulator," in *Proceedings of the International Workshop on Computational Electronics (IWCE)*, 2012, pp. 1–4. DOI: 10.1109/IWCE.2012.6242861

[34] *Genius TCAD Open*. URL: https://github.com/cogenda/Genius-TCAD-Open/

[35] *Gold Standard Simulations*. URL: http://www.goldstandardsimulations.com/

[36] S. Wagner, "Small-Signal Device and Circuit Simulation," Dissertation, Technische Universität Wien, 2005. URL: http://www.iue.tuwien.ac.at/phd/wagner/

[37] R. Klima, "Three-Dimensional Device Simulation with Minimos-NT," Dissertation, Technische Universität Wien, 2002. URL: http://www.iue.tuwien.ac.at/phd/klima/

[38] S. Selberherr, A. Schütz, and H. Pötzl, "MINIMOS - A Two-Dimensional MOS Transistor Analyzer," *IEEE Transactions on Electron Devices*, vol. 27, no. 8, pp. 1540–1550, 1980. DOI: 10.1109/T-ED.1980.20068

[39] *nanoHUB*. URL: http://nanohub.org/

[40] G. Klimeck, M. McLennan, S. Brophy, G. Adams, and M. Lundstrom, "nanoHUB.org: Advancing Education and Research in Nanotechnology," *Computing in Science Engineering*, vol. 10, no. 5, pp. 17–23, 2008. DOI: 10.1109/MCSE.2008.120

[41] *Count Lines of Code (CLOC)*. URL: http://cloc.sourceforge.net/

[42] *NanoTCAD ViDES*. URL: http://vides.nanotcad.com/

[43] *Silvaco*. URL: http://www.silvaco.com/

[44] *Synopsys*. URL: http://www.synopsys.com/

[45] *ViennaSHE*. URL: http://viennashe.sourceforge.net/

[46] K. Rupp, "Deterministic Numerical Solution of the Boltzmann Transport Equation," Dissertation, Technische Universität Wien, 2011. URL: http://www.iue.tuwien.ac.at/phd/rupp/

[47] *Boost*. URL: http://www.boost.org/

[48] *Boost Graph*. URL: http://www.boost.org/libs/graph/

[49] *Boost MPI*. URL: http://www.boost.org/libs/mpi/

[50] *Boost Serialization*. URL: http://www.boost.org/libs/serialization/

[51] *Boost Smart Pointers*. URL: http://www.boost.org/libs/smart_ptr/

[52] *Boost uBLAS*. URL: http://www.boost.org/libs/numeric/ublas/

[53] *Boost Variant*. URL: http://www.boost.org/libs/variant/

[54] S.-W. Cheng, T. K. Dey, and J. R. Shewchuk, *Delaunay Mesh Generation*. Taylor & Francis, 2012, ISBN: 9781584887300.

[55] J. Shewchuk, "What is a Good Linear Element?" in *Proceedings of the International Meshing Roundtable (IMR)*, 2002, pp. 115–126. URL: http://www.imr.sandia.gov/papers/imr11/shewchuk2.pdf

[56] S. J. Owen, "A Survey of Unstructured Mesh Generation Technology," in *Proceedings of the International Meshing Roundtable (IMR)*, 1998, pp. 239–267. URL: http://www.imr.sandia.gov/papers/imr7/owen_meshtech98.ps.gz

[57] *ViennaMesh*. URL: http://viennamesh.sourceforge.net/

[58] *Computational Geometry Algorithms Library (CGAL)*. URL: http://www.cgal.org/

[59] *Gmsh*. URL: http://geuz.org/gmsh/

[60] *Netgen*. URL: http://sourceforge.net/projects/netgen-mesher/

[61] *TetGen*. URL: http://tetgen.berlios.de/

[62] *Triangle*. URL: http://www.cs.cmu.edu/~quake/triangle.html

[63] *ParaView*. URL: http://www.paraview.org/

[64] *Qt*. URL: http://qt-project.org/

[65] *Software at the Institute for Microelectronics*. URL: http://www.iue.tuwien.ac.at/software/

[66] *ViennaProfiler*. URL: http://viennaprofiler.sourceforge.net/

[67] *ViennaWD*. URL: http://viennawd.sourceforge.net/

[68] *ViennaCL*. URL: http://viennacl.sourceforge.net/

[69] *ViennaData*. URL: http://viennadata.sourceforge.net/

[70] *ViennaFEM*. URL: http://viennafem.sourceforge.net/

[71] *ViennaFVM*. URL: http://viennafvm.sourceforge.net/

[72] *ViennaGrid*. URL: http://viennagrid.sourceforge.net/

[73] *ViennaIPD*. URL: http://viennaipd.sourceforge.net/

[74] *ViennaMaterials*. URL: https://github.com/viennamaterials/

[75] *ViennaMath*. URL: http://viennamath.sourceforge.net/

[76] *Visualization Toolkit (VTK)*. URL: http://www.vtk.org/

[77] R. Heinzl, "Concepts for Scientific Computing," Dissertation, Technische Universität Wien, 2007. URL: http://www.iue.tuwien.ac.at/phd/heinzl/

[78] P. Schwaha, "Beyond Atavistic Structures in Scientific Computing," Dissertation, Technische Universität Wien, 2010. URL: http://www.iue.tuwien.ac.at/phd/schwaha/

[79] P. Wegner, "Concepts and Paradigms of Object-Oriented Programming," *SIGPLAN OOPS Messenger*, vol. 1, no. 1, pp. 7–87, 1990. DOI: 10.1145/382192.383004

[80] P. Hudak, "Conception, Evolution, and Application of Functional Programming Languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 359–411, 1989. DOI: 10.1145/72551.72554

[81] J. Backus, "Can Programming Be Liberated from the Von Neumann Style?" *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978. DOI: 10.1145/359576.359579

[82] *Haskell*. URL: http://www.haskell.org/

[83] *Boost Lambda*. URL: http://www.boost.org/libs/lambda/

[84] *Boost Phoenix*. URL: http://www.boost.org/libs/phoenix/

[85] B. Stroustrup, "Evolving a Language in and for the Real World: C++ 1991-2006," in *Proceedings of the ACM SIGPLAN Conference on History of Programming Languages*, 2007, pp. 4–1–4–59. DOI: 10.1145/1238844.1238848

[86] D. Musser and A. A. Stepanov, "Generic Programming," in *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC)*, 1988, pp. 13–25, ISBN: 3540510842.

[87] G. D. Reis and J. Järvi, "What is Generic Programming?" in *Proceedings of the Object-Oriented Programming Systems, Languages, and Applications Conference (OOPSLA), Workshop on Library-Centric Software Design (LCSD)*, 2005. URL: http://lcsd05.cs.tamu.edu/papers/dos_reis_et_al.pdf

[88] T. Veldhuizen, "Expression Templates," *C++ Report*, vol. 7, no. 5, pp. 26–31, 1995.

[89] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming*. Addison-Wesley, 2004, ISBN: 0321227255.

[90] *The Matrix Template Library*. URL: http://www.simunova.com/

[91] W. Bangerth, R. Hartmann, and G. Kanschat, "deal.II - A General-Purpose Object-Oriented Finite Element Library," *ACM Transactions on Mathematical Software*, vol. 33, no. 4, pp. 24:1–24:27, 2007. DOI: 10.1145/1268776.1268779

[92] *deal.II*. URL: http://www.dealii.org/

[93] *The Blitz++ Library*. URL: http://blitz.sourceforge.net/

[94] B. Stroustrup, "Evolving a Language in and for the Real World: C++ 1991-2006," in *Proceedings of the ACM SIGPLAN Conference on History of Programming Languages (HOPL)*, 2007, pp. 4:1–4:59. DOI: 10.1145/1238844.1238848

[95] *TIOBE Programming Community Index*. URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[96] *C++ Standards Committee*. URL: http://open-std.org/JTC1/SC22/WG21/

[97] *GNU Compiler Collection*. URL: http://gcc.gnu.org/

[98] *Clang*. URL: http://clang.llvm.org/

[99] *Intel Developer Zone*. URL: http://software.intel.com/

[100] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010, ISBN: 9781439811924.

[101] *Boost Python*. URL: http://www.boost.org/libs/python/

[102] D. E. Bernholdt, R. C. Armstrong, and B. A. Allan, "Managing Complexity in Modern High End Scientific Computing through Component-Based Software Engineering," in *Proceedings of the Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, 2004. URL: http://www.research.ibm.com/people/r/rajamony/pphec2004-proceedings.pdf

[103] W. Van Roosbroeck, "Theory of the Flow of Electrons and Holes in Germanium and Other Semiconductors," *Bell System Technical Journal*, vol. 29, no. 4, pp. 560–607, 1950. DOI: 10.1002/j.1538-7305.1950.tb03653.x

[104] D. Scharfetter and H. Gummel, "Large-Signal Analysis of a Silicon Read Diode Oscillator," *IEEE Transactions on Electron Devices*, vol. 16, no. 1, pp. 64–77, 1969. DOI: 10.1109/T-ED.1969.16566

[105] T. Grasser, T.-W. Tang, H. Kosina, and S. Selberherr, "A Review of Hydrodynamic and Energy-Transport Models for Semiconductor Device Simulation," *Proceedings of the IEEE*, vol. 91, no. 2, pp. 251–274, 2003. DOI: 10.1109/JPROC.2002.808150

[106] B. M. Klingner and J. R. Shewchuk, "Agressive Tetrahedral Mesh Improvement," in *Proceedings of the International Meshing Roundtable (IMR)*, 2007, pp. 3–23. URL: http://www.imr.sandia.gov/papers/imr16/klinger.pdf

[107] P. Fleischmann and S. Selberherr, "Enhanced Advancing Front Delaunay Meshing in TCAD," in *Proceedings of the International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*, 2002, pp. 99–102. DOI: 10.1109/SISPAD.2002.1034526

[108] V. Palankovski, "Simulation of Heterojunction Bipolar Transistors," Dissertation, Technische Universität Wien, 2000. URL: http://www.iue.tuwien.ac.at/phd/palankovski/

[109] M. Gayer and G. Iannaccone, "A Software Platform for Nanoscale Device Simulation and Visualization," in *Proceedings of the International Conference on Advances in Computational Tools for Engineering Applications (ACTEA)*, 15-17 2009, pp. 432–437. DOI: 10.1109/ACTEA.2009.5227880

[110] A. Logg and G. N. Wells, "DOLFIN: Automated Finite Element Computing," *ACM Transactions on Mathematical Software*, vol. 37, no. 2, pp. 1–28, 2010. DOI: 10.1145/1731022.1731030

[111] B. Stroustrup, "Software Development for Infrastructure," *Computer*, vol. 45, no. 1, pp. 47–58, 2012. DOI: 10.1109/MC.2011.353

[112] *pugixml*. URL: http://pugixml.org/

[113] *Libxml2*. URL: http://www.xmlsoft.org/

[114] *FEniCS*. URL: http://fenicsproject.org/

[115] M. G. Knepley, J. Brown, K. Rupp, and B. F. Smith, "Achieving High Performance with Unified Residual Evaluation," *Computing Research Repository*, 2013. URL: http://arxiv.org/abs/1309.1204

[116] R. N. Hall, "Electron-Hole Recombination in Germanium," *Physical Review*, vol. 87, pp. 387–387, 1952. DOI: 10.1103/PhysRev.87.387

[117] W. Shockley and W. T. Read, "Statistics of the Recombinations of Holes and Electrons," *Physical Review*, vol. 87, pp. 835–842, 1952. DOI: 10.1103/PhysRev.87.835

[118] M. R. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," *Journal of Research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, 1952. DOI: 10.6028/jres.049.044

[119] Y. Saad and M. Schultz, “GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems,” *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, pp. 856–869, 1986. DOI: 10.1137/0907058

[120] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Society for Industrial and Applied Mathematics, 2003, ISBN: 9780898715347.

[121] *Trilinos*. URL: http://trilinos.sandia.gov/

[122] *PETSc*. URL: http://www.mcs.anl.gov/petsc/

[123] *UDUNITS*. URL: http://www.unidata.ucar.edu/software/udunits/

[124] *Lua*. URL: http://www.lua.org/

[125] J. Weinbub, R. Heinzl, P. Schwaha, F. Stimpfl, and S. Selberherr, “A Lightweight Material Library for Scientific Computing in C++,” in *Proceedings of the European Simulation and Modelling Conference (ESM)*, 2010, pp. 454–458, ISBN: 9789077381571.

[126] B. Reed and D. D. E. Long, “Analysis of Caching Algorithms for Distributed File Systems,” *ACM SIGOPS Operating Systems Review*, vol. 30, no. 3, pp. 12–21, 1996. DOI: 10.1145/230908.230913

[127] H. Gummel, “A Self-Consistent Iterative Scheme for One-Dimensional Steady State Transistor Calculations,” *IEEE Transactions on Electron Devices*, vol. 11, no. 10, pp. 455–465, 1964. DOI: 10.1109/T-ED.1964.15364

[128] J. Dongarra and A. van der Steen, “High-Performance Computing Systems,” *Acta Numerica*, vol. 21, pp. 379–474, 2012, ISBN: 9781107026155. URL: http://www.netlib.org/utk/people/PAPERS/acta-num-2012.pdf

[129] S. Borkar and A. A. Chien, “The Future of Microprocessors,” *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011. DOI: 10.1145/1941487.1941507

[130] H. Sutter and J. Larus, “Software and the Concurrency Revolution,” *ACM Queue*, vol. 3, no. 7, pp. 54–62, 2005. DOI: 10.1145/1095408.1095421

[131] C. Lameter, “NUMA (Non-Uniform Memory Access): An Overview,” *ACM Queue*, vol. 11, no. 7, pp. 40:40–40:51, 2013. DOI: 10.1145/2508834.2513149

[132] *OpenMP*. URL: http://openmp.org/

[133] *Intel Cilk Plus*. URL: http://www.cilkplus.org/

[134] IEEE and The Open Group, *IEEE Stdandard 1003.1, 2004 Edition*, 2004. URL: http://pubs.opengroup.org/onlinepubs/000095399/

[135] Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard Version 3.0,” 2012. URL: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[136] CORPORATE Rice University, “High Performance Fortran Language Specification,” *SIGPLAN Fortran Forum*, vol. 12, no. 4, pp. 1–86, 1993. DOI: 10.1145/174223.158909

[137] K. Kennedy, C. Koelbel, and H. Zima, “The Rise and Fall of High Performance Fortran,” in *Proceedings of the ACM SIGPLAN Conference on History of Programming Languages (HOPL)*, 2007, pp. 7:1–7:22. DOI: 10.1145/1238844.1238851

[138] *Partitioned Global Address Space*. URL: http://www.pgas.org/

[139] R. W. Numrich and J. Reid, “Co-Array Fortran for Parallel Programming,” *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998. DOI: 10.1145/289918.289920

[140] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, “Introduction to UPC and Language Specification,” IDA/CCS, Tech. Rep. CCS-TR-99-157, 1999. URL: http://www3.uji.es/~aliaga/UPC/upc_intro.pdf

[141] *Titan*. URL: http://www.olcf.ornl.gov/titan/

[142] *Sequoia*. URL: https://asc.llnl.gov/computing_resources/sequoia/

[143] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, “Dark Silicon and the End of Multicore Scaling,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3, pp. 365–376, 2011. DOI: 10.1145/2024723.2000108

[144] V. Sarkar, W. Harrod, and A. Snavely, “Software Challenges at Extreme Scale,” *SciDAC Review, Special Issue on Advanced Computing: The Roadmap to Exascale*, vol. 16, 2010. URL: http://www.scidacreview.org/1001/html/software.html

[145] *Stampede*. URL: http://www.tacc.utexas.edu/resources/hpc/stampede/

[146] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors*. Morgan Kaufman, 2010, ISBN: 0123814723.

[147] K. Rupp, “CPU, GPU and MIC Hardware Characteristics over Time.” URL: http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

[148] M. Arora, S. Nath, S. Mazumdar, S. Baden, and D. Tullsen, “Redefining the Role of the CPU in the Era of CPU-GPU Integration,” *IEEE Micro*, vol. 32, no. 6, pp. 4–16, 2012. DOI: 10.1109/MM.2012.57

[149] A. Dios, R. Asenjo, A. Navarro, F. Corbera, and E. Zapata, “High-Level Template for the Task-Based Parallel Wavefront Pattern,” in *Proceedings of the International Conference on High Performance Computing (HiPC)*, 2011, pp. 1–10. DOI: 10.1109/HiPC.2011.6152717

[150] J. Weinbub, K. Rupp, and S. Selberherr, “ViennaX: A Parallel Plugin Execution Framework for Scientific Computing,” *Engineering with Computers*, 2013. DOI: 10.1007/s00366-013-0314-1

[151] J. Weinbub, K. Rupp, and S. Selberherr, "Highly Flexible and Reusable Finite Element Simulations with ViennaX," *Journal of Computational and Applied Mathematics*, 2013. DOI: 10.1016/j.cam.2013.12.013

[152] *MPICH*. URL: http://www.mpich.org/

[153] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994, ISBN: 0201633612.

[154] D. Kharrat and S. Quadri, "Self-Registering Plug-ins," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2005, pp. 1324–1327. DOI: 10.1109/CCECE.2005.1557221

[155] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library*. Addison-Wesley Professional, 2001, ISBN: 0201729148.

[156] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2009, ISBN: 0262033844.

[157] Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, 1999. DOI: 10.1145/344588.344618

[158] J. C. Tiernan, "An Efficient Search Algorithm to Find the Elementary Circuits of a Graph," *Communications of the ACM*, vol. 13, no. 12, pp. 722–726, 1970. DOI: 10.1145/362814.362819

[159] K. Agrawal, C. Leiserson, and J. Sukha, "Executing Task Graphs Using Work-Stealing," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470403

[160] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable Work Stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009, pp. 53:1–53:11. DOI: 10.1145/1654059.1654113

[161] *METIS*. URL: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview/

[162] P. Colella, J. Bell, N. Keen, T. Ligocki, M. Lijewski, and B. van Straalen, "Performance and Scaling of Locally-Structured Grid Methods for Partial Differential Equations," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012013, 2007. DOI: 10.1088/1742-6596/78/1/012013

[163] M. Möller and D. Kuzmin, "Adaptive Mesh Refinement for High-Resolution Finite Element Schemes," *International Journal for Numerical Methods in Fluids*, vol. 52, no. 5, pp. 545–569, 2006. DOI: 10.1002/fld.1183

[164] D. W. Kelly, J. P. De S. R. Gago, O. C. Zienkiewicz, and I. Babuska, "A Posteriori Error Analysis and Adaptive Processes in the Finite Element Method," *International Journal for Numerical Methods in Engineering*, vol. 19, no. 11, pp. 1593–1619, 1983. DOI: 10.1002/nme.1620191103

[165] *FastFlow*. URL: http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:about

[166] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "FastFlow: High-Level and Efficient Streaming on Multi-Core," in *Programming Multi-core and Many-core Computing Systems*, ser. Parallel and Distributed Computing. Wiley, 2013, ISBN: 0470936908.

[167] *HECToR*. URL: http://www.hector.ac.uk/

[168] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, "Algorithms and Data Structures for Massively Parallel Generic Adaptive Finite Element Codes," *ACM Transactions on Mathematical Software*, vol. 38, no. 2, pp. 14:1–14:28, 2011. DOI: 10.1145/2049673.2049678

[169] C. Burstedde, L. C. Wilcox, and O. Ghattas, "`p4est`: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees," *SIAM Journal on Scientific Computing*, vol. 33, no. 3, pp. 1103–1133, 2011. DOI: 10.1137/100791634

[170] *p4est*. URL: http://www.p4est.org/

[171] *Hypre*. URL: http://acts.nersc.gov/hypre/

[172] *Visualization and Computer Graphics (VCG) Library*. URL: http://vcg.isti.cnr.it/~cignoni/newvcglib/html/

Curriculum Vitae

Personal Information

Name	Josef Weinbub
Gender	Male
Nationality	Austrian
Place of Birth	Vienna, Austria

Academic Career and Positions

10/2012-11/2012	Visiting researcher Device Modelling Group, University of Glasgow and EPCC, University of Edinburgh, Scotland, UK
since 2/2010	Doctoral student Institute for Microelectronics, TU Wien Supervisor: Prof. Dr. Siegfried Selberherr
since 1/2010	Research assistant Institute for Microelectronics, TU Wien
10/2009	Master's degree in Microelectronics Thesis: *Adaptive Mesh Generation*, TU Wien
2/2009	Bachelor's degree in Electrical Engineering Thesis: *Multi-Paradigm Programming for Scientific Computing with GSSE*, TU Wien

Sideline Positions

since 11/2013	Program committee member of the *SCS/ACM High Performance Computing Symposium (HPC 2014)*, Tampa, FL, USA
11/2013	Invited lecturer at SPOMECH Autumn School *Parallel Solution of Large Engineering Problems* Technical University of Ostrava, Ostrava, Czech Republic
3/2013 - 9/2013	Project head of the Google Summer of Code 2013 organization *Computational Science and Engineering at TU Wien*
since 1/2013	Project head of the Vienna Scientific Cluster project *Modeling Silicon Spintronics, project number: 70388*
3/2012 - 8/2012	Debuty head of the Google Summer of Code 2012 organization *Computational Science and Engineering at TU Wien*
3/2011 - 8/2011	Co-founder and debuty head of the Google Summer of Code 2011 organization *Computational Science and Engineering at TU Wien*

Additional Professional Training

6/2013	Participated in XSEDE/PRACE/RIKEN Summer school *HPC Challenges in Computational Sciences* New York University, New York City, NY, USA
9/2012	Participated in EuroMPI Workshop *Advanced MPI including new MPI-3 Features* Austrian Academy of Sciences, Vienna, Austria
6/2012	Participated in PRACE/Intel Summer school *Code Optimisation for Multi-Core and Intel MIC Architecture* CSCS, Lugano, Switzerland
9/2010	Participated in Summer school *Dresden Microelectronics Academy* TU Dresden, Dresden, Germany

Awards

2012	*HPC-Europa2* fellowship, Edinburgh, Scotland, UK
2012	Best paper award International Conference of Information Engineering (ICIE) IAENG World Congress on Engineering (WCE) London, England, UK

Own Publications

Journal Articles

[1] J. Weinbub, K. Rupp, and S. Selberherr, "Highly Flexible and Reusable Finite Element Simulations with ViennaX," *Journal of Computational and Applied Mathematics*, 2013. DOI: 10.1016/j.cam.2013.12.013

[2] F. Rudolf, J. Weinbub, K. Rupp, and S. Selberherr, "The Meshing Framework ViennaMesh for Finite Element Applications," *Journal of Computational and Applied Mathematics*, 2013, submitted.

[3] J. Weinbub, K. Rupp, and S. Selberherr, "ViennaX: A Parallel Plugin Execution Framework for Scientific Computing," *Engineering with Computers*, 2013. DOI: 10.1007/s00366-013-0314-1

[4] F. Ortmann, S. Roche, J. C. Greer, G. Huhs, T. Shulthess, T. Deutsch, P. Weinberger, M. Payne, J. M. Sellier, J. Sprekels, J. Weinbub, K. Rupp, M. Nedjalkov, D. Vasileska, E. Alfi nito, L. Reggiani, D. Guerra, D. Ferry, M. Saraniti, S. Goodnick, A. Kloes, L. Colombo, K. Lilja, J. Mateos, T. Gonzalez, E. Velazquez, P. Palestri, A. Schenk, and M. Macucci, "Multi-Scale Modelling for Devices and Circuits," *E-Nano Newsletter*, vol. Special Issue April 2012, 2012. URL: http://issuu.com/phantoms_foundation/docs/enn_special_april2012/

Book Contributions

[1] J. Rodriguez, J. Weinbub, D. Pahr, K. Rupp, and S. Selberherr, "Distributed High-Performance Parallel Mesh Generation with ViennaMesh," in *Lecture Notes in Computer Science, Vol. 7782*, P. Manninen and P. Öster, Eds. Springer, 2013, pp. 548–552. DOI: 10.1007/978-3-642-36803-5_44

[2] J. Weinbub, K. Rupp, and S. Selberherr, "A Flexible Dynamic Data Structure for Scientific Computing," in *Lecture Notes in Electrical Engineering, Vol. 229*, G.-C. Yang, S.-L. Ao, and L. Gelman, Eds. Springer, 2013, pp. 565–577. DOI: 10.1007/978-94-007-6190-2_43

[3] J. Weinbub, K. Rupp, and S. Selberherr, "A Lightweight Task Graph Scheduler for Distributed High-Performance Scientific Computing," in *Lecture Notes in Computer Science, Vol. 7782*, P. Manninen and P. Öster, Eds. Springer, 2013, pp. 563–566. DOI: 10.1007/978-3-642-36803-5_47

[4] J. Weinbub, K. Rupp, L. Filipovic, A. Makarov, and S. Selberherr, “Towards a Free Open Source Process and Device Simulation Framework,” in *The 15th International Workshop on Computational Electronics*. IEEE Xplore, 2012, pp. 1–4. DOI: 10.1109/IWCE.2012.6242867

[5] J. Weinbub, K. Rupp, and S. Selberherr, “Towards Distributed Heterogenous High-Performance Computing with ViennaCL,” in *Lecture Notes in Computer Science, Vol. 7116*, I. Lirkov, S. Margenov, and J. Wasniewski, Eds. Springer, 2012, pp. 359–367. DOI: 10.1007/978-3-642-29843-1_41

Conference Contributions

[1] J. Weinbub, K. Rupp, and F. Rudolf, “A Flexible Material Database for Computational Science and Engineering,” in *Abstracts 4th European Seminar on Computing*, 2014, accepted.

[2] K. Rupp, F. Rudolf, J. Weinbub, A. Jüngel, and T. Grasser, “Automatic Finite Volume Discretizations Through Symbolic Computations,” in *Abstracts 4th European Seminar on Computing*, 2014, accepted.

[3] F. Rudolf, Y. Wimmer, J. Weinbub, K. Rupp, and S. Selberherr, “Mesh Generation Using Dynamic Sizing Functions,” in *Abstracts 4th European Seminar on Computing*, 2014, accepted.

[4] V. Sverdlov, H. Mahmoudi, A. Makarov, D. Osintsev, J. Weinbub, T. Windbacher, and S. Selberherr, “Modeling Spin-Based Devices in Silicon,” in *Proceedings of the 16th International Workshop on Computational Electronics (IWCE 2013)*, 2013, pp. 70–71.

[5] J. Weinbub, K. Rupp, and S. Selberherr, “Increasing Flexibility and Reusability of Finite Element Simulations With ViennaX,” in *Abstracts 4th International Congress on Computational Engineering and Sciences*, 2013.

[6] K. Rupp, F. Rudolf, and J. Weinbub, “A Discussion of Selected Vienna-Libraries for Computational Science,” in *Proceedings of C++Now (2013)*, 2013.

[7] J. Weinbub, K. Rupp, and S. Selberherr, “A Flexible Execution Framework for High-Performance TCAD Applications,” in *Proceedings of the 17th International Conference on Simulation of Semiconductor Processes and Devices*, 2012, pp. 400–403.

[8] J. Weinbub, K. Rupp, and S. Selberherr, “A Generic Multi-Dimensional Run-Time Data Structure for High-Performance Scientific Computing,” in *Proceedings of the World Congress on Engineering (WCE)*, 2012, pp. 1076–1081.

[9] J. Weinbub, “A Lightweight Task Graph Scheduler for Distributed High-Performance Scientific Computing,” in *Proceedings of the International Workshop on the State-of-the-Art in Scientific and Parallel Computing*, 2012.

[10] J. Weinbub, “Distributed High-Performance Parallel Mesh Generation with ViennaMesh,” in *Proceedings of the International Workshop on the State-of-the-Art in Scientific and Parallel Computing*, 2012.

[11] J. Weinbub, K. Rupp, L. Filipovic, A. Makarov, and S. Selberherr, “Towards a Free Open Source Process and Device Simulation Framework,” in *Proceedings of the 15th International Workshop on Computational Electronics (IWCE 2012)*, 2012, pp. 141–142.

[12] J. Weinbub, K. Rupp, and S. Selberherr, “Utilizing Modern Programming Techniques and the Boost Libraries for Scientific Software Development,” in *Proceedings of C++Now (2012)*, 2012.

[13] M. Wagner, K. Rupp, and J. Weinbub, “A Comparison of Algebraic Multigrid Preconditioners using Graphics Processing Units and Multi-Core Central Processing Units,” in *Proceedings of the Spring Simulation Multiconference 2012*, 2012.

[14] A. Makarov, V. Sverdlov, D. Osintsev, J. Weinbub, and S. Selberherr, “Modeling of the Advanced Spin Transfer Torque Memory: Macro- and Micromagnetic Simulations,” in *Proceedings of the 25th European Simulation and Modelling Conference*, 2011, pp. 177–181.

[15] J. Weinbub, J. Cervenka, K. Rupp, and S. Selberherr, “High-Quality Mesh Generation Based on Orthogonal Software Modules,” in *Proceedings of the 16th International Conference on Simulation of Semiconductor Processes and Devices*, 2011, pp. 139–142.

[16] J. Weinbub, J. Cervenka, K. Rupp, and S. Selberherr, “A Generic High-Quality Meshing Framework,” in *Proceedings of the 11th US National Congress on Computational Mechanics (USNCCM)*, 2011.

[17] J. Weinbub, K. Rupp, and S. Selberherr, “Distributed Heterogenous High-Performance Computing with ViennaCL,” in *Abstracts Intl. Conf. on Large-Scale Scientific Computations*, 2011, pp. 88–90.

[18] D. Osintsev, V. Sverdlov, Z. Stanojevic, A. Makarov, J. Weinbub, and S. Selberherr, “Properties of Silicon Ballistic Spin Fin-Based Field-Effect Transistors,” in *219th ECS Meeting*, Vol.35, No.5, 2011, pp. 277–282. DOI: 10.1149/1.3570806

[19] A. Makarov, J. Weinbub, V. Sverdlov, and S. Selberherr, “First-Principles Modeling of Bipolar Resistive Switching in Metal-Oxide Based Memory,” in *Proceedings of the European Simulation and Modelling Conference (ESM)*, 2010, pp. 181–186.

[20] J. Weinbub, R. Heinzl, P. Schwaha, F. Stimpfl, and S. Selberherr, “A Lightweight Material Library for Scientific Computing in C++,” in *Proceedings of the European Simulation and Modelling Conference (ESM)*, 2010, pp. 454–458.

[21] K. Rupp, J. Weinbub, and F. Rudolf, “Automatic Performance Optimization in ViennaCL for GPUs,” in *Proceedings of the 9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, 2010. DOI: 10.1145/2039312.2039318

[22] P. Gottschling, R. Heinzl, J. Weinbub, N. Kirchner, M. Sauer, A. Klomfass, C. Steinhardt, and J. Wensch, “Generic C++ Implementation of High-Performance BFS-RBF-based Mesh Motion Schemes,” in *AIP Conference Proceedings*, 1281, 2010, pp. 1631–1634.

[23] G. Mach, R. Heinzl, P. Schwaha, F. Stimpfl, J. Weinbub, and S. Selberherr, “A Modular Tool Chain for High Performance CFD Simulations in Intracranial Aneurysms,” in *AIP Conference Proceedings*, 2010, pp. 1647–1650.

[24] F. Stimpfl, J. Weinbub, R. Heinzl, P. Schwaha, and S. Selberherr, “A Unified Topological Layer for Finite Element Space Discretization,” in *AIP Conference Proceedings*, 2010, pp. 1655–1658.

[25] J. Weinbub, P. Schwaha, R. Heinzl, F. Stimpfl, and S. Selberherr, “A Dispatched Covariant Type System for Numerical Applications in C++,” in *AIP Conference Proceedings*, 2010, pp. 1663–1666.

[26] K. Rupp, F. Rudolf, and J. Weinbub, “ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs,” in *Proceedings of the International Workshop on GPUs and Scientific Applications (GPUScA 2010)*, 2010, pp. 51–56.

[27] J. Weinbub, K. Rupp, and S. Selberherr, “ViennaIPD - An Input Control Language for Scientific Computing,” in *Proceedings of the Industrial Simulation Conference*, 2010, pp. 34–38.

Professional Blog Articles

[1] J. Weinbub, “Why Isn’t There More Open Source in Research?” Blog of the Software Sustainability Institute, University of Edinburgh, Scotland, UK, 2013. URL: http://software.ac.uk/blog/2013-01-21-why-isnt-there-more-open-source-research/.

[2] J. Weinbub, “Research: What a Wonderful Open-Source World,” Blog of the Software Sustainability Institute, University of Edinburgh, Scotland, UK, 2013. URL: http://www.software.ac.uk/blog/2013-07-02-research-what-wonderful-open-source-world/.

Contributed Open Source Projects

[1] *ViennaCL*. URL: http://viennacl.sourceforge.net/

[2] *ViennaFVM*. URL: http://viennafvm.sourceforge.net/

[3] *ViennaGrid*. URL: http://viennagrid.sourceforge.net/

[4] *ViennaIPD*. URL: http://viennaipd.sourceforge.net/

[5] *ViennaMaterials*. URL: https://github.com/viennamaterials/

[6] *ViennaMesh*. URL: http://viennamesh.sourceforge.net/

[7] *ViennaMini*. URL: https://github.com/viennamini/

[8] *ViennaMOS*. URL: http://viennamos.sourceforge.net/

[9] *ViennaX*. URL: http://viennax.sourceforge.net/