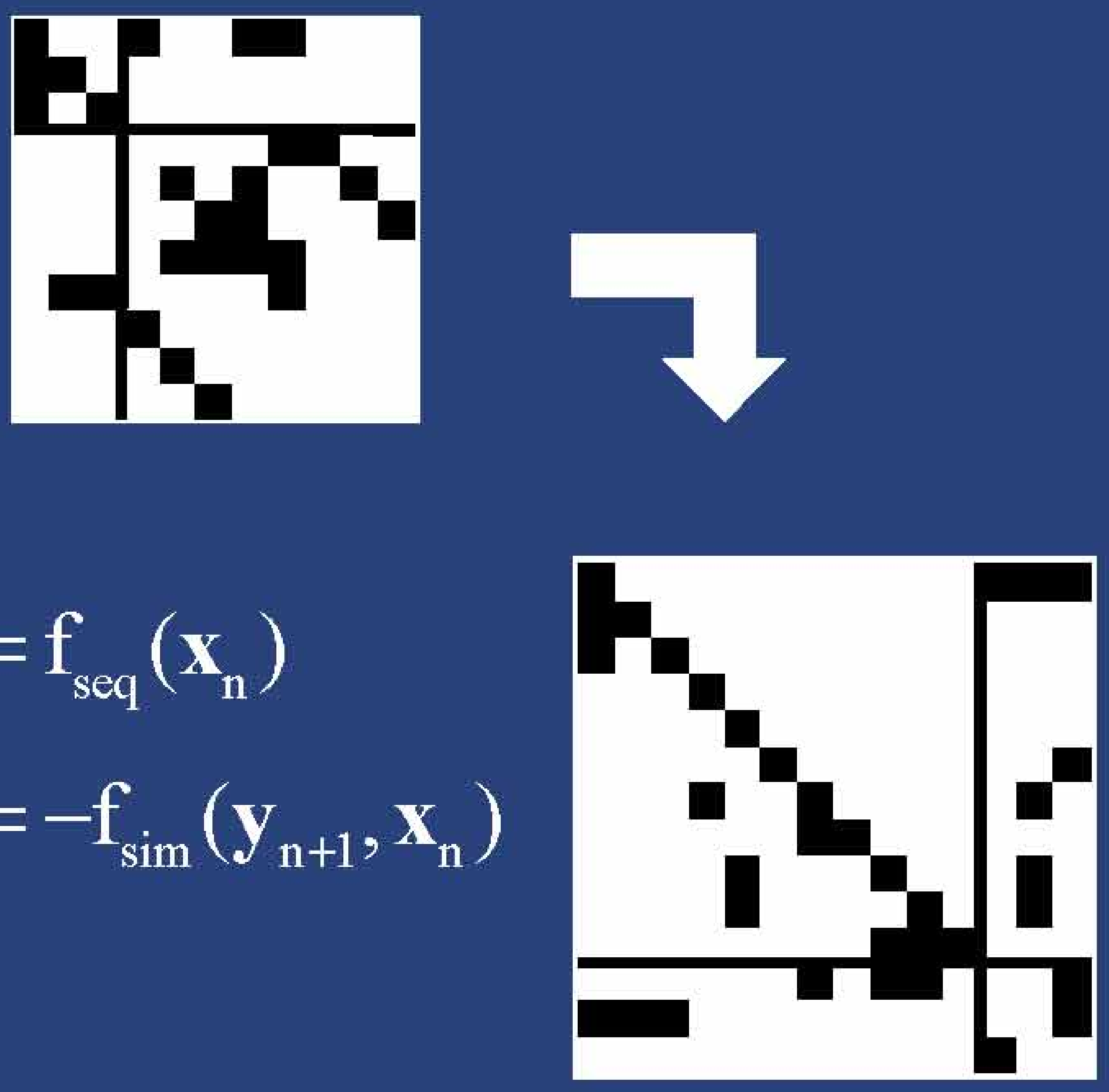
Daniel Platte

Simulation Efficiency of Analog Behavioral Models

Analyses and Improvements



$$\mathbf{y}_{n+1} = f_{seq}(\mathbf{x}_n)$$

$$\left(\mathbf{J}_{22} - \mathbf{J}_{21} \cdot \mathbf{J}_{11}^{-1} \cdot \mathbf{J}_{12}\right) \cdot \Delta\mathbf{x}_n = -f_{sim}(\mathbf{y}_{n+1}, \mathbf{x}_n)$$

Cuvillier Verlag Göttingen

Simulation Efficiency of Analog Behavioral Models – Analyses and Improvements

Von der Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des akademischen Grades

Doktor-Ingenieur
(abgekürzt Dr.-Ing.)

genehmigte

Dissertation

von

Dipl.-Ing. Daniel Platte

geboren am 15. November 1977 in Aachen

2008

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.
1. Aufl. - Göttingen : Cuvillier, 2008
Zugl.: Hannover, Univ., Diss., 2008

978-3-86727-683-2

Referent:	Prof. Dr.-Ing. Erich Barke
Korreferent:	Prof. Dr.-Ing. Ralf Sommer
Tag der Promotion:	14. Juli 2008

Nonnenstieg 8, 37075 Göttingen
Telefon: 0551-54724-0
Telefax: 0551-54724-21
www.cuvillier.de

1. Auflage, 2008
Gedruckt auf säurefreiem Papier

978-3-86727-683-2

Acknowledgements

First of all, many thanks to Prof. Dr. Erich Barke and Prof. Dr. Ralf Sommer for supervising and supporting my research through constant feedback, ideas, and discussions. I would also like to thank the *Titan* team of Qimonda AG, especially Dr. Uwe Feldmann and Dr. Reinhart Schultz, for their great support and patience as well as for many fruitful discussions. Furthermore, I greatly appreciated the discussions and technical support of Dr. Eckhard Hennig, Dr. Manfred Thole, Thomas Halfmann, and my former colleagues at Infineon Technologies AG.

This work would not have been possible without many helping hands. I express my graditude to Christoph Knoth, Shangjing Jing, and Jiong Ou for their dedication and effort during their internships and master theses. Special thanks go to my family, my girlfriend Elke Schmidt, and my friends for their support, motivation, and patience throughout my studies and research.

As I authored this work while being employed by Infineon Technologies AG, I sincerely thank the company for the opportunity to spend three years of challenging research within the company's Ph.D.-program. Furthermore, the valuable cooperation with Qimonda AG and the Fraunhofer ITWM have been of great importance for these developments.

Last but not least, I would like to cordially thank everyone whom I might have forgotten to mention, Prof. Dr. Matthias Heinitz for encouraging me to author this book in English, and everybody who revised this dissertation during the past months.

Neubiberg, July 2008
Daniel Platte

This work was financially supported by the Federal Ministry of Education and
Research of the Federal Republic of Germany (Project No. 01 M 3079).
The author is responsible for the content of the book.

Abstract

In structured top-down design methodologies for the development of complex mixed-signal systems on chip, it is highly desirable to apply automated bottom-up modeling methods. These modeling methods are suited to generate behavioral models of analog blocks at the transistor level with the aim of speeding up simulations at higher abstraction levels. The application of such bottom-up generated models in mixed-mode simulations is essential to verify correct functionality at system or full-chip level. Without the application of behavioral models, verification at the system level is extremely costly in terms of computational effort for most modern designs – if not impossible.

Symbolic analysis offers promising approaches for automated, highly accurate, and flexible bottom-up modeling methods. These methods perform an automated model reduction that is applied to the circuit's network equations and results in simplified but still complex differential algebraic equation (DAE) systems. Based on the DAEs, behavioral models for different modeling languages can be generated. These models can subsequently be used to replace their corresponding transistor-level subsystem in order to enhance the simulation performance. Even though the applied model reduction algorithms are highly efficient, the resulting models' performance is often unsatisfactory – sometimes even slower than at the transistor-level – making the application of such models impossible.

The objective of this work was to analyze and improve the simulation efficiency of such complex analytical models. This has been achieved by an adaptation of the behavioral models and the applied simulation algorithms, thus enhancing the simulation performance without loss of accuracy. The method is based on a highly efficient model compilation as well as on optimization strategies to reformulate the models' DAEs with respect to the applied simulation algorithms. Thus, the simulation performance has been significantly improved by factors of up to two orders of magnitude. An important step towards efficient future use of symbolic methods for bottom-up model generation of analog circuits has been taken.

Keywords: *Analog Behavioral Modeling, Symbolic Analysis, Model Compilation*

Kurzfassung

In einer strukturierten top-down Designmethodik zur Entwicklung komplexer mixed-signal Systeme ist der Einsatz einer automatisierten bottom-up Modellgenerierung von hoher Wichtigkeit. Derartige Modellierungsmethoden sind zur Erzeugung von Verhaltensmodellen analoger Blöcke auf Transistorlevel mit dem Ziel der Simulationsbeschleunigung auf höheren Abstraktionsebenen geeignet. Die Anwendung der erzeugten Verhaltensmodelle in mixed-mode Simulationen ist zur Verifikation auf System- oder Full-Chip-Ebene unerläßlich. Ohne den Einsatz von Verhaltensmodellen ist die Verifikation moderner Mikrochips auf Systemebene extrem rechenzeitaufwändig – wenn nicht sogar unmöglich.

Die symbolische Analyse eröffnet vielversprechende Ansätze zur automatischen, hoch genauen und flexiblen bottom-up Modellierung. Diese Methode basiert auf automatisierten Modellreduktionstechniken, die auf die Netzwerkgleichungen der Schaltung angewendet werden. Die daraus resultierenden vereinfachten aber noch immer äußerst komplexen nichtlinearen Algebrodifferentialgleichungen (ADGL) können zur Erzeugung von Verhaltensmodellen in verschiedenen Modellierungssprachen eingesetzt werden. Die erzeugten Modelle können anschließend zur Ersetzung des entsprechenden Teilsystems auf Transistorebene zur Simulationsbeschleunigung verwendet werden. Trotz sehr hoher Effizienz der Modellreduktionsalgorithmen kann oft nur eine unbefriedigende Simulationsperformanz erzielt werden. Teilweise sind die Modelle sogar weniger performant als die Realisierung auf Transistorebene. Dadurch wird ein effizienter Einsatz der erzeugten Modelle derzeit verhindert.

Ziel dieser Arbeit war die Analyse und Verbesserung der Simulationsperformanz komplexer analytischer Verhaltensmodelle. Dafür wurde eine Anpassung zwischen den Verhaltensmodellen und den angewendeten Simulationsalgorithmen zur Verbesserung der Performanz ohne weiteren Verlust von Genauigkeit durchgeführt. Diese Anpassung basiert auf der Entwicklung eines effizienten Modellcompilers und Algorithmen zur automatischen Umformulierung der ADGL im Hinblick auf die Simulationsalgorithmen. Dadurch konnte die Performanz deutlich um bis zu zwei Größenordnungen verbessert werden. Auf diese Weise ist ein wichtiger Schritt in Hinblick auf die zukünftige Nutzung symbolischer Methoden zur Modellerzeugung für analoge Schaltungen erzielt worden.

Stichworte: *Analoge Verhaltensmodellierung, Symbolische Analyse, Modellkompilierung*

Table of Contents

Abbreviations

AHDL	Analog Hardware Description Language
AMS	Analog/Mixed-Signal
BLT	Block Lower-Triangular (Form/Matrix)
CPU	Central Processing Unit
CSE	Common Subexpression Elimination
DAE	Differential Algebraic Equation
ILP	Instruction Level Parallelism
KCL	Kirchhoff’s Current Law
KVL	Kirchhoff’s Voltage Law
LMS	Linear Multi-Step (Formulae)
LRM	Language Reference Manual
MLNR	Multilevel Newton-Raphson
MNA	Modified Nodal Analysis
PWL	Piece-Wise Linear
RHS	Right-Hand Side (residual)
SPICE	Simulation Program with Integrated Circuit Emphasis
SOC	System on Chip
STA	Sparse Tableau Analysis
ZMS	Z-Element Model Specification

Symbols

General

x	Scalar
$\boldsymbol{x}$	Vector
$\boldsymbol{M}$	Matrix
$\mathbf{1}$	Identity matrix

Simulation Methods

$\Delta\boldsymbol{x}$	Newton correction
λ	Damping factor
$\boldsymbol{f}_{seq}$	Sequential equations
$\boldsymbol{f}_{sim}$	Simultaneous equations
$\boldsymbol{x}$	Simultaneous variables
$\boldsymbol{y}$	Sequential variables
$\boldsymbol{J}$	Jacobian matrix
h	Step size

Network Theory

$\boldsymbol{i}_b$	Branch current
$\boldsymbol{u}_b$	Branch voltage
$\boldsymbol{u}_n$	Node potential
$\boldsymbol{u}_{acr}$	Across quantity
$\boldsymbol{i}_{thr}$	Through quantity
$\boldsymbol{x}_{free}$	Free quantity

Performance Measurements

CPI	Cycles per instruction
Dim	Dimension of a (square) matrix
IC	Instruction count
$N_{EvalCycles}$	Estimated number of evaluation cycles for an equation set
$N_{MemAccess}$	Estimated number of memory accesses for the evaluation of an equation set
$N_{nonzero}$	Number of non-zero entries of a matrix
N_{iter}	Total number of Newton iterations (for transient analysis)
N_{step}	Number of time steps
$N_{iter/step}$	Average number of Newton iterations per time step
N_{SeqEqs}	Number of sequential equations
N_{SimEqs}	Number of simultaneous equations
N_{ports}	Number of model ports
N_{deriv}	Number of differential variables
$N_{condition}$	Number of conditional statements within a model
S	Speed-up
S^{-1}	Slow-down
Spa	Sparsity of a matrix
T_{CPU}	CPU time
T_{*}	a fraction of a CPU time (for profiling)
$T_{*/step}$	a CPU time normalized to N_{step}
$T_{*/iter}$	a CPU time normalized to N_{iter}
T_{load}	CPU time for loading
T_{solve}	CPU time for solving
T_{tran}	CPU time for transient analysis

1 Introduction

In the semiconductor industry one of the key factors for staying competitive is to continuously improve efficiency. In 1965, Gordon E. Moore predicted a rapid exponential growth of the number of transistors that can be integrated on a chip. His prediction became commonly known as “Moore’s Law”. Since then, the semiconductor industry’s growth rates have proven Moore’s early extrapolation. The semiconductor market continuously creates demand for more complex integrated circuits at ever lower cost. This trend pushes the semiconductor companies to improve their efficiency in all fields. Cost efficiency in production is achieved by shrinking transistor sizes and increasing productivity of the semiconductor fabs. Higher system integration leads to complex systems being integrated on a single chip in order to improve the assembly cost for customers. Last but not least, the design efficiency has to keep track with the rapid development of technology. Resources to develop a chip are limited and the project cycle times play a crucial role in timely bringing profitable products to the market. Thus, the lever to become more efficient within a chip’s design phase are design methodology and design automation.

Design efficiency is mostly driven by Electronic Design Automation (EDA). The EDA industry is closely connected to the semiconductor companies as the fulfillment of Moore’s law would not have been possible without a large degree of automation in chip design. The design is based on a design flow, which is composed of a large variety of specialized tools to support the process steps from specification to production. While the design of digital systems is largely automated and dominated by a clearly defined design methodology, analog designs are mostly left to the experience of specialized design engineers. Even though the analog subsystems of a chip are increasingly becoming more important, analog design is insufficiently supported by automated design tools. New design methodologies are necessary to improve efficiency and prevent costly redesigns due to the late detection of errors.

Design and Verification of Microelectronic Circuits

Traditionally, mixed-signal design was performed by bottom-up design. Starting from the design and verification of individual circuit blocks, the obtained components were integrated into the system and verified at transistor level. This design methodology posed several problems such as high simulation effort, disadvantages for architectural changes, risk of communication errors, and late recognition of errors. In order to tackle these problems, top-down design methodologies are increasingly applied [41, 42]. They enhance the efficiency and quality of the design process due to their well-structured refinement from an architecture to a transistor level realization. Each level is thoroughly partitioned, designed, and refined to the next level followed by a verification step. Thereby, the system is step-wise designed from an algorithmic description at the system level down to a transistor level realization of all blocks.

Verification is the process of proving the compliance of a circuit with its specification. For microelectronic systems, verification is of major importance as there is no possibility for prototyping, and redesigning after the production start causes enormous costs. The most common verification method is circuit simulation, which – in a strict sense – does not prove but only validate the circuit. Circuit simulation tools are intended to numerically predict the behavior of a circuit's electrical quantities without having an actual realization of it. They are based on parameterized device models to describe the behavior of the basic electrical components of the circuit. Netlists are used to list the circuit's components and describe their interconnecting network. The simulation of electrical systems can be performed at different abstraction levels:

- **Digital simulation** – time- and value-discrete simulation method based on boolean logic that is capable of simulating large digital circuits with considerable computing resources (millions of transistors within a day).
- **Analog simulation** – continuous value simulation method with adaptive time steps for analog circuits that yields accurate results for currents and voltages, strongly restricted by computing power (thousands of transistors within a day).
- **Mixed-signal simulation** – a combination of the previously mentioned methods that adaptively uses one of the methods for digital or analog partitions of the circuit (hundred-thousands of transistors within a day).
- **Device simulation** – highly accurate field solver to calculate physical behavior of a single or very few semiconductor devices that requires large amounts of computing power (few transistors within weeks, inappropriate for circuit simulation).

The examples for the simulation time give a rough idea of the typical capability of each simulation type. A comparison of the simulation methods shows that the accuracy of the simulation results and the necessary simulation times are conflicting interests. The term accuracy within the simulation-context specifies the degree of conformity of the calculated to the measured values. In order to achieve a high accuracy within simulation, new devices as well as new technologies require the characterization of the devices to achieve suitable parameter sets for the corresponding simulation models. The determination of the device parameters is based on measured characteristics to calibrate the device models' behavior. Due to limited computing resources, performance is often the limiting factor that requires the application of less accurate simulation methods. For circuit simulation, analog simulation is considered to be the most accurate and feasible solution. Device simulators are not suited for circuit simulation due to the required amount of computing power, even though they would be more accurate.

Figure 1.1 visualizes the relationship between accuracy and performance of a simulation. Considering computing resources and efficiencies as constants, an increased accuracy of the models used within the simulation proportionally increases the simulation effort. Thereby, the simulation time increases and performance is affected. The only possibility to enhance

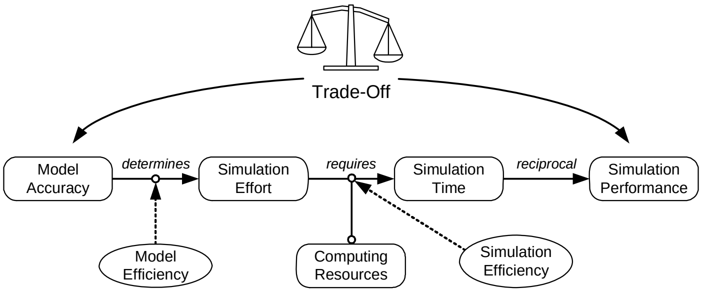


Figure 1.1: Accuracy Performance Trade-Off

performance without loosing accuracy is to increase computing resources (as e.g. applied in parallel computing) or to improve efficiency of the model or the simulator. The scaling of the simulation effort with the model accuracy is influenced by the model efficiency, which is determined by the realization and formulation of the model. The efficiency of the simulator determines the necessary simulation time for a defined simulation effort with a given amount of computing resources. Depending on the application, a suitable trade-off between accuracy and simulation time has to be found. In order to make this trade-off as profitable as possible, the efficiency of the model as well as the simulator must be optimized. Enhancing simulation efficiency is typically impossible for the user of a simulator. Improving the model's efficiency is possible for the creator of the model but requires some internal knowledge of the simulation algorithms and should ideally be done automatically by the simulation environment.

Performance-wise, the verification of large mixed-signal systems on chip level is the most crucial issue in circuit verification. Simulating the entire chip with analog accuracy is almost always not a feasible solution due to the extremely high computational effort. Using a digital simulator is impossible due to the analog subsystems of the chip that cannot be simulated with digital simulation algorithms. In most cases, even the application of a mixed-signal simulator does not reduce the computation time to target (typically over-night simulation).

Behavioral Modeling

The use of behavioral models is a strategy to speed-up simulations. It becomes increasingly important for top-down as well a bottom-up design methodologies. A behavioral model is a functional description of a specific circuit that is suited to predict the relevant behavior of the corresponding circuit with reduced simulation effort. According to Figure 1.1, this reduction of the simulation effort comes along with reduced accuracy of the simulation results. Typically, this is achieved by neglecting physical effects of the circuit implementation that are considered irrelevant for the application of the model. Subsequently, the behavioral model can be used to replace its circuit-counterpart in order to speed-up the verification in larger contexts. The strategy to simulate a system partly represented by its circuit netlist and partly by behavioral models is called multi-level or mixed-mode simulation [11]. By simulating varying combinations of circuits and behavioral models, the functionality of the whole system or of specific components within the system context can be verified.

The most common types of behavioral models are:

- **Electrically equivalent circuits** (macro models) – simplified circuits to model the terminal characteristics of the original circuit [9, 10, 67]. Historically, this is the first approach of behavioral modeling as the models can be simulated with an ordinary circuit simulator.
- **Equation-based models** – behavioral models based on mathematical systems which are typically realized in an Analog Hardware Description Language (AHDL) and require a corresponding simulator interface [4, 33, 48].
- **Look-up tables** – behavioral models based on sampling points stored within data tables. In conjunction with an interpolation method, it is possible to “look up” output characteristics of the model dependent on the input values [78, 86]. This model type is desirable for modeling applications where no equation-based description is available.

For detailed comparisons and discussions on modeling approaches and classifications of behavioral models please refer to [3, 58, 66]. Modeling approaches can be classified into empirical and analytical methods. The former only uses observations, e.g. measurements or simulation data, to reproduce a circuit’s behavior. This has disadvantages as the model does not reflect physical properties of the circuit. Analytical modeling methods are based on physical laws and interrelationships of the modeled circuit. Therefore, a precise analysis and understanding of the circuit is necessary. Analytical modeling methods are superior to empirical methods as they provide insight into the model’s behavior and offer the possibility of adapting the model to circuit changes. Analytical models are equation-based, but not all equation-based models are analytical.

As manual modeling is time-consuming, error-prone, and requires a high level of modeling knowledge, an automated modeling technique is highly desirable. Especially for bottom-up modeling with the intention of deriving a behavioral model from an already implemented circuit block, several automated modeling approaches exist:

- **Characterization** – a library of parameterized model templates allows modeling of specific circuit classes. The parameters for the selected model are determined by characterization of the circuit [19, 37].
- **Neural networks** – behavioral models based on neural networks that are trained with simulation or measurement data [17, 52, 53].
- **Symbolic analysis** – an approach to generate equation-based models using a computer algebra system in combination with network analysis algorithms [3, 5, 27, 31, 65, 87].

This work focuses on automated bottom-up generation of equation-based behavioral models for nonlinear analog circuit blocks through symbolic analysis as introduced in [3]. The approach is based on the automated derivation of symbolic network equations from a circuit within a computer algebra system. The core of a symbolic analysis system is its model reduction algorithm – the process of simplifying equations until a user-specified accuracy-criterion is reached. This method is very useful for bottom-up modeling as it approximates the circuit with its own network equations. The resulting simplified set of equations can be used as core of an equation-based behavioral model. The suggested modeling method has several advantages over other approaches:

- Automated modeling process
- Model accuracy specified in advance
- Very high accuracy attainable with limited modeling effort
- Applicable to all circuit classes (limited to analog block size)
- Resulting models parameterized with dominant circuit parameters
- Insight into the model equations

Chapter 2 will discuss symbolic analysis and its application for behavioral modeling in more detail. An introduction to the relevant simulation algorithms for nonlinear dynamic systems and to behavioral simulation methods will follow in Chapter 3.

Motivation

Even though highly efficient model reduction techniques exist, the generated behavioral models contain equation systems of exceptionally high complexity. Unfortunately, the simulation performance of the generated models is often significantly lower than the performance of the corresponding netlist-based simulation, making their use impossible. Example 1.1 illustrates this problem.

Example 1.1: Performance Problem

In [100], the behavioral model generation for a complementary folded-cascode operational amplifier was published. The operational amplifier consists of 19 MOS-transistors (modeled with BSIM3v3 [89]). This analog block was intended to be modeled through symbolic analysis to achieve a behavioral model with a 10 % error bound of the amplifier’s output voltage. Initially, the equation setup resulted in a complex equation system of 1177 equations – with the majority being highly nonlinear. Through automated model reduction, the equation system was reduced to 29 equations only – still fulfilling the required error margin. The simulation time for the generated behavioral model was enhanced by a factor of 16. Still, the simplified model’s simulation performance was 4 times worse than the performance achieved through the netlist-based simulation of the original circuit.

■

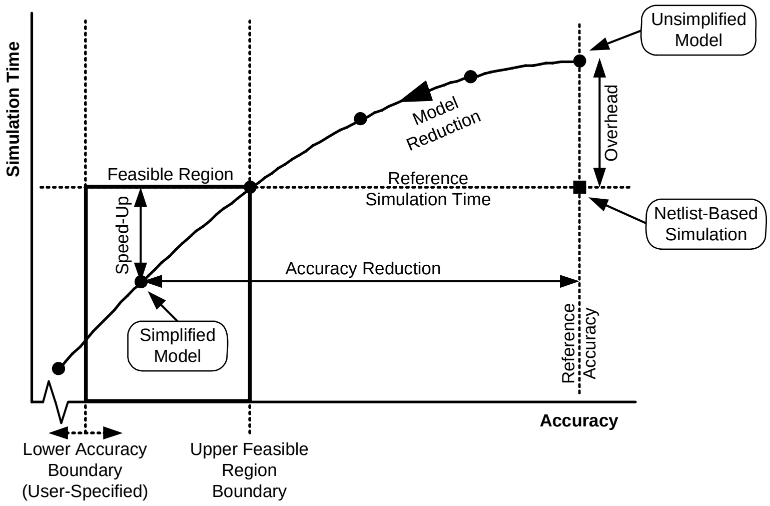


Figure 1.2: Simulation Time vs. Accuracy for Modeling through Symbolic Analysis

Figure 1.2 qualitatively depicts the current situation with respect to the accuracy performance trade-off for this modeling approach. The reference in terms of accuracy and simulation time for all bottom-up modeling methods is the netlist-based simulation (black square in Figure 1.2), as the model is intended to replace (and speed-up) this representation of the circuit block. It is supposed to be the most accurate simulation type for the modeled circuit block. Based on the netlist, symbolic analysis offers the possibility to generate a behavioral model containing equivalent network equations as used simulator-internally for the netlist-based simulation (unsimplified model). This model has the same accuracy as the netlist-based simulation but typically a significantly higher simulation effort, resulting in an overhead in simulation time. Starting from this unsimplified model, a plurality of simplified models along a decreasing trajectory can be achieved through model reduction. The trajectory reflects the trade-off between accuracy and simulation time for different degrees of model reduction. The shape of the curve was chosen exemplarily. In practice, it highly depends on the structure of the model equations and the applied model reduction algorithms. Hence, the trajectory may be of arbitrary shape but should be monotonic decreasing.

The feasible region for a practically useful model is limited by the user-specified minimum accuracy and the requirement to speed-up the simulation (upper feasible region boundary). A simplified model close to the lower accuracy boundary minimizes simulation time. Due to the reduction in accuracy, a certain speed-up compared to the reference simulation time is achieved. In the case of Example 1.1, no feasible compromise between simulation time and accuracy could be found as the upper boundary did not comply with the accuracy requirements. A certain amount of accuracy reduction is necessary to compensate for the overhead in simulation time and to speed-up the model to reach the reference simulation time. Thereby, the efficiency of the modeling approach significantly degrades – this amount of accuracy reduction is “wasted” without achieving a speed-up compared to the reference simulation.

Objectives of this Work

Within Chapter 4, analyses with respect to the behavioral models’ simulation performance will be presented. They show that the overhead is far from being negligible – in most cases the unsimplified model is in the order of one to two magnitudes slower than the netlist-based simulation. The main objectives of the performance analyses are the quantification of the overhead, the investigation for the root causes of the inefficiency, and the determination of influencing factors that account for the overhead.

Certainly, further reducing the model’s accuracy to compensate for the initially bad performance of the unsimplified models is not a satisfying solution. The main objective of this work is to maximize performance through efficiency improvements of both the models and the simulation process. The overhead should be reduced to a minimum in order to make this modeling approach competitive in terms of simulation performance and to effectively use the powerful model reduction algorithms for speeding-up the behavioral models compared to the netlist-based simulation.

Based on the results of the performance analyses, Chapter 5 will present approaches to enhance the behavioral simulation efficiency. Automated optimization methods to increase the model efficiency by reducing the simulation effort at constant accuracy are presented in Chapter 6. Both measures are strongly related to each other as optimal efficiency requires an adaptation between the behavioral model structure as well as the applied simulation algorithms.

2 Behavioral Modeling Through Symbolic Analysis

Traditionally, circuits are analyzed by simulation. While numerical methods are useful for rapidly checking a circuit's functionality and characteristics, they are not well suited to gain insight into and understanding of a circuit. Especially for debugging, optimization, dimensioning, and modeling of analog circuits, a more detailed analysis is of great advantage. The use of symbolic analysis methods [27, 71] provides understanding of interdependencies between variables and parameters of the circuit. These methods are based on symbolic equations that analytically describe the system. A symbolic analysis system is used to set up equations, perform algebraic manipulations, analyze the equations, and apply model reduction based on a computer algebra system. In contrast to numerical methods, all variables and parameters of the equations are contained in symbolic form. By applying numerical values for symbolic parameters and inputs of the system, numerical analyses are easily possible. A weakness of symbolic methods is the extremely high complexity of the equations. In order to cope with the high complexity, symbolic model reduction techniques are applied to achieve a simplified equation system.

Symbolic analysis of linear systems was previously presented in [31]. It is for example used to perform stability analyses, derive symbolic transfer functions, and perform symbolic pole-zero analyses. Nonlinear symbolic methods typically focus on modeling applications for time-domain simulations [5, 26, 58, 82, 83]. This chapter will give an introduction to an analytical modeling method for nonlinear analog circuits based on symbolic analysis. A modeling flow based on the toolbox *Analog Insydes* [2] will be applied. It uses the computer algebra system *Mathematica* [49, 84]. The tool is suited to analyze systems of different physical domains including electrical [6, 36, 81, 100], mechanical [8], thermodynamic [55], as well as feedback-control systems [7].

Figure 2.1 depicts a typical bottom-up modeling flow for analog circuits. Starting from the circuit's netlist, an analytical behavioral model based on the circuit equations is generated. Therefore, symbolic circuit equations are set up using symbolic device models. They are equivalent to the equations used within circuit simulators and are hence considered to be as accurate as the netlist-based simulation of the circuit. By applying nonlinear model reduction algorithms, a simplified equation system of user-defined accuracy is achieved. It can be used as the core of a behavioral model by exporting it to a simulator-compatible model representation. After introducing some basic definitions for analytical modeling, the major processes within the flow will be discussed in more detail and demonstrated through an example application.

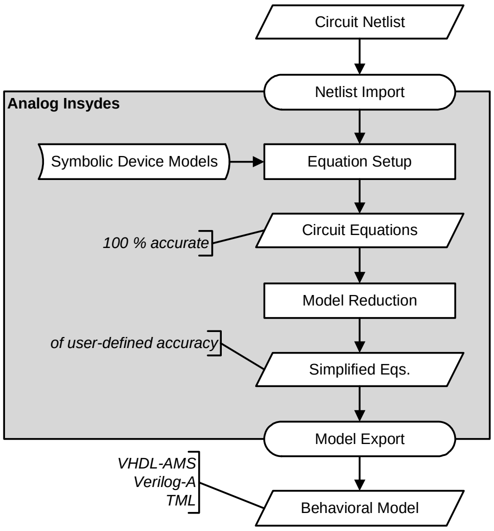


Figure 2.1: Bottom-Up Modeling Process

2.1 Fundamentals of Analytical Modeling

Electrical circuits are built from components and their interconnecting network. A component within the network can be either a so-called primitive or a subcircuit that itself hierarchically consists of components. Primitives are the basic building blocks (e.g. resistors, transistors, voltage sources) that provide a corresponding device model within the simulator. They embody the physical relationship between the primitive's ports by branch-constitutive equations. A behavioral model within an electrical network can be considered as a user-specified primitive. Each component and primitive has a certain number of ports to connect to other components. A connection between two or more ports is called a node or a net. A connection between two nodes is denoted as a branch. Primitives may provide parameters to adapt their behavior (e.g. resistance, geometrical properties). Netlists are used to store the network structure and make it accessible for simulators. A netlist hierarchically contains a set of subcircuit declarations and a set of parameterized instances of components as well as the connectivity information. For further details on the terminology of networks see [88].

Analog circuits are continuous systems since all quantities of the network (voltages, branch currents, node potentials) are considered to be time- and value-continuous, in contrast to digital systems that only use discrete values and time points. For modeling purposes, signal flow systems and conservative systems are distinguished. Within signal flow systems, a port has a certain orientation (input or output) and a specified type (voltage or current port). Thus, a signal-flow port is non-reactive – meaning an input of a component does not influence the electrical quantities of connected nodes and vice versa for outputs. Signal-flow systems are typically used for digital simulation or in mixed-mode simulation at higher abstraction levels (e.g. system level).

Definition 2.1 - Conservative System

In a conservative (electrical) system, each port is associated with a node potential (across quantity) and a port current (through quantity or flow). The node potential is shared with all connected nets. The sum of all port currents into a connected node must sum to zero. The system embodies Kirchhoff's Current Law (KCL) and Kirchhoff's Voltage Law (KVL).

■

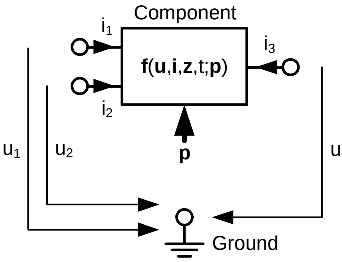


Figure 2.2: Conservative System

A restriction to signal-flow systems does not hold true for most analog circuits. Thus, all components are modeled conservatively (Definition 2.1). Inputs and outputs can not be distinguished in conservative systems. In electrical systems, the across quantities of a component are typically interpreted as the independent input variables whereas through quantities are thought to be the system's outputs. In other words, the port currents of a component are thought to be calculated from the node potentials of its ports. Internal variables are denoted as free quantities and may only relate the across and through quantities of the component to each other.

Apart from this network-related definition of port directions, graphical design tools categorize ports into inputs, outputs, and bidirectional ports. For analog circuitry, this distinction is only of informative value for circuit designers as analog simulators handle all ports as conservative ports. Figure 2.2 shows an exemplary conservative system with three ports bound to the across quantities $u_{1,2,3}$ with corresponding through quantities $i_{1,2,3}$. In combination with the internal variables **z** and the parameters **p**, the equations **f** represent the behavior of the component.

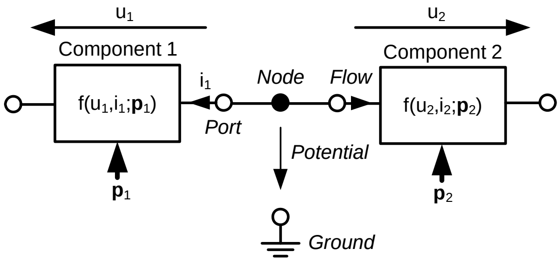


Figure 2.3: Interconnected Components

Figure 2.3 gives an example for two interconnected components. The node potential is shared between both components. The port currents of the inner ports have to be enforced by the network equations to be of equal absolute value and reverse sign (due to KCL).

Definition 2.2 - Differential Algebraic Equations (DAE)

Let $\boldsymbol{f}(\boldsymbol{x}, \dot{\boldsymbol{x}}, t) = 0$ *be a system of differential algebraic equations where*

$\boldsymbol{x} \in \mathbb{R}^n$ *variable vector*

$t \in \mathbb{R}$ *continuous time*

$\boldsymbol{f} : \mathbb{R}^n \times \mathbb{R} \rightarrow \mathbb{R}^n$ *equations.*

The variables $\boldsymbol{x}$ *can be partitioned into*

$\boldsymbol{x}_{diff} \in \mathbb{R}^{n_{diff}}$ *differential variables*

$\boldsymbol{x}_{algebr} \in \mathbb{R}^{n_{algebr}}$ *algebraic variables*

so that

$$\boldsymbol{f}(\boldsymbol{x}, \dot{\boldsymbol{x}}, t) = \boldsymbol{f}(\boldsymbol{x}_{diff}, \dot{\boldsymbol{x}}_{diff}, \boldsymbol{x}_{algebr}, t) = 0$$

with subsystems

$\boldsymbol{f}_{diff}(\boldsymbol{x}_{diff}, \dot{\boldsymbol{x}}_{diff}, \boldsymbol{x}_{algebr}, t) = 0$ *differential equations*

$\boldsymbol{f}_{algebr}(\boldsymbol{x}_{diff}, \boldsymbol{x}_{algebr}, t) = 0$ *algebraic equations*

$$\boldsymbol{f}_{diff} : \mathbb{R}^{n_{diff}} \times \mathbb{R} \rightarrow \mathbb{R}^{n_{diff}}$$

$$\boldsymbol{f}_{algebr} : \mathbb{R}^{n_{algebr}} \times \mathbb{R} \rightarrow \mathbb{R}^{n_{algbr}}$$

■

Continuous systems in the time-domain are described by nonlinear differential algebraic equations (DAE, Definition 2.2). Solving a DAE system in most cases requires the application of numerical methods to simultaneously determine a solution for the system's variables. The next chapter will discuss the most common numerical methods. Most simulation methods require a first-order system. Any DAE system of higher order can be reduced to a first-order system by introducing auxiliary variables and equations (see Appendix B.1.1). Although not mentioned within Definition 2.2 and any upcoming sections, DAE systems may contain symbolic parameters.

Definition 2.3 introduces DAE systems with sequential structure. This concept simplifies modeling and will be of significant advantage for numerical analyses. Sequential equations have to be of an explicit formulation for their according sequential variable and may only depend on simultaneous variables $\boldsymbol{x}$ as well as on previously determined sequential variables $\boldsymbol{y}$. The explicit formulation as well as the sequential ordering enable an efficient processing of this equation type.

Definition 2.3 - DAE System with Sequential Structure

The first-order DAE system $\boldsymbol{f}(\boldsymbol{y}, \dot{\boldsymbol{y}}, \boldsymbol{x}, \dot{\boldsymbol{x}}, t) = 0$ *with*

$\boldsymbol{y} \in \mathbb{R}^n$	*sequential variables*
$\boldsymbol{x} \in \mathbb{R}^m$	*simultaneous variables*
$\boldsymbol{f} = \{\boldsymbol{f}_{seq}, \boldsymbol{f}_{sim}\}$	
$\boldsymbol{f}_{seq} : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \rightarrow \mathbb{R}^n$	*sequential equations*
$\boldsymbol{f}_{sim} : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \rightarrow \mathbb{R}^m$	*simultaneous equations*

is a DAE system with sequential structure, if $\boldsymbol{f}_{seq}$ *is of the form*

$$y_i = \boldsymbol{f}_{seq,\,i}(y_1 \ldots y_{i-1}, \dot{y}_1 \ldots \dot{y}_{i-1}, \boldsymbol{x}, \dot{\boldsymbol{x}}, t) \text{ for } i = 1 \ldots n\,.$$

As the system with sequential structure retains its first order, there exists a decomposition of the system so that

$\boldsymbol{y}_{algebr} = \boldsymbol{f}_{seq,\,algebr}(\boldsymbol{y}_{algebr}, \boldsymbol{x})$	*algebraic sequential subsystem*
$\boldsymbol{y}_{diff} = \boldsymbol{f}_{seq,\,diff}(\boldsymbol{y}_{algebr}, \dot{\boldsymbol{y}}_{algebr}, \boldsymbol{y}_{diff}, \boldsymbol{x}, \dot{\boldsymbol{x}}, t)$	*differential sequential subsystem*
$\boldsymbol{f}_{sim}(\boldsymbol{y}_{algebr}, \dot{\boldsymbol{y}}_{algebr}, \boldsymbol{y}_{diff}, \boldsymbol{x}, \dot{\boldsymbol{x}}, t) = 0$	*simultaneous subsystem*

The decomposed system ensures the absence of any (implicit) second order derivatives and therefore ensures the solvability of the system with Newton's method.

■

Example 2.1: Sequential Equations (Foucault Pendulum)

Take as an example the differential equations describing the motion of a Foucault Pendulum as shown in (2.1).

$$\begin{aligned} \mathtt{x1''[t]} &== -\frac{\mathtt{g\,x1[t]}}{\mathtt{l}} + \mathtt{2\,w\,Cos[lam]\,x2'[t]} + \frac{\mathtt{2\,w\,Sin[lam]\,x1[t]\,x2'[t]}}{\mathtt{l}} \\ \mathtt{x2''[t]} &== -\frac{\mathtt{g\,x2[t]}}{\mathtt{l}} - \mathtt{2\,w\,Cos[lam]\,x1'[t]} + \frac{\mathtt{2\,w\,Sin[lam]\,x2[t]\,x2'[t]}}{\mathtt{l}} \end{aligned} \quad \text{with} \qquad (2.1)$$

`x1[t] , x2[t]`	variables (coordinates of the pendulum bob)
`w , lam , g , l`	parameters (rot. frequency, longitude, gravity, pendulum length)

This DAE system was exemplarily transformed to first-order and rewritten in a sequential form as shown in (2.2a) and (2.2b). Both DAE systems are equivalent.

$$\begin{aligned} \mathtt{y1[t]} &== \mathtt{2\,w\,x4[t]} \\ \mathtt{y2[t]} &== \mathtt{x3'[t]} \\ \mathtt{y3[t]} &== \frac{\mathtt{Sin[lam]\,y1[t]}}{\mathtt{l}} \end{aligned} \qquad \text{(sequential equations)} \qquad (2.2a)$$

$$
\begin{aligned}
&\mathtt{y2[t]} == -\frac{\mathtt{g\,x1[t]}}{\mathtt{l}} + \mathtt{Cos[lam]\;y1[t]} + \mathtt{x1[t]\;y3[t]}\\
&\mathtt{x4'[t]} == -\frac{\mathtt{g\,x2[t]}}{\mathtt{l}} - \mathtt{2\,w\,Cos[lam]\;x3[t]} + \mathtt{x2[t]\;y3[t]}\\
&\mathtt{x3[t]} == \mathtt{x1'[t]}\\
&\mathtt{x4[t]} == \mathtt{x2'[t]}
\end{aligned}
\qquad \text{(simultaneous equations)} \qquad (2.2b)
$$

(2.2a) represents a set of sequential equations. The first and second equations only depend on x4[t] and x3[t], which are simultaneous variables. The third sequential equation only depends on the sequential variable y1[t], which was already defined by the first sequential equation. Thus, the equation set fulfills the requirements for sequential equations.

The equations (2.2b) can not be handled as sequential equations as they either contradict with their dependent variables (Eqs. 1 and 2) or would result in a second order system (Eqs. 3 and 4). Partitioning the sequential equations into algebraic and differential sequential equations yields only one differential sequential equation (second equation of (2.2a)) as this equation depends on a differential variable.

■

2.2 Setup of Symbolic Network Equations

In order to illustrate the modeling process, an example circuit including a diode will be used (see Figure 2.4). VIN is a sinusoidal voltage source whereas VGND and VOUT serve as current probes. The circuit was set up as simple as possible but contains different relevant features like dynamics, nonlinearity, sequential equations, and a case differentiation to illustrate the modeling and simulation process.

The basis for symbolic analysis is the automatic setup of symbolic circuit equations for the design to be analyzed or modeled. Therefore, *Analog Insydes* provides interfaces to import several common netlist formats as well as a direct integration into the Cadence Design Framework II (available from [2]). The network equations are set up based on the circuit de-

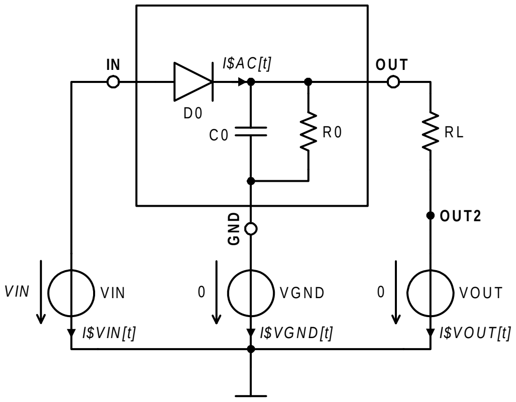


Figure 2.4: Schematic of the Diode Example

sign and a library of symbolic device models. The symbolic device model library contains a variety of commonly used device models in a fully symbolic realization. The contained equations are equivalent to the internal implementation of circuit simulators. In addition to these accurate device models, a selection of simplified device models is available. These models are supplied in different abstraction levels (from ideal to accurate modeling). Hence, the user is able to change the device model for all or only selected devices of the design to achieve a first abstraction.

The basis for all (analytical) simulation methods are systems of equations describing the physical laws and characteristics of the simulated devices. In circuit analysis, the describing equations of a conservative electrical system are based on the element constitutive relations (current-voltage relations of the primitives of the network) as well as on the conservation laws (KCL/KVL) to enforce physically correct behavior within the network. The equations are derived from a graph that represents the structure of the electrical network [14]. It consists of nodes and branches (edges) that represent directed connections between two nodes. Two simplifying assumptions are made to enable network analysis. On the one hand, all primitives are treated as lumped elements neglecting their geometry and any field effects. On the other hand, all connections are assumed to be perfect conductors.

Kirchhoff’s Current Law (2.3) states that the currents of all branches connected to a network node sum to zero. This constitutes the principle of charge conservation. In addition, Kirchhoff’s Voltage Law (2.4) assures the principle of energy conservation. The directed sum of the branch voltages around a closed loop within the network must be zero.

$$\sum_{k=1}^{n} i_{b,k} = 0 \qquad \text{Kirchhoff's Current Law} \tag{2.3}$$

$$\sum_{k=1}^{n} u_{b,k} = 0 \qquad \text{Kirchhoff's Voltage Law} \tag{2.4}$$

There are several graph-theoretical methods to derive network equations from a topology of an electrical network [14, 50, 74]. The most common formulation is the Modified Nodal Analysis (MNA). By applying MNA, a compact system of network equations based on node potentials $\boldsymbol{u}_n$ and branch currents $\boldsymbol{i_b}$ through voltage sources as well as inductors is set up. The nodal equations (2.5) can be achieved by applying KCL to each node of the network. Without the additional equations for voltage-controlled branches (2.6), voltage sources and inductors could not be handled (the pure Nodal Analysis). This analysis method can be efficiently automated and leads to compact equation systems. Therefore, it is used in *SPICE*-like circuit simulators.

$$\boldsymbol{f}_{nodal}(\boldsymbol{u}_n, \boldsymbol{i_b}) = 0 \tag{2.5}$$

$$\boldsymbol{f}_{vbranch}(\boldsymbol{u}_n, \boldsymbol{i_b}) = 0 \tag{2.6}$$

Besides MNA, the Sparse Tableau Analysis (STA) will be used for some performance analyses in this work. The resulting equations in STA are based on branch voltages $\boldsymbol{u_b}$ as well as branch currents $\boldsymbol{i_b}$. The equation system in STA consists of nodal equations (2.7) (relating branch currents to each other via KCL), voltage loop equations (2.8) (enforcing KVL), as well as branch constitutive equations (2.9).

$$\boldsymbol{f}_{nodal}(\boldsymbol{i_b}) = 0 \tag{2.7}$$

$$\boldsymbol{f}_{loop}(\boldsymbol{u}_b) = 0 \tag{2.8}$$

$$\boldsymbol{f}_{bce}(\boldsymbol{i_b}, \boldsymbol{u}_b) = 0 \tag{2.9}$$

Both formulations obtain equivalent systems of equations. MNA obtains more compact network equations than STA:

- MNA: $N_{eqs} = N_{nodes} + N_{vsources} + N_{inductors}$
- STA: $N_{eqs} = 2 \cdot N_{branches}$

In symbolic analyses, the resulting system of DAEs typically consist of the network equations as well as internal equations of the device models. The DAEs can be partitioned into sequential and simultaneous equations and variables (Definition 2.3). The declaration of sequential equations is essential for an efficient numerical solution of the problem as a large percentage of the internal equations is typically given in an explicit formulation. Device models like BSIM3 contain a high percentage of internal sequential equations and result only in a small number of simultaneous equations. Solving these internal equations simultaneously causes significant numerical problems and results in low performance as the dimension of the equation system is quite high. As the complexity of the circuit equations in symbolic formulation is tremendously increasing with the circuit's size and the complexity of the device models, the achieved DAEs are often extremely complex and impossible to set up manually.

Example 2.2: Network Equations for the Diode Example

Setting up the network equations in MNA for the diode example (cf. Figure 2.4) results in a DAE system of 3 sequential equations (2.10a) and 8 simultaneous equations (2.10b). The equations (2.10a) and the 5th equation of (2.10b), which is an internal nodal equation of the diode, were added by the symbolic diode model. Within (2.10b), Equations 1 to 4 are nodal equations, whereas Equations 6 to 8 are voltage equations resulting from the voltage sources.

$$\begin{aligned}
&\mathrm{vd}[t] == -\frac{\mathrm{RS\ I\$AC}[t]}{\mathrm{AREA}} + \mathrm{V\$IN}[t] - \mathrm{V\$OUT}[t] \\
&\mathrm{id}[t] == \mathrm{AREA}\left(-e^{-\frac{0.00333167\,q\,(\mathrm{BV}+\mathrm{vd}[t])}{k}}\,\mathrm{IBV} + \left(-1 + e^{\frac{0.00181069\,q\,\mathrm{vd}[t]}{k}}\right)\mathrm{IS}\right) + \mathrm{GMIN}\ \mathrm{vd}[t] \\
&\mathrm{cj}[t] == \mathrm{AREA\ CJO\ If}\left[\mathrm{vd}[t] < 0.25,\ \frac{1}{(1.-2.\,\mathrm{vd}[t])^{0.333}},\ 2.51926\,(0.3335 + 0.666\,\mathrm{vd}[t])\right]
\end{aligned} \tag{2.10a}$$

$$\begin{array}{l}
\mathtt{I\$VGND[t]} + \dfrac{\mathtt{V\$GND[t]-V\$OUT[t]}}{\mathtt{R0}} + \mathtt{C0}\,(\mathtt{V\$GND'[t]} - \mathtt{V\$OUT'[t]}) == 0 \\
\mathtt{I\$AC[t]} + \mathtt{I\$VIN[t]} == 0 \\
-\mathtt{I\$AC[t]} + \dfrac{\mathtt{-V\$GND[t]+V\$OUT[t]}}{\mathtt{R0}} + \dfrac{\mathtt{V\$OUT[t]-V\$OUT2[t]}}{\mathtt{RL}} + \mathtt{C0}\,(-\mathtt{V\$GND'[t]} + \mathtt{V\$OUT'[t]}) == 0 \\
\mathtt{I\$VOUT[t]} + \dfrac{\mathtt{-V\$OUT[t]+V\$OUT2[t]}}{\mathtt{RL}} == 0 \\
\mathtt{I\$AC[t]} == \mathtt{id[t]} + 1.15 \times 10^{-8}\,\mathtt{id'[t]} + \mathtt{cj[t]}\,\mathtt{vd'[t]} \\
\mathtt{V\$IN[t]} == \mathtt{VIN} \\
\mathtt{V\$OUT2[t]} == 0 \\
\mathtt{V\$GND[t]} == 0
\end{array} \tag{2.10b}$$

The corresponding variables of the system are given in (2.11a) to (2.11c). The vector of simultaneous variables consists of (2.11b) and (2.11c).

$$\mathtt{vd[t],\ id[t],\ cj[t]} \qquad \text{(sequential variables)} \tag{2.11a}$$

$$\mathtt{V\$GND[t],\ V\$IN[t],\ V\$OUT[t],\ V\$OUT2[t]} \qquad \text{(node potentials, simultaneous)} \tag{2.11b}$$

$$\mathtt{I\$AC[t],\ I\$VIN[t],\ I\$VOUT[t],\ I\$VGND[t]} \qquad \text{(currents, simultaneous)} \tag{2.11c}$$

Additionally, the DAE system contains symbolic parameters and constants as listed in (2.12):

$$\mathtt{AREA,\ BV,\ C0,\ CJO,\ GMIN,\ IBV,\ IS,\ k,\ q,\ R0,\ RL,\ RS,\ VIN} \qquad \text{(parameters)} \tag{2.12}$$

The netlist for the *Titan* simulator is given in Appendix A.2. It also contains the numerical values for the parameters of the network elements.

■

2.3 Model Reduction Techniques

Based on these symbolic network equations, different applications like symbolic analysis or model generation can be performed. A major process is the model reduction in order to get a grip on the complexity problem. The intention of model reduction is to generate a compatible DAE system with reduced accuracy and complexity from a given DAE system that does not exceed a user-specified maximal error boundary. The term model reduction (or symbolic approximation) refers to a class of mixed symbolic/numerical methods for the simplification of symbolic equation systems [4, 58, 81, 82]. These methods iteratively perform simplifications within the symbolic DAEs under continuous error control by numerical methods.

In order to perform the numerical error control, a user-specified simulation setup including a testbench is required to determine the model's operating conditions. Furthermore, different error-criteria (e.g. for output voltages of interest) have to be specified. The process starts with a numerical reference analysis of the DAE system that is to be simplified. The calculated data serves as reference solution against which the deviations of the iteratively performed approximation steps are measured. Subsequently, the iterative approximation process is carried out.

The performed simplifications within the DAEs are of the following types:

- **Algebraic simplifications:** Elimination and substitution of variables that do not cause an error, but reduce the DAE systems' complexity [82].
- **Branch simplification:** Branches of piecewise-defined functions, which are not relevant during the simulation, are removed from the equations [65, 82].
- **Switch simplification:** Built-in (binary) parameters within the device models allow to neglect certain physical effects within the device [60].
- **Term substitution:** Terms that are detected to be (nearly) constant during simulation will be substituted by their numerical mean value [3, 82].
- **Term deletion:** Summands within equations with a contribution below the error-margin are hierarchically removed from the system of equations [3, 82].

After each simplification, a numerical analysis is performed to control the resulting error [64]. Simplifications that violate the user-specified error-margins are discarded. This process requires a considerable computational effort for error control. To reduce the number of necessary numerical analyses a ranking process to determine an advantageous order of simplifications and a clustering process to perform multiple simplifications within one iteration are applied in the initialization phase of the algorithm [57, 64, 82]. Thus, a significant improvement in terms of simulation effort could be achieved. Furthermore, the index of the resulting DAEs is monitored during the model reduction process to ensure stability and solvability. Another approach is to monitor convergence and simulation time after each simplification step to ensure that the model's performance is enhanced [58]. A comprehensive overview of the model reduction algorithms of *Analog Insydes* is given in [83].

As the resulting error is controlled during model reduction, the algorithm is one of very few methods that permits satisfying a predefined user-specified accuracy. Furthermore, highly accurate models can be generated as the approximation starts from a 100 % accurate Ansatz, the network equations itself. As the equations' complexity decreases despite the resulting error growing with the degree of model reduction, the problem of finding a suitable trade-off between complexity and accuracy of the model remains. Experiments show that for reasonable error margins the complexity can be reduced by a factor of 10 to 100.

Example 2.3: Simplified Network Equation(s) of the Diode Example

Applying the nonlinear model reduction to the network equations of the diode with the intention to achieve a model only representing the output voltage (10 % absolute error) in terms of the input voltage for the given simulation setup yields

$$\mathrm{AREA}\, e^{\frac{0.00181069\, q\, (\mathrm{VIN}-\mathrm{V\$OUT}[t])}{k}}\, \mathrm{IS} - \frac{\mathrm{V\$OUT}[t]}{\mathrm{R0}} == 0\,. \tag{2.13}$$

The resulting equation (2.13) is quite trivial and is 100 % accurate (rf. to Appendix A.2 for waveforms). As the testbench did not drive the diode into the break-through region and the input source’s frequency was too low to cause dynamic effects, (2.13) only represents the static equilibrium of the diode current and the current through the internal resistor.

■

2.4 Model Generation

The final process step in the bottom-up modeling flow is the generation of the behavioral model for the target simulator based on the simplified DAE system. Behavioral models can be realized in an Analog Hardware Description Language (AHDL) or a hard-coded implementation. The former approach is more flexible as it uses a standardized AHDL. Languages like Accellera Verilog-AMS [88], IEEE VHDL-AMS [90], and Saber MAST are well-known and widely used. The models implemented in an AHDL can be integrated into many different simulation environments and are hence flexible in terms of simulator dependency and reusability. They can be easily distributed and stored in libraries for later reuse. The simulation efficiency of AHDL-based models mainly depends on the processing by the target simulator’s model compiler.

Hard-coded models are implemented using a programming language (e.g. C/C++ or FORTRAN) and are compiled for a specific simulator. They use proprietary interfaces provided by the simulator and are thus not easily portable for other simulators. Typically, a much higher simulation efficiency can be achieved by hard-coded models [58]. The downside of these models is that the effort to manually develop such models is quite high and the implementation requires knowledge about the simulator-internal processing.

Analog Insydes’ model export function supports several output formats allowing to create models for almost every behavioral simulator. The probably most relevant ones are Verilog-A and VHDL-AMS. Although both languages are standardized, behavioral models are not always portable between simulators. The main reasons are unsupported features of modeling languages (requiring a different modeling strategy) or inabilities to cope with certain model contents (e.g. due to bad convergence or low performance). The user’s choice for one of the AHDLs is typically dominated by environmental requirements (simulator, corporate regulations) and personal preferences.

Nevertheless, the combination of an AHDL and a specific simulator has a major influence on robustness and performance (as will be discussed in Chapters 4 and 6) and should therefore be thoroughly taken into consideration. In order to consider simulator-specific properties and to generate a model optimized for the target simulator, the model generation in *Analog Insydes* has been extended by several alternative modeling strategies to optimize the AHDL-generation for a specific simulator (see Appendix B.1.2 for details).

The generated behavioral models consist of the following modeling features:

- **Model declaration / connectivity:** Ports, definition of through/across quantities, model name, parameters.
- **Sequential equations / variables:** Explicit DAEs including branch statements and derivatives.
- **Simultaneous equations / variables:** Implicit DAEs including initial values, tolerances, branch statements, and derivatives.

Compared to the powerful modeling constructs supported by the AHDLs, these requirements are only very basic features of the modeling languages. However, even these features are not sufficiently supported by current versions of some commercial simulators. VHDL-AMS simulators do not support a satisfactory method to model sequential equations whereas Verilog-AMS lacks from a direct and efficient way to model simultaneous equations.

Example 2.4: Model Generation for the Diode Example

Finally, the DAEs of the unsimplified model have been exported to VHDL-AMS and Verilog-A (the behavioral model for the simplified equation would not be very informative). The AHDL codes and the according simulation results are presented in Appendix A.2.

■

3 Algorithms for Circuit and Behavioral Simulation

A basic knowledge about simulation algorithms is essential for creating efficient simulation models. Within this chapter, relevant aspects will be highlighted. The focus is on numerical analysis methods for nonlinear dynamic systems as this is the most typical application for behavioral models.

One of the first analog circuit simulators was *SPICE* (published 1973). Since then, analog simulators have been continuously improved and many commercial simulators have been developed, most of them still being more or less similar to the original *SPICE*. The underlying algorithms have been published in several books [14, 39, 74, 75]. This chapter will provide a brief introduction to nonlinear dynamic simulation algorithms and is mostly based on Vlach [74].

The presented performance analyses within Chapter 4 and all improvements to the behavioral simulation process (Chapter 5) are based on the simulator *Titan*[1] [24]. This *SPICE*-like analog circuit simulator supports behavioral simulation using the *Titan Modeling Language* (*TML*) [16]. The *Titan* simulator was chosen as a platform for this work as it provides deep insight into the internal processing and thereby makes detailed analyses of the simulation performance possible. Furthermore, the inhouse-development of the simulator allowed for realizing prototypical enhancements to the model compilation.

3.1 Solving Linear Equation Systems

Solving linear equation systems numerically is a standard problem for almost all continuous simulation methods. In circuit simulation, all analysis types are based on reducing the problem to a series of linear equation systems that is iteratively solved and refined. Hence, the linear solver is the most basic component within circuit simulators. Consider a linear equation system

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{3.1}$$

where $\boldsymbol{A}$ is an $n \times n$ matrix of constants, $\boldsymbol{b}$ is an n-vector of constants, and $\boldsymbol{x}$ is the vector of unknowns of dimension n. The solution vector $\boldsymbol{x}$ of this equation system could be directly derived through

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b} \,. \tag{3.2}$$

---

[1] *Titan* is an inhouse-simulator of Qimonda AG (formerly Infineon Technologies AG)

However, the computation of (3.2) requires either the matrix-inversion of $\boldsymbol{A}$ or the application of Cramer's rule to solve for $\boldsymbol{x}$, with $\boldsymbol{A}$ typically being a matrix of high dimension. Both methods are expensive in terms of computational effort. Alternatively, Gaussian elimination can be used to transform $\boldsymbol{A}$ into an upper-triangular form which can be subsequently solved for $\boldsymbol{x}$ starting with the last equation (backward substitution). Gaussian elimination requires $n^3/3$ operations, while backward substitution requires $n^2/2$ operations.

SPICE-like simulators are typically using an LU decomposition to solve (3.1). Therefore, the matrix $\boldsymbol{A}$ is decomposed into a lower triangular matrix $\boldsymbol{L}$ and an upper triangular matrix $\boldsymbol{U}$ with ones in the main diagonal so that

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}. \tag{3.3}$$

The decomposition is quite similar to the Gaussian elimination. If $\boldsymbol{A}$ is a non-singular matrix, an LU decomposition exists. It requires approximately $n^3/3$ operations. After decomposing $\boldsymbol{A}$ (e.g. with Crout's method), the decomposed system

$$\boldsymbol{L}\boldsymbol{U}\boldsymbol{x} = \boldsymbol{b} \tag{3.4}$$

is achieved. By introducing an auxiliary (intermediate) solution vector $\boldsymbol{z}$, the problem can be rewritten as shown in (3.5) and solved through backward substitution.

$$\boldsymbol{U}\boldsymbol{x} = \boldsymbol{z} \tag{3.5}$$

By substituting (3.5) into (3.4), the calculation for the forward substitution is derived:

$$\boldsymbol{L}\boldsymbol{z} = \boldsymbol{b} \tag{3.6}$$

Due to the properties of $\boldsymbol{L}$ (lower triangular, non-zero main diagonal elements), (3.6) can be easily solved for $\boldsymbol{z}$ through forward substitution by procedurally calculating

$$z_1 = b_1/l_{1,1} \text{ and} \tag{3.7}$$

$$z_i = \left(b_i - \sum_{j=1}^{i-1} l_{i,j} z_j\right) / l_{i,i} \text{ for } i = 2, 3, \ldots, n. \tag{3.8}$$

By using $\boldsymbol{z}$, the backward substitution (3.5) solves for $\boldsymbol{x}$ through

$$x_n = z_n \text{ and} \tag{3.9}$$

$$x_i = z_i - \sum_{j=i+1}^{n} u_{i,j} x_j \text{ for } i = n-1, n-2, \ldots, 1. \tag{3.10}$$

Both forward and backward substitution are of complexity $O(n^2)$. Through efficient implementation of the LU decomposition and the forward-backward substitution, some vectors may share the same storage (as they are only sequentially needed in intermediate steps). The matrices $\boldsymbol{L}$ and $\boldsymbol{U}$ can be stored within one matrix data structure (as the diagonal of $\boldsymbol{U}$ consists of ones only and hence does not need to be stored).

The forward substitution requires “good” pivot elements l_{ii} for two reasons. First of all, the pivot elements have to be non-zeros to be able to calculate (3.7) and (3.8). Furthermore, the numerical precision of the process depends on the absolute value of the pivot elements that should preferably be large. The former problem leads to a non-solvable system (due to singular matrix $\boldsymbol{A}$) whereas the latter may result in serious accuracy problems (due to the bad conditioning of $\boldsymbol{A}$).

In order to avoid such problems, the linear equation system is pre-ordered by pivoting algorithms. During pivoting, the equation system is reordered by successive permutation of rows and/or columns. Thus, advantageous pivot elements can be achieved and singularity of the matrix is prevented.

Another important aspect of linear solvers is the ability to deal with very large equation systems. Therefore, it is essential to make use of the high sparsity of the matrices. As electrical systems are typically loosely coupled, the matrices are primarily populated by zeros. The sparsity Spa of a matrix provides information on the ratio of non-zero entries compared to the number of total entries of the matrix and is calculated as follows:

$$Spa = \left(1 - \frac{N_{nonzero}}{Dim^2}\right) \tag{3.11}$$

Typical values for the sparsity of electrical systems are between 80 and 99 %. The linear solver is able to make use of the sparsity in order to avoid operations including structural zero entries. Furthermore, specialized sparse data structures are used to reduce the storage for sparse matrices (refer to [68] for details). By using sparse algorithms, the computational complexity has been significantly reduced close to linear complexity (approximately $O(Dim^{1,2\dots1,5})$ depending on the sparsity). However, sparse solvers also lead to additional considerations to improve their efficiency. Without a specialized preordering of the sparse matrix, former zero entries of the matrix are likely to become non-zeros during the LU factorization, so-called fill-ins. In order to keep the sparsity of the matrix as high as possible, preordering strategies with respect to the necessary fill-ins are applied. For this purpose, *Titan* uses the Markowitz preordering that has the property to permute rows and columns pairwise, which preserves the pivot elements and thus does not interfere with previously applied pivoting strategies.

Pivoting strategies are computationally expensive. Especially when solving a series of structurally equal or similar linear equation systems, as it is the case in circuit simulation, reordering might not be performed for each of the linear equation systems. In fact, an initial preordering (static pivoting) based on the structural information of the nonlinear equation system is sufficient to achieve (structurally) non-zero pivot elements and to determine an advantageous ordering for a minimal number of fill-ins. This saves the overhead of reordering the linear system for each iteration. Dynamic pivoting strategies perform the reordering for each of the linear systems (or adaptively whenever necessary). Therefore, they can cope much better with numerically bad pivot elements that can not be considered in static pivoting strategies. Especially in behavioral simulation, this feature becomes important as the equation system is numerically less optimized than in netlist-based simulations.

Table 3.1: Overview of Solvers Available in *Titan*

Solver	***Sparse Solver?***	***Pivoting Strategy***	***Comment***
Titan	yes	static	default solver for netlist-based simulation, very efficient, robustness critical for badly conditioned matrices
LAPACK	no	dynamic, column	default solver for behavioral models, very robust, inefficient for high dimensional systems due to missing sparse algorithms [45]
MUMPS	yes	dynamic, adaptive	**MU**ltifrontal **M**assively **P**arallel sparse direct **S**olver [56], very robust, not as efficient as the *Titan* solver due to pivoting overhead

Titan uses three different linear solvers as shown in Table 3.1. For all netlist-based simulations the highly optimized *Titan* solver is applied. It depends on an initially good pivoting which is typically available in netlist-based simulations. Due to the static pivoting approach, the solver can not cope with numerically bad pivot elements as may result from behavioral models. The *LAPACK* and *MUMPS* solvers are especially relevant for behavioral simulations. Both apply dynamic pivoting strategies, enabling them to cope with numerically complicated behavioral models by dynamically reordering within each iteration. The choice and performance of the solvers will be discussed in more detail in Section 3.5 and Section 4.5.

3.2 DC Analysis

Calculating an operating point or a DC solution for an electrical network is the basis for almost all analysis types. The DC analysis requires the solution of a nonlinear algebraic system of equations to determine the DC node voltages of the network. Especially active devices may contain highly nonlinear branch constitutive equations that could cause numerical problems. Performing the DC analysis is based on Newton's method to determine the roots of a nonlinear equation system $\boldsymbol{f}(\boldsymbol{x}) = 0$ numerically. Newton's method iteratively refines the solution for the nonlinear equation system. This method is widely used and has quadratic convergence, if the initial solution $\boldsymbol{x}_0$ is sufficiently close to the solution.

In order to approximate the nonlinearities within the equation system, a Taylor series expansion at the current solution $\boldsymbol{x}_n$ for the vector of unknowns $\boldsymbol{x}$ is performed:

$$\boldsymbol{f}(\boldsymbol{x}) = 0 \approx \boldsymbol{f}(\boldsymbol{x}_n) + \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x} = \boldsymbol{x}_n} (\boldsymbol{x}_{n+1} - \boldsymbol{x}_n) + \dots \tag{3.12}$$

As $\boldsymbol{x}_n$ is supposed to be close enough to the solution of $\boldsymbol{f}(\boldsymbol{x})$, the Taylor series is truncated to a linear approximation by neglecting terms of an order higher than one. In order to achieve this property, a good initial value $\boldsymbol{x}_0$ is necessary. The iteration sequence for Newton's method is obtained by reformulation of (3.12):

$$\boldsymbol{J}_n \cdot (\boldsymbol{x}_{n+1} - \boldsymbol{x}_n) = \boldsymbol{J}_n \cdot \Delta\boldsymbol{x}_n = -\boldsymbol{f}(\boldsymbol{x}_n) \tag{3.13}$$

where

$$\boldsymbol{J}_n = \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x} = \boldsymbol{x}_n} \quad \text{Jacobian matrix}$$

$$\Delta\boldsymbol{x}_n = \boldsymbol{x}_{n+1} - \boldsymbol{x}_n \quad \text{Newton correction}$$

$$-\boldsymbol{f}(\boldsymbol{x}_n) \quad \text{Residual}$$

As an explicit solution of (3.13) for $\Delta\boldsymbol{x}_n$ would require the calculation of the (expensive) inverse of the Jacobian matrix, the linearized systems are solved by LU decomposition and forward-backward substitution as explained in Section 3.1. Subsequently, a refined solution vector is calculated from

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \lambda \cdot \Delta\boldsymbol{x}_n \tag{3.14}$$

where $\lambda \in \mathbb{R}$ is a damping factor with $0 < \lambda \leq 1$. This (global) damping factor is controlled by the simulator to improve convergence and damp unreasonable large Newton corrections. While $\lambda \neq 1$, Newton's method is continued to avoid false convergence.

The process iteratively converges towards the solution of the equation system and is repeated until the desired accuracy has been achieved. Therefore, convergence criteria are used to compare the current solution to the specified tolerances of the system. The residual vector and the Newton correction are of specific interest as their error norm should be compliant with the tolerances:

$$\|\boldsymbol{f}(\boldsymbol{x}_{n+1})\| < \varepsilon_{abs} + \varepsilon_{rel} \cdot \|\boldsymbol{f}(\boldsymbol{x}_n)\| \tag{3.15}$$

$$\|\Delta\boldsymbol{x}_n\| < \varepsilon_{abs} + \varepsilon_{rel} \cdot \|\Delta\boldsymbol{x}_{n-1}\| \tag{3.16}$$

Due to bad initial values, local minima, oscillations, or highly nonlinear functions, convergence problems might cause Newton's method to fail. In this case, most simulators provide homotopy methods to find a DC solution (also known as continuation methods). Homotopy methods are based on gradually modifying a simplified problem whose solution is known or easy to calculate towards the original problem. Therefore, a homotopy parameter is introduced to scale the selected property of the circuit. Starting with the simplest problem, the DC solution is determined and used as an initial value for the next (more complex) problem. The simplification is completely deactivated and the original system is solved with a good initial value.

There are different types of “simplifications” for homotopy methods (e.g. source stepping, gmin stepping, pseudo-transient analysis), none of them being generally applicable or in all cases successful. Therefore, different homotopy methods are sequentially executed until one of them finishes the DC analysis. All present homotopy methods are based on topological changes within the network. Source stepping for instance introduces a scaling factor for all sources within the network as homotopy parameter. It is initially set to zero. Thus, the DC solution is known and all node voltages are zero. Successively, the scaling factor is swept from zero to one to achieve the solution of the original system. Some DC convergence difficulties resulting from bad initial values can be overcome by applying homotopy methods. For other analysis types, the DC values are typically used as an initial solution. Thus, non-convergence within DC analysis is especially critical as it disables any subsequent analysis that is based on the operating point (e.g. AC, Transient).

3.3 Transient Analysis

Transient analysis calculates the circuit’s response in the time domain over a time interval. The nonlinear dynamic behavior of circuits is therefore described by a nonlinear system of DAEs (see Section 3.4 on how to derive the DAEs):

$$\boldsymbol{f}(\boldsymbol{x}, \dot{\boldsymbol{x}}, t) = 0 \tag{3.17}$$

The analysis starts with an initial DC analysis at time zero to determine a consistent initial value for the system’s unknowns. Starting from the DC solution, the system is discretized in time by numerical integration. Therefore, a variable step size h is used and the resulting nonlinear equations are solved at each time point. Furthermore, time dependent sources are updated within each time point. The most commonly used integration methods are the backward Euler formula (3.18) and the trapezoidal rule (3.19). Implicit linear multistep (LMS) formulae are used for higher order integration methods to achieve “smoother” waveforms and increased stability.

$$\dot{x}_\tau \approx \frac{1}{h_\tau}(x_\tau - x_{\tau-1}) \tag{3.18}$$

$$\dot{x}_\tau \approx -\dot{x}_{\tau-1} + \frac{2}{h_\tau}(x_\tau - x_{\tau-1}) \tag{3.19}$$

The index τ indicates the actual index of the timepoint, the length of the timestep is $h_\tau = t_\tau - t_{\tau-1}$. Both backward Euler formula and trapezoidal rule are implicit and only require values of the previous timestep (first-order). Backward Euler is applied within the first timestep (as $\dot{x}_{\tau-1}$ is typically not known), trapezoidal rule or LMS methods of higher order are applied for subsequent timesteps. Whereas backward Euler is very stable and hence tends to unintentionally damp oscillations, trapezoidal rule is weakly instable and may result in numerical oscillation (“ringing”, propagation of integration errors). A comprehensive discussion regarding stability and properties of integration methods can be found in [39, 74].

Different integration methods can be formulated in a common form

$$\dot{x}_\tau \approx \alpha_\tau x_\tau + r_\tau \tag{3.20}$$

where α_τ is a coefficient (e.g. $1/h_\tau$) and r_τ contains summarized values of previous timepoints. The differential variables are approximated by finite differences by applying numerical integration to (3.17) and the DAE system is transformed into a sequence of nonlinear equation systems:

$$\boldsymbol{f}_\tau(\boldsymbol{x}_\tau) = \boldsymbol{f}(\boldsymbol{x}_\tau, \alpha_\tau \boldsymbol{x}_\tau + \boldsymbol{r}_\tau, t_\tau) \tag{3.21}$$

These systems are solved by iteratively applying Newton's method (subscript τ is omitted for simplicity):

$$\boldsymbol{J}_n \cdot \Delta\boldsymbol{x}_n = (\boldsymbol{J}_{stat,\,n} + \alpha_n \boldsymbol{J}_{dyn,\,n}) \cdot \Delta\boldsymbol{x}_n = -\boldsymbol{f}(\boldsymbol{x}_n) \tag{3.22}$$

where

$$\boldsymbol{J}_{stat,\,n} = \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x} = \boldsymbol{x}_n} \quad \text{(static Jacobian matrix)}$$

$$\boldsymbol{J}_{dyn,\,n} = \left.\frac{\partial \boldsymbol{f}}{\partial \dot{\boldsymbol{x}}}\right|_{\boldsymbol{x} = \boldsymbol{x}_n} \quad \text{(dynamic Jacobian matrix)}$$

As the numerical integration results in a local discretization error, which is dependent on the length of the timestep, the timestep control has an important influence on the accuracy of the solution. Hence, the stepsize is chosen in such a way that the error per step is below a user-specified tolerance. Considerations on the theory of DAE systems and their solvability (related to the index of DAEs) are extensively discussed in [22, 23, 72, 73].

Convergence problems in transient analysis are not as problematic as in DC analysis. This results from starting with a good DC solution and subsequently taking only small timesteps. The solution of a timestep is typically close to the solution of the previous timestep and hence can be determined within few Newton iterations. Furthermore, the numerical integration tends to “smooth” some numerical problems and thereby enhances convergence. Primary sources of dynamic convergence problems are unphysically steep signal edges in time (e.g. independent sources), numerical oscillation by instable integration methods, and discontinuities within models. Furthermore, oscillator circuits might require initial conditions (initial value for charges or fluxes) to “disturb” the equilibrium and thereby initiate the oscillation.

Several specialized simulation methods exist that are used to enhance robustness as well as performance of transient simulations and to cope with special requirements of specific circuit classes. As these methods are not within the scope of this work, only a short overview will be given:

- **Multirate Methods** – are of advantage for systems including widely differing time constants. They are based on multirate integration methods that use local time steps (instead of adaptive global timesteps). Thus, latencies within subsystems can be efficiently utilized [24].

- **FastMOS** – enables the simulation of very large mixed-signal systems with considerably high speed-up compared to conventional transient analysis (factor of 10-1000). The method is based on several simplifications (MOS table models, RC reduction) and uses automatic partitioning methods to deploy multirate methods and hierarchical isomorphism (calculate similar subsystems only once). The speed-up is achieved by solving parts of the circuit with adaptively reduced accuracy. The speed-accuracy trade-off can be controlled by simulator options. Hence, FastMOS engines could replace bottom-up modeling methods as they automatically reduce accuracy for enhanced performance without requiring complex modeling processes. However, these algorithms suffer from missing transparency of the error bound. As it is hardly possible to determine the accuracy of the results, this method suffers reliability. [92] provides a good overview of the related algorithms.
- **Multilevel Newton Methods** – offer the possibility to parallelize the calculation of subsystems on different CPUs. Therefore, loosely coupled systems are identified and solved in parallel [11, 77]. Afterwards, the subsystems' contributions to the higher level system are combined and solved. Finding a suitable partitioning of the circuit's topology for high parallelism is an essential process step [25].
- **RF Algorithms** – are specialized simulation methods to analyze radio-frequency circuits. There are two established methods: harmonic balance and shooting methods. These simulation methods are highly effective for analyzing base-band signals in RF systems. [40] provides a good introduction.
- **Affine Arithmetic Simulation** – provides an innovative semi-symbolic approach to take parameter variations into account. The resulting affine expressions provide a bounded result that still reflects the correlations and causes of uncertainties [28, 29, 32].

3.4 Setup of Network Equations for Circuit Simulation

Within Section 2.2, the setup of network equations with the background of symbolic analysis has been introduced. The derivation of the network equations within a circuit simulator differs from the symbolic setup, as the simulator does not use symbolic equation sets but rather sets up the (numerical) linearized systems directly.

Equation Formulations

As circuit simulation requires a preferably compact equation system, the modified nodal analysis (MNA) is used in most simulators. One of the main advantageous of MNA is that it enables an automated system setup by direct inspection. The linearized equation system is derived incrementally by superposing the contributions ("stamps") of all network elements. Finally, the (numerical) Jacobian matrix and the residual vector resulting from direct inspection of the network elements are transferred to the linear solver. The linear system is updated by the device models within each Newton iteration.

The MNA equation system is shown in Equation (3.23). Its variables are the node potentials $\boldsymbol{u}_n$ and the branch currents $\boldsymbol{i}_b$ through voltage sources as well as inductors. $\boldsymbol{Y}$ denotes the nodal admittance matrix, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{D}$ are coefficient matrices containing coupling coefficients of the network, e.g. resulting from voltage sources, controlled sources, or inductors. $\boldsymbol{i}_{src}$ is the node-current excitation vector and $\boldsymbol{u}_{src}$ represents branch voltage contributions.

$$\begin{bmatrix} \boldsymbol{Y} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{u_n} \\ \boldsymbol{i_b} \end{bmatrix} = \begin{bmatrix} \boldsymbol{i}_{src} \\ \boldsymbol{u}_{src} \end{bmatrix} \tag{3.23}$$

In contrast to MNA, the equation system in STA is represented by (3.24), where $\boldsymbol{A}$ denotes the nodal incidence matrix of the network and $\boldsymbol{B}$ a loop incidence matrix of maximum rank. The branch constitutive equations are represented by $\boldsymbol{Y}$, $\boldsymbol{Z}$, and $\boldsymbol{u}_{src}$.

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{B} \\ \boldsymbol{Y} & \boldsymbol{Z} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{i_b} \\ \boldsymbol{u_b} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{u}_{src} \end{bmatrix} \tag{3.24}$$

As the dimension of the network equations in STA is typically much higher than in MNA, this method is not used in common circuit simulators. It results in very sparsely populated Jacobian matrices.

An extension to the standard MNA used in circuit simulation is the charge-flux oriented equation formulation. It introduces additional variables for the charges of capacitors and the fluxes of inductors to ensure charge conservation. The latter problem results from differential variables in nonlinear equations (e.g. $i = C(u) \cdot \dot{u}$) with the capacitor being modeled nonlinearly. By using this equation formulation, charge conservation is not guaranteed thus possibly resulting in the propagation of integration errors (accumulating voltage offsets due to “lost” charges) [85]. A reformulation of the element relation results in $q = C(u) \cdot u$ and $i = \dot{q}$. This formulation requires an additional variable q, but yields a linear equation for i. Considering the charge-flux based MNA, a nonlinear DAE system of the form

$$\boldsymbol{f}(\dot{\boldsymbol{q}}(\boldsymbol{x}), \boldsymbol{x}, t) = \begin{bmatrix} \boldsymbol{f}_1(\boldsymbol{x}, t) + \dot{\boldsymbol{q}}(\boldsymbol{x}) \\ \boldsymbol{f}_2(\boldsymbol{x}, t) \end{bmatrix} = 0 \tag{3.25}$$

is achieved where $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ describe the static part of the network equations while q represents charges and fluxes [24].

Device Models

Device models are the basis for deriving MNA equations within circuit simulators. Besides the tremendous effort to achieve exact model equations to describe modern semiconductor devices, major effort is spent on optimizing the device models for high simulation performance and numerical robustness.

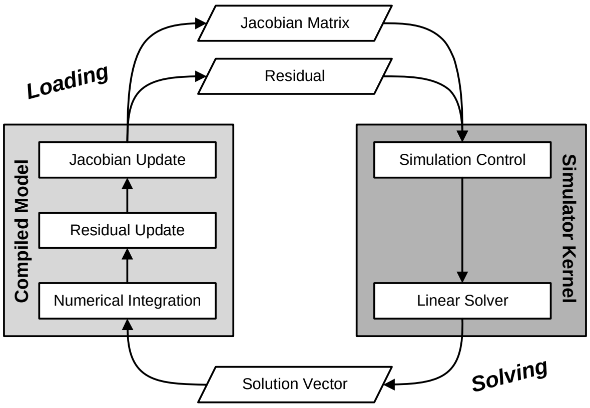


Figure 3.1: Simulation Cycle

Figure 3.1 gives an impression of the simulation cycle and the interaction between simulator kernel and compiled models. The simulator kernel provides a solution vector for the system's unknowns to the model routine. Within the model, the Jacobian matrix and the residual for the next iteration are calculated and returned to the simulator. This stamp of the model is integrated into the linearized system by direct inspection. Afterwards, a new solution vector is determined through the linear solver.

Typical properties of device models are:

- Determination of port currents by means of port voltages
- Admittance formulation of the equations
- Few internal nodes as they increase the dimension of the linear system
- Large amount of internal procedurally calculated equations
- Charge conservation by calculating charges first and then determining the port currents from charge derivatives
- Branches defined to a common reference node (e.g. bulk) leading to $n_{ports} - 1$ branches
- Hard-coded derivatives to determine the Jacobian matrix entries

Common "tricks" to enhance performance and robustness of highly complex device models have been discussed in [15]:

- Limiting functions to avoid numerical problems (e.g. pn-junction limiting)
- Avoiding discontinuities in functions and their first-order derivatives, e.g. by smoothing functions
- Prevention of division by zero through guarding by conditional statements
- Pre-evaluation of common subexpressions to avoid multiple evaluation,
- Approximated derivatives to reduce complexity

Most of these strategies (except the ones related to the Jacobian matrix) may also be applied to behavioral models, always assuming the behavioral simulator supports the required modeling features.

Instead of tediously implementing device models in standard programming languages (typically C/C++), there are approaches to automatically compile device models realized in an AHDL for a simulator specific interface [35, 44]. These device model compilers will be discussed in more detail in Chapter 5. Within [79], an approach to automatically adapt device models to the design was presented (adaptively neglecting certain effects). Furthermore, table models provide an effective measure to avoid the highly complex calculations within device models and thus speed-up simulations.

3.5 Behavioral Model Compilation

Almost every up-to-date circuit simulator provides an interface for at least one of the major AHDLs. Whereas mixed-signal simulators support analog as well as digital features of the modeling languages, common circuit simulators only support the purely analog features. This provides the possibility to extend the simulator by user-specified models, e.g. customized device models or any type of (analog) behavioral models. Most simulation environments generate compiled intermediates from AHDL-based models in order to combine the high performance of hard-coded models with the flexibility achieved by a standardized modeling language. Only very few simulators interpret the AHDL-based models, as the performance is typically lower than for compiled models.

Hence, behavioral simulation requires a model compiler to translate the model into a shared library that can be accessed by the simulator kernel. Therefore, a special interface to the simulator kernel (e.g. CMI [91]) is necessary to enable the communication between model and

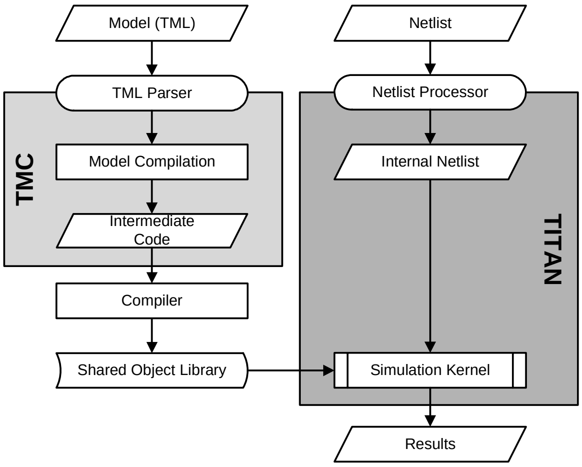


Figure 3.2: Architecture for the Model Compilation (TML)

simulator kernel. In principle, the (compiled) behavioral models are structurally very similar to built-in device models. However, they may differ significantly in the formulation of the modeled DAE system as AHDL-based models are much more general in their formulation.

Titan provides a model interface for the *Titan Modeling Language* (TML). TML is a (mainly analog) subset of VHDL-AMS, some digital features are covered by a digital simulator kernel. As mixed-signal simulation is not within the scope of this work, the focus will be on the purely analog features. Models written in TML are compiled through the *Titan Model Compiler* (tmc) and dynamically linked with the simulator kernel [16, 47, 69]. Figure 3.2 visualizes the architecture of the model compilation process. The communication between the models and the kernel is realized via shared memory.

During the model compilation, the TML code is parsed, stored into an intermediate format, elaborated, and subsequently exported as an intermediate code. The Jacobian matrix is derived by automatic differentiation of the contained simultaneous statements. Finally, the intermediate code is compiled into a shared library by a standard compiler.

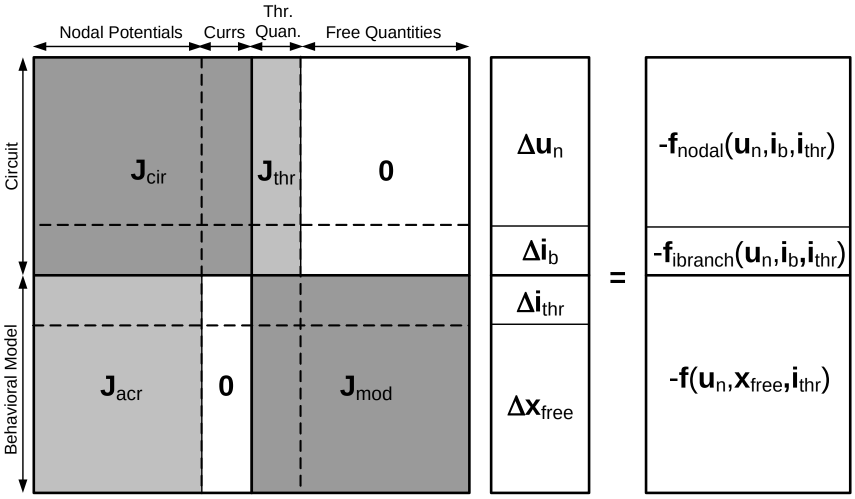


Figure 3.3: Structure of a Linearized System for Behavioral Simulation

Figure 3.3 shows the structure of the linear system for a behavioral simulation in *Titan*. The vector of unknowns consists of the node potentials, the currents through voltage sources and inductors, the through quantities (port currents of the behavioral model) $\boldsymbol{i}_{thr}$, and the free quantities (model internal variables) $\boldsymbol{x}_{free}$. The subsystem for the netlist-based elements of the simulation (upper half) consists of its Jacobian matrix $\boldsymbol{J}_{cir}$, the Newton correction for node potentials and currents, the residual for nodal and voltage equations, and the contributions of the models' port currents to the nodal equations ($\boldsymbol{J}_{thr}$). The model equations result in another subsystem consisting of its internal Jacobian matrix $\boldsymbol{J}_{mod}$, the influences of (selected) node potentials (across quantities or port potentials) $\boldsymbol{J}_{acr}$, and the residual vector for the model equations (simultaneous statements). Both systems are coupled through the across and through quantities of the model.

Titan uses different solvers for the subsystems resulting from the structural description (by direct inspection of the netlist elements) and the subsystem of the models' equations. Please refer to Section 3.1 for a description of the solvers. The subsystem resulting from the circuit's topology is solved by the *Titan* solver. By default, the *LAPACK* solver is used for behavioral models. As the *LAPACK* solver is a dense solver with dynamic pivoting, it is very robust but should preferably be applied only for low dimensional problems. For hand-written models, these preconditions hold as they typically consist only of very few equations that might be numerically critical. Alternatively, the *MUMPS* solver can be applied to combine high robustness (due to adaptive dynamic pivoting) and sparse algorithms. Finally, it is also possible to solve the behavioral models' equations using the standard *Titan* solver. As this solver only performs static pivoting, a specialized preordering and processing of the model equations based on topological information is necessary to ensure solvability, convergence, and accuracy.

Example 3.1: Linear Equation System for the Foucault Pendulum Example

In Example 2.1 (cf. page 13), a sequential DAE system for the Foucault pendulum was introduced. Within this example, the linearized equation system as used in Newton's method for the DAE system is presented. (3.26) shows the static Jacobian matrix of the system, whereas (3.27) represents the dynamic Jacobian matrix of the equation system. The system contains three sequential equations resulting in a lower triangular subblock in the upper left corner of the Jacobian matrix. The last two equations represent dummy equations for modeling the second order derivatives of the system (refer to Appendix B.1.1 for the algorithm).

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & -2\,\mathrm{w} \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -\frac{\mathrm{Sin}[\mathrm{lam}]}{\mathrm{l}} & 0 & 1 & 0 & 0 & 0 & 0 \\ -\mathrm{Cos}[\mathrm{lam}] & 1 & -\mathrm{x1}[\mathrm{t}] & \frac{\mathrm{g}}{\mathrm{l}} - \mathrm{y3}[\mathrm{t}] & 0 & 0 & 0 \\ 0 & 0 & -\mathrm{x2}[\mathrm{t}] & 0 & \frac{\mathrm{g}}{\mathrm{l}} - \mathrm{y3}[\mathrm{t}] & 2\,\mathrm{w}\,\mathrm{Cos}[\mathrm{lam}] & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{(stat. Jacobian)} \tag{3.26}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 \end{pmatrix} \quad \text{(dyn. Jacobian)} \tag{3.27}$$

The corresponding right-hand side of the linearized equation system with a backward Euler integration method applied is shown in (3.28).

$$\begin{pmatrix} 2\,\mathrm{w}\,\mathrm{x4}[\mathrm{t}] - \mathrm{y1}[\mathrm{t}] \\ \frac{-\mathrm{x3}[\mathrm{t}]+\mathrm{x3}[\mathrm{t}+\triangle\mathrm{t}]}{\triangle\mathrm{t}} - \mathrm{y2}[\mathrm{t}] \\ \frac{\mathrm{Sin}[\mathrm{lam}]\,\mathrm{y1}[\mathrm{t}]}{\mathrm{l}} - \mathrm{y3}[\mathrm{t}] \\ -\frac{\mathrm{g}\,\mathrm{x1}[\mathrm{t}]}{\mathrm{l}} + \mathrm{Cos}[\mathrm{lam}]\,\mathrm{y1}[\mathrm{t}] - \mathrm{y2}[\mathrm{t}] + \mathrm{x1}[\mathrm{t}]\,\mathrm{y3}[\mathrm{t}] \\ -\frac{\mathrm{g}\,\mathrm{x2}[\mathrm{t}]}{\mathrm{l}} - 2\,\mathrm{w}\,\mathrm{Cos}[\mathrm{lam}]\,\mathrm{x3}[\mathrm{t}] - \frac{-\mathrm{x4}[\mathrm{t}]+\mathrm{x4}[\mathrm{t}+\triangle\mathrm{t}]}{\triangle\mathrm{t}} + \mathrm{x2}[\mathrm{t}]\,\mathrm{y3}[\mathrm{t}] \\ \frac{-\mathrm{x1}[\mathrm{t}]+\mathrm{x1}[\mathrm{t}+\triangle\mathrm{t}]}{\triangle\mathrm{t}} - \mathrm{x3}[\mathrm{t}] \\ \frac{-\mathrm{x2}[\mathrm{t}]+\mathrm{x2}[\mathrm{t}+\triangle\mathrm{t}]}{\triangle\mathrm{t}} - \mathrm{x4}[\mathrm{t}] \end{pmatrix} \qquad \text{(RHS)} \qquad (3.28)$$

■

Typically, today's highly complex device models still lead to a reasonably high performance even though they contain hundreds of equations. In contrast, quite simple behavioral models that consist of a few equations only might already show astonishingly low performance. The reason for the high performance of device models is their specialized realization to achieve a high degree of adaptation to the simulation algorithms for extremely good performance. Furthermore, their interface to the simulator kernel is highly optimized. However, most of the optimizations applied to device models could also be applied to behavioral models. As the next chapter will indicate, model compilers still lag behind device models in terms of performance.

4 Performance Analyses

This chapter presents a systematic approach for analyzing the simulation performance of analytical behavioral models. Its main intention is to track the root causes for avoidable computational effort and analyze the behavioral simulation efficiency. In general, performance measurements are influenced by various parameters, and interpretations of the results can easily lead to misinterpretation. Thus, benchmarks, operating conditions, and drawn conclusions have to be selected and treated very accurately. Section 4.1 gives an impression what kind of problems have to be taken into account and how statistics for all performance analyses have to be conducted in order to guarantee a solid basis for an improvement of the behavioral simulation performance. The analyses require a common terminology and some basics on computer architecture. Used terms and definitions are consistent with Hennessy/ Patterson [27]. Their book provides comprehensive information on computer architecture.

Measuring Performance

Performance is the reciprocal of the execution time of a program. The most basic definition of execution time is the elapsed time T_{Total}, the latency to conclude a task (also known as wall-clock time, response time). Unfortunately, the elapsed time also accounts for file I/O, multithreading, and operating system activities. Therefore, it is a bad metric for a program's performance as it strongly depends on the usage and load of the computer system. The CPU time T_{CPU} only takes into account the time the processor is executing the program and is therefore much more appropriate for comparing performance.

The CPU time depends on the number of instructions evaluated during a computation (instruction count IC), the cycles per instruction to calculate the instruction (CPI), and the clock cycle time $T_{Clock} = 1/f_{CPU}$. Assuming only instructions of the same type (or an average CPI figure), T_{CPU} is determined by (4.1a), in the more general case of i different instructions by (4.1b).

$$T_{CPU} = IC \cdot CPI \cdot T_{Clock} \tag{4.1a}$$

$$T_{CPU} = \left(\sum_{i=1}^{n} IC_i \cdot CPI_i \right) \cdot T_{Clock} \tag{4.1b}$$

When measuring performance, it is common practice to compare the performance of different programs or computational tasks. The resulting figure of comparing the achieved performance (e.g. of a new program or hardware) to a reference's performance is commonly denominated as speed-up S and is defined as follows:

$$S = \frac{Performance_{new}}{Performance_{ref}} = \frac{T_{CPU,\,ref}}{T_{CPU,\,new}} \tag{4.2}$$

Inversely to speed-up, which is typically a number greater than one, the term slow-down will be used (being the reciprocal of speed-up, S^{-1}). The overall speed-up S_{total} obtained by improving some portion ($Fraction_{enhanced}$) of a computational task by a factor of $S_{enhanced}$ is determined by Amdahl's Law:

$$S_{total} = \frac{T_{CPU,\,ref}}{T_{CPU,\,new}} = \frac{1}{(1 - Fraction_{enhanced}) + \dfrac{Fraction_{enhanced}}{S_{enhanced}}} \tag{4.3}$$

According to Amdahl's Law performance improvements should be accompanied by exact measurements to achieve maximum speed-up with limited effort. Only reasonable fractions of the over-all computational effort are worth the effort to enhance their performance.

Another important characteristic of the performance of a program is the access to data within the memory. During the evaluation of a program data is fetched from the memory, processed by the CPU, and the results are stored back to the memory. In order to avoid high latency at limited cost (fast memory is expensive), up-to-date memory architectures consist of one to three cache levels located between the CPU and the main memory. The cache level closest to the CPU (first level cache or L1 cache) provides the fastest access but has the lowest capacity due to its high cost. The strategies and mechanisms how to load/store data are quite complex and, more important, cannot be influenced by the user. Regardless of that, it is advisable to improve the interaction between CPU and memory by taking care of data locality. Given a memory architecture with only one cache level, the (average) access time to fetch a datum from the cache is determined by

$$T_{Access} = T_{Hit} + MissRate \cdot T_{Penalty} \text{ with } T_{Penalty} \gg T_{Hit} .$$

The miss rate denominates the ratio of unsuccessful cache accesses (data has to be loaded from the underlying memory level) and total accesses. Unfortunately, accesses to the lower levels of the cache architecture take significantly longer ($T_{Penalty}$) than to the first level cache (Pentium 4 – L1: 2 clock cycles, L2: 22 clock cycles, Memory: >> 22). As data is loaded and stored block wise between the hierarchy levels, it is advantageous to access data sequentially instead of randomly. The closer the data of an expression is stored within the memory the less cache misses occur during expression evaluation (data locality). In the worst case (completely random distribution of the data), each cache access would trigger a cache miss (miss rate of 100 %) resulting in an evaluation that is easily more than 10 times slower than in the optimal case.

4.1 Analysis Environment and Objective

In order to equitably analyze simulation performance, a clearly defined reference and simulation environment is necessary. As the simulation performance of bottom-up generated models is to be analyzed, the most appropriate reference is the netlist-based simulation of the original circuit the model was derived from. Previous research has mostly been evaluating the efficiency of the model reduction algorithms by measuring the speed-up of simplified models compared to unsimplified models [3, 58, 82]. The achieved speed-up has typically been between two and three orders of magnitude, but strongly depends on the targeted accuracy of the resulting model. The primary target of this modeling approach is to speed-up the over-all simulation. Hence, the speed-up has to be compared to the performance of the original netlist-based simulation.

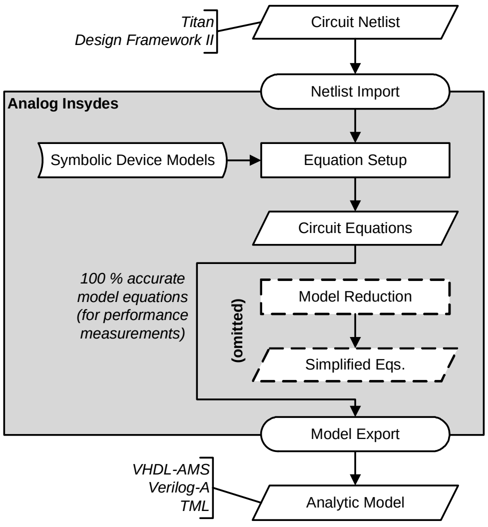


Figure 4.1: Bottom-Up Modeling Process for Performance Analyses

The major objective of this research is to effectively increase the performance of behavioral simulations by improving the simulation efficiency (in contrast to reducing the models' accuracy). An approach based on unsimplified behavioral models was chosen in order to assess the simulator's efficiency in processing behavioral models without distorting the comparison by model reduction. Therefore, the model reduction during modeling was omitted as depicted in Figure 4.1. The goal was to set up and solve equations being mathematically equivalent to the netlist-based simulation in order to have a clearly defined reference for accuracy and performance. In principle, both problems are of equivalent complexity, although the problem's conditioning is completely different. The models were directly generated from the circuit netlist. In order to make this strategy as accurate as the circuit simulation itself, symbolic device models corresponding to the simulator internal device models are used within Analog Insydes.

Thus, the comparison between netlist-based simulation and unsimplified behavioral models results in a slow-down factor representing the overhead spent on processing the same (or at least very similar) problem with the more general approach of behavioral simulation. The *SPICE*-like topological way of solving this problem is very efficient as it is based on highly specialized device models. Therefore, the performance of the circuit simulation represents the optimum for the behavioral simulation performance under the assumption of unsimplified behavioral models.

Beyond doubt, there will always remain a non-negligible slow-down as the behavioral simulation has to solve a much more general problem without utilizing structural knowledge (e.g. from topology). Furthermore, the nonlinear element relations have to be solved in almost the same way as the network equations. A “smart preprocessing” as usually done by device models in order to simplify the solving process is hardly possible (Chapter 6 will address this problem).

Measurements and Criteria

When performing simulations, the resulting waveforms and their analysis in terms of the circuit’s performance matter most. Within the scope of analyzing the simulation performance, the resulting waveforms are only of subordinate interest as they merely verify that the unsimplified behavioral model has the same solution as the netlist-based simulation. In fact, some important characteristics of the simulation as listed in Table 4.1 are of greater interest:

Table 4.1: Characteristics of the Simulation Process

Symbol	***Characteristic***	***Information on***
Dim	Dimension of the linear system	Matrix size
Spa	Sparsity of the Jacobian matrix	Number of non-zero elements
T_{tran}	CPU time for transient analysis	Over-all performance
T_{load}, T_{solve}	Profiling data	Distribution of computational effort
N_{step}	Number of time steps	Differences in time-step control
N_{iter}	Total number of Newton iterations	Convergence

These criteria provide a solid basis as benchmarks to compare behavioral versus netlist-based simulations. Based on those characteristics other figures of merit are derived (see Table 4.2). Above all, the number of Newton iterations is of major importance for the analyses: Only one Newton iteration per time step indicates perfect convergence whereas the maximum number of iterations per step is limited by a simulator option. The normalized measurements for the CPU time provide a basis for comparing different simulators or benchmark circuits without considering their individual behavior in terms of performed time steps and necessary number of iterations. By normalizing the total CPU time over iterations, the pure efficiency in loading and solving the contained equations for one iteration is denoted.

Table 4.2: Derived Characteristics of the Simulation Process

Symbol	*Characteristic*	*Information on*
$N_{iter/step}$	Average iterations per time step	Convergence
$T_{tran/step}$	CPU time per time step	Normalized performance without consideration of time-step control
$T_{tran/iter}$	CPU time per iteration	Normalized performance without consideration of convergence
$N_{nonzero}$	Number of nonzero elements	Storage and processing of the Jacobian matrix

Simulation Setup

All analyses have been performed with nonlinear dynamic models and transient simulations in order to select the most general simulation method. A direct comparison of the results of the netlist-based and behavioral simulations requires equal simulation environments for the experiments. Therefore, the following prerequisites were defined:

- Same simulator, testbench, transient inputs and analysis setup
- Default simulator options (if not mentioned)
- CPU time of each experiment must be measured on the same processor
- CPU time is determined by a mean value of 10 simulations (in order to eliminate load variation)
- Characteristics extracted from simulator protocol files

The default simulator was *Titan*, as it facilitates for in-depth analyses of the simulation performance. Nevertheless, the better part of the drawn conclusions is also valid for other *SPICE*-like behavioral simulators (as Section 4.8 will show). The proposed enhancements introduced in Chapter 5 and Chapter 6 could also be applied to other simulators likewise.

As analog simulators are very sensitive to accuracy options (influencing convergence and number of time steps), particular attention was paid to equitably set those options equivalent to the default accuracy options of *Titan*'s netlist-based simulation. These defaults aimed at providing a high level of accuracy.

Furthermore, the numerical behavior (performance, robustness) of the analyzed simulators differs from platform to platform (Solaris, Linux, 32/64bit), from version to version (not always getting better), from simulator to simulator, and even from model compiler to model compiler (VHDL-AMS, Verilog-A for the same A/MS simulator). Thus, an absolute comparison of performances measured under different operating conditions cannot be drawn. It is impossible to guarantee identical conditions for all possible environments. Hence, the focus was to ensure optimal conditions or perform relative comparisons (e.g. not comparing different simulators absolutely to each other).

An open problem (to be solved by simulator developers) is the time-step control for behavioral models. The AHDLs provide language features to limit the number of time steps for the simulator kernel from within a behavioral model. Unfortunately, the automatic time-step control tends to select time steps that were too large for these types of models. Therefore, the step size has been limited by an adequate maximum step size to achieve comparable results between netlist-based and behavioral simulation.

Applied Model Generation Process

The principles of generating behavioral models with *Analog Insydes* have already been pointed out in Section 2.4. As the following sections and analyses will show, it is essential (in terms of simulation efficiency and numerical robustness) to provide a “good” formulation of the describing model equations within the chosen modeling language. Unfortunately, the optimal formulation also depends on the simulator. This conflicts with the approach of providing simulator-independent models. Hence, the model generation in Analog Insydes was extended by simulator-specific “flavors” of the AHDLs leading to a variety of different modeling alternatives for DAEs (see Appendix B.1 for detailed descriptions of the modeling methods).

The bottom-up modeling flow as shown in Figure 4.1 was automated within a new *Analog Insydes* function taking as input the circuit netlist and various modeling options (`WritePincompatibleModel`, see Appendix B.1 for details). The modeling process generates a fully pin-compatible and 100 % accurate behavioral model (no model reduction performed) for a selected subcircuit of the circuit netlist. Like in *SPICE* device models, only a small portion of a device’s equations has to be treated simultaneously, the major portion of the equations is provided in a sequential structure. In order to use these sequential equations, additional information is returned by symbolic device models for handling sequential equations separately (reducing the number of simultaneous equations, see Section 4.7).

As far as supported by the modeling language and the simulator, initial values were provided to improve DC convergence (derived from an internal operating-point calculation). Specific tolerances for the contained equations and quantities were specified as appropriate in order to ensure accurate processing of the DAEs. Modeling options that (non-intentionally) influence the number of resulting equations by adding auxiliary variables to the system of DAE require particular attention. Within the following analyses, the number of model equations will always be stated by the number of effective equations (resulting from the equation processing). In some cases, the model export to an AHDL requires to introduce auxiliary variables for order reduction or handling of conditional statements as well as sequential equations. For details on methods resulting in such auxiliary variables please refer to Appendix B.

Furthermore, the connectivity between the behavioral model and the circuit (resp. testbench) results in additional variables (port currents). Thus, the dimension of the linearized system as reported by the simulator may differ from the number of effective equations contained within the model. If possible, the number of auxiliary variables is kept to a minimum. The over-all dimension of the models results in the following number of equations:

- TML $Dim_{model} = N_{SimEqs} + N_{ports} + N_{deriv} + [N_{condition}]$
- Verilog-A $Dim_{model} = N_{SimEqs} + 2N_{ports} + N_{deriv}$
- VHDL-AMS $Dim_{model} = N_{SimEqs} + N_{SeqEqs} + 2N_{ports} + N_{deriv}$

Analyzed Circuits (Benchmarks)

In general, any analog circuit that can be modeled with the discussed Analog Insydes based modeling flow is suitable as benchmark for analyzing simulation performance. For practical reasons, the modeling process is limited by the following factors:

- Available symbolic device models within *Analog Insydes* (a matter of spent effort in order to implement further device models; the possibility to automatically set up symbolic device models from imported Verilog-A code has been shown).
- Number of model equations (correlated with the number of transistors and the complexity of the used device models) should not exceed 1500 equations as the complexity of the DAE systems rises significantly. Thereby, various problems might be caused, from computation times for model generation to unacceptably high compilation times for the behavioral models. Using a hierarchical modeling approach provides a generally advisable solution to this limitation.

For the presented analyses, a selection of analog circuit blocks with different characteristics has been chosen. The intention was to select circuits of different sizes and different device models to cover a wide range of future applications. All measurements have been performed with transient analyses. The testbenches and input stimuli have been designed to show nonlinear as well as dynamic characteristics of the circuits and models. For most examples, periodic stimuli sources were chosen to provide a measurable amount of CPU time over several periods. Table 4.3 provides an overview of the analyzed circuits. For all BSIM3-modeled devices, the parameter set of Infineon's 130 nm CMOS technology was used.

A short summary of used circuit designs and their testbenches:

- *cfcamp* – a complementary folded-cascode operational amplifier in degenerative feedback (unity gain buffer), pulse input voltage (500 kHz frequency, 0.45 V amplitude, 0.75 V DC), 19 BSIM3 instances
- *emitter* – a common emitter configuration with periodic piece-wise linear input
- *multiplier* – a small circuit “multiplying” two input voltages, 8 Gummel-Poon instances
- *nand2* – a NAND gate, input stimuli trigger all states (1.5 V supply voltage, 1.5 GV/s slew rate), 4 BSIM3 instances
- *opamp741* – the uA741 operational amplifier in degenerative feedback (unity gain buffer), pulse wave input voltage (220 kHz frequency, 200 mV amplitude, 2 MV/s slew rate), 26 Gummel-Poon instances
- *sqrt* – a circuit implementing a sqrt-function from current input to current output, periodic piece-wise linear source

Table 4.3: Overview of the Analyzed Circuits

Example	***Transistors***	***Device Model***	***Ports***	***Equations***	***Parameters***
cfcamp	19	BSIM3v3	6	1328	1621
cfcamp_mos1	19	MOS Level 1	6	134	155
emitter	1	Gummel-Poon	4	5	9
multiplier	6	Gummel-Poon	8	94	104
nand2	4	BSIM3v3	5	247	371
opamp741	26	Gummel-Poon	5	357	354
sqrt	4	Gummel-Poon	4	17	19

As indicated in Table 4.3, the analyzed circuit blocks are of relatively small size. Nevertheless, the generated models of the circuits are of high complexity. The corresponding schematics, testbenches, and waveforms can be found in Appendix A.

4.2 Basic Performance Measurements

Based on the assumption that unsimplified behavioral models simulated with a behavioral simulation method pose an equivalent problem to netlist-based simulation (as based on the same equations), nearly equal simulation performance should be possible. The intention of the first presented analysis is to demonstrate and quantify the performance problem of such analytical behavioral models. Therefore, several behavioral models have been directly compared to their corresponding netlist-based simulation. Figure 4.2 shows the slow-down of the behavioral simulation compared to the netlist-based simulation. Here, the discrepancy between the performance of both simulation methods increases significantly up to a factor of 192 for the µA741 operational amplifier. Even for very small models (like the emitter circuit), a non-negligible difference in simulation performance remains. It is obvious that the performance problem increases with the complexity of the modeled circuits.

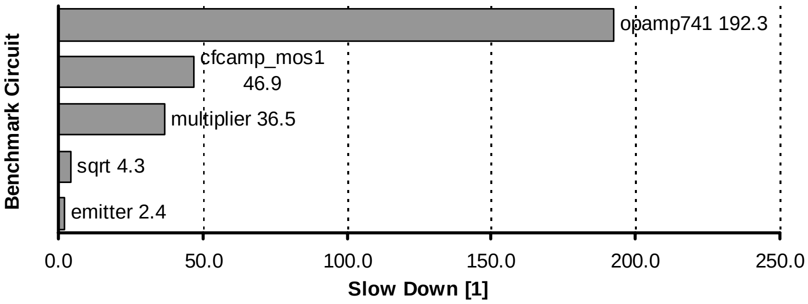


Figure 4.2: Slow Down Factors (Circuit vs. Unsimplified Behavioral Model)

Table 4.4: Results of Basic Performance Measurements

Example	***Type***	Dim	Spa	N_{step}	N_{iter}	T_{tran}	$SlowDown$
emitter	Circuit	11	72.0 %	6901	16851	0.252 s	2.4
	Model	19	83.33 %	6901	16023	0.603 s	
sqrt	Circuit	10	58.03 %	2509	9625	0.178 s	4.3
	Model	29	88.39 %	2509	6804	0.757 s	
multiplier	Circuit	28	75.72 %	1181	2361	0.102 s	36.5
	Model	113	96.12 %	1181	2801	3.727 s	
cfcamp_mos1	Circuit	23	70.87 %	1007	2022	0.099 s	46.9
	Model	149	97.51 %	1007	2042	4.642 s	
opamp741	Circuit	58	88.89 %	1669	3753	0.403 s	192.3
	Model	368	99.01 %	1391	4856	77.5 s	

Table 4.4 summarizes the simulation results also giving information on the number of time steps as well as iterations. As the internal equations of the used device models result in additional equations within the linearized system, the dimension of the behavioral models is noticeable higher than that of the corresponding circuit simulation. This also results in a much higher sparsity of those systems. The number of time steps does only reveal different behavior of the time-step control for the *opamp741*. In terms of convergence, the examples *sqrt* and *opamp741* are particularly interesting. The convergence of *sqrt* improves (which is untypical) whereas the *opamp741* faces (noncritical) convergence problems (3.5 instead of 2.25 iterations per time step) in the behavioral simulation.

In summary, the performance problems are neither caused by problems of the time-step control nor by (serious) convergence problems. The root causes for the performance problem are a matter of the processing of the models' equations and will be subject to further investigations within the following sections. Nevertheless, convergence is a critical issue for such complex behavioral models. All behavioral models created by using the symbolic BSIM3 model did reveal serious problems in DC convergence and could therefore not be simulated under these specific conditions. This problem has been solved by taking advantage of sequential equations and will be discussed within later sections.

For the time being, the above stated comparison gives the impression that the analytical modeling method by symbolic analysis is inadequate for enhancing simulation performance. The performance comparison reveals an enormous overhead. Still, the main difference between both compared simulation setups is their problem conditioning, a matter of modeling efficiency. Therefore, a major improvement of the behavioral simulation performance should be attainable. Figure 4.3 exemplarily shows the current situation and the target performance. The bottom-up generated behavioral models have to achieve a major speed-up compared to the original circuit simulation in order to apply this modeling method in an appropriate and acceptable way.

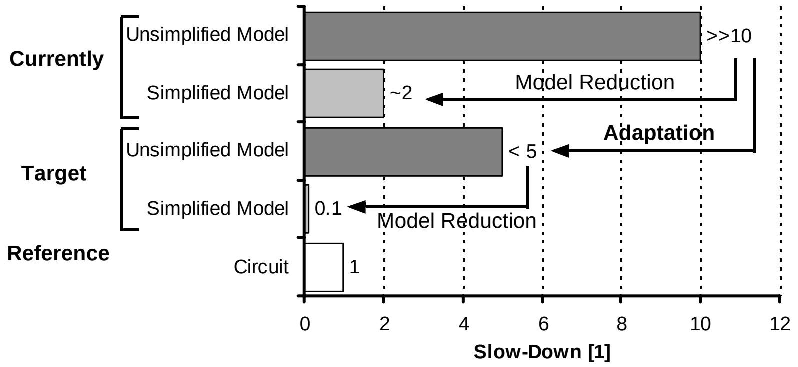


Figure 4.3: Motivation for Efficiency Improvements

The slow-down of an unsimplified behavioral model is significantly higher than 10 (for reasonable size examples) and increases considerably with the dimension of the model. For the analyzed examples, the slow-down increases up to ~200. Although the model reduction algorithms for symbolic analysis have been proven to reach highly efficient reduction rates, the reduced models are rarely faster than the netlist-based simulation. The relatively great effort of the model reduction is primarily spent on compensating for the bad simulation performance of the unsimplified model. The following research and improvements will show that a major part of this inefficiency can be eliminated by adapting the interface between the behavioral models and the simulators. This adaptation does not reduce the accuracy of the behavioral models. Its approach is rather to improve the conditioning of the problem (by reformulation and restructuring of the behavioral models) and to handle such complex behavioral models more efficiently within the model compiler. The target is to reduce the slow-down to a minimum (at most a factor of 5 slower than the reference simulation). Starting from this optimized model, nearly the full effectiveness of the existing model reduction algorithms can be used to speed-up the generated models thus making them an attractive solution for bottom-up model generation.

4.3 Distribution of the Computational Effort

Profiling is a technique to analyze where computational effort was spent during the evaluation of a program. Common profiling tools like *gprof* work on instrumented code containing additional code to count function calls and to sum-up the spent CPU time by function. The resulting tables allow statistics on how often each function was executed and how much CPU time was actually spent on it. As the results are quite comprehensive and the usage of profiling tools requires specific binaries (including the instrumentation), it is too complicated within this scope to completely profile a circuit simulator. Nevertheless, built-in time measurements facilitate analyzing the major process steps during a simulation. Such built-in profiling methods have been used to examine the distribution of the computational effort in behavioral simulations and identify possible bottlenecks and shortcomings in terms of simulation performance.

Main contributors to the total CPU time spent for a transient analysis (T_{tran}) are the loading process and the solving of the linearized system. During the loading process, the linearized system is determined by the contained models. This involves the evaluation of the Jacobian matrix (T_{jacob}) as well as the evaluation of all expressions determining the right-hand side (T_{func}) of the linearized system. Furthermore, any checks for convergence and numerical problems (T_{checks}), numerical integration, and interfacing between models and linear solver (T_{stamp}) contribute to the CPU time of the loading process (T_{load}). T_{misc} accumulates additional overhead during the loading process.

The amount of CPU time spent on solving the linear equation system (T_{solve}) can be subclassified into the CPU time for the LU decomposition ($T_{LUDecomp}$) and the forward-backward substitution to determine the solution ($T_{FBSubst}$) from the decomposed system. Equations (4.4) to (4.6) summarize the classification of the main contributors to the over-all CPU time. T_{overh} accounts for any supplementary processes during the transient analysis (initialization, time-step control, etc.) and is typically negligible small compared to T_{load} and T_{solve}.

$$T_{tran} = T_{load} + T_{solve} + T_{overh} \approx T_{load} + T_{solve} \tag{4.4}$$

$$T_{load} = T_{func} + T_{jacob} + T_{checks} + T_{stamp} + T_{misc} \tag{4.5}$$

$$T_{solve} = T_{LUDecomp} + T_{FBSubst} \tag{4.6}$$

Table 4.5: Over-All Distribution of the CPU Time

Example		T_{tran}	T_{load}	T_{solve}	T_{overh}
multiplier	absolute	3.727 s	2.49 s	1.17 s	0.06 s
	relative	100 %	66.8 %	31.5 %	1.7 %
cfcamp_mos1	absolute	4.642 s	3.28 s	1.26 s	0.11 s
	relative	100 %	70.6 %	27.1 %	2.4 %
opamp741	absolute	77.5 s	49.36 s	27.84 s	0.33 s
	relative	100 %	63.7 %	35.9 %	0.4 %

Within circuit simulations, the approximate ratio of T_{load} to T_{solve} is 80 % to 20 %. Table 4.5 indicates the distribution of CPU time in behavioral simulation, taking three behavioral models of different dimensions (113 to 368) and types (MOS Level 1 and Gummel-Poon transistor model) as example. As already mentioned, T_{overh} only contributes with less than 2.5 % and will thus be neglected for following analyses. As can be seen in Figure 4.4, the distribution for loading and solving is quite consistent within the different models, and

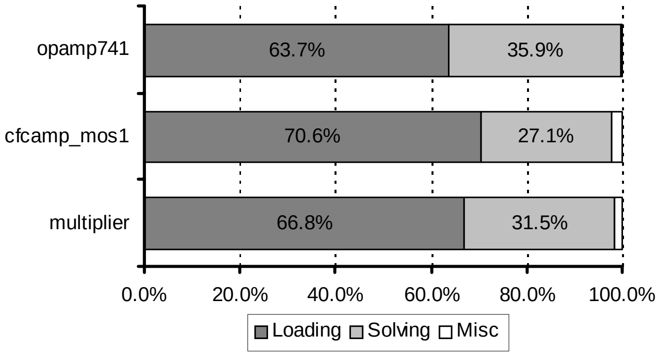


Figure 4.4: Profiling Results (Over-All)

the ratio is close to what would be expected for any circuit simulation. The over-all CPU time is slightly less dominated by the loading process.

Figure 4.5 visualizes the distribution of the computational effort spent on loading. Astonishingly, the evaluation of the right hand side (T_{func}) has the least influence on T_{load} (3 to 9 %) even though it consists of highly complex nonlinear expressions. Table 4.6 reveals the contributors and their distribution within the loading process in detail. Above all the processes handling the Jacobian matrix (T_{jacob}, T_{stamp}, T_{checks}) require a large amount of CPU time and thereby appear to be ideal for further investigation and enhancements.

Finally, Figure 4.6 and Table 4.7 show the distribution of the CPU time for solving. The LU decomposition dominates the solving process with roughly 80 % (similar to circuit simulation). As the linear solver is a fundamental component of the simulator, it was not considered to be enhanced within this context. Still, *Titan* provides different linear solvers for behavioral models that will be introduced and compared within Section 4.5. Apparently, loading and solving are equally contributing to the overhead observed in behavioral simulation. Thus, it is desirable to speed up both processes.

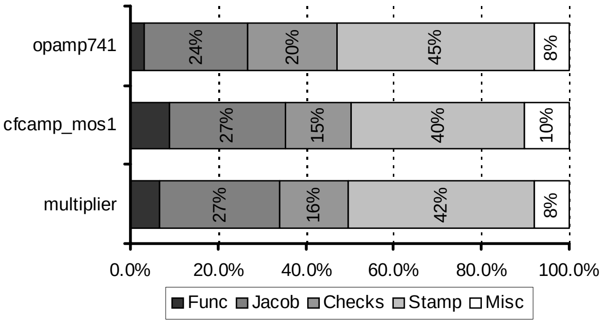


Figure 4.5: Profiling Results (Loading)

Table 4.6: Distribution of the CPU Time for Loading

Example		T_{load}	T_{func}	T_{jacob}	T_{checks}	T_{stamp}	T_{misc}
multiplier	absolute	2.49 s	0.17 s	0.68 s	0.4 s	1.05 s	0.2 s
	relative	100 %	6.7 %	27.2 %	15.9 %	42.3 %	7.9 %
cfcamp_mos1	absolute	3.28 s	0.29 s	0.87 s	0.49 s	1.29 s	0.34 s
	relative	100 %	8.9 %	26.5 %	14.8 %	39.5 %	10.3 %
opamp741	absolute	49.36 s	1.51 s	11.63 s	10.09 s	22.23 s	3.89 s
	relative	100 %	3.1 %	23.6 %	20.4 %	45 %	7.9 %

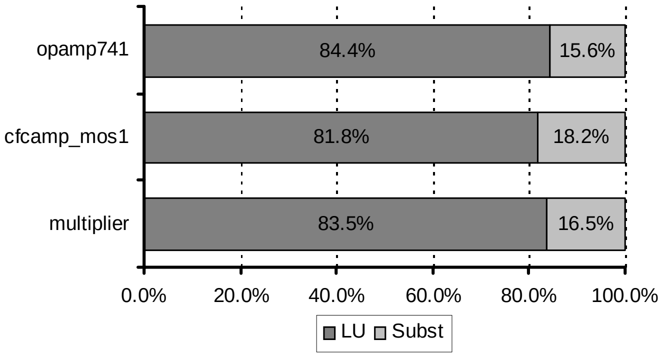


Figure 4.6: Profiling Results (Solving)

Table 4.7: Distribution of the CPU Time for Solving

Example		T_{solve}	$T_{LUDecomp}$	$T_{FBSubst}$
multiplier	absolute	1.17 s	0.98 s	0.19 s
	relative	100 %	83.5 %	16.5 %
cfcamp_mos1	absolute	1.26 s	1.03 s	0.23 s
	relative	100 %	81.8 %	18.2 %
opamp741	absolute	27.84 s	23.5 s	4.33 s
	relative	100 %	84.4 %	15.6 %

4.4 Computational Complexity of Behavioral Models

In the next step, influencing factors on both loading and solving are analyzed. Therefore, models with different characteristics but equal complexity (same accuracy) will be compared. In general, the amount of computational effort should be proportional to the complexity of the behavioral model. However, there are no sufficient metrics to measure the computational complexity of a model. Various characteristics may be used as indicators for a model's complexity. The most obvious is the number of model equations. The complexity of the contained equations can be judged by the number of terms contained, unfortunately there is no normal form for nonlinear expressions making this metric an inexact criterion. Within Section 4.7 an approach to classify models by their expression complexity will be presented (based on the instruction count IC). The number of resulting non-zeros within the Jacobian matrix is also an important characteristic influencing the computational complexity. Finally, conditioning and sorting equations and variables is essential for the linear solver.

To overcome this dilemma an approach based on the following assumptions was chosen: Different model formulations generated from the same circuit should be of equal computational complexity for the behavioral simulator. Hence, the following equation formulations have been used to set up behavioral models for three selected examples:

- Network equations set up by Modified Nodal Analysis (*MNA*)
- Network equations set up by Sparse Tableau Analysis (*STA*)
- MNA equations processed by the function `CompressNonlinearEquations` that removes some internal variables and equations by algebraic processing (*Compr*)
- MNA equations and substitution of all explicit equations (sequential equations from device models) (*Subst*)

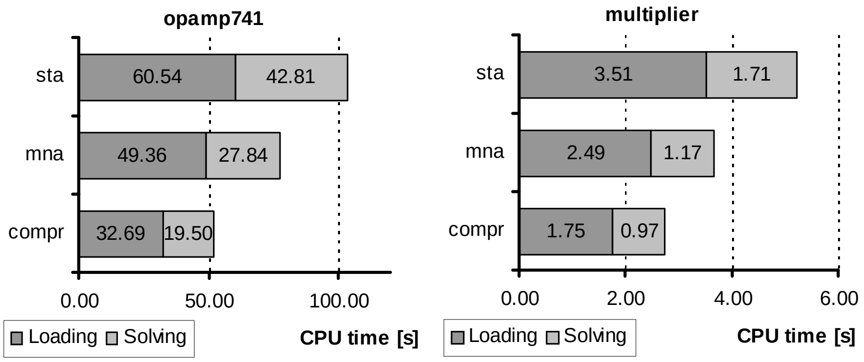


Figure 4.7: T_{tran} for Different Model Formulations (*opamp741*, *multiplier*)

Table 4.8: Results of the Complexity Analyses

Example	***Type***	Dim	Spa	$N_{nonzero}$	T_{tran}	S^{-1}
multiplier	Circuit	28	75.72 %	190	0.102 s	n/a
	Compr	94	92.95 %	623	2.77 s	27.1
	MNA	113	96.12 %	496	3.73 s	36.5
	STA	136	96.91 %	571	5.34 s	52.4
cfcamp_mos1	Circuit	23	70.87 %	154	0.099 s	n/a
	Subst	62	92.74 %	279	5.11 s	51.6
	Compr	113	95.97 %	514	2.64 s	26.7
	MNA	149	97.51 %	552	4.64 s	46.9
	STA	161	97.82 %	565	5.47 s	55.3
opamp741	Circuit	58	88.89 %	374	0.403 s	n/a
	Compr	294	98.4 %	1387	52.4 s	130.2
	MNA	368	99.01 %	1347	77.5 s	192.3
	STA	416	99.15 %	1469	103.7 s	257.3

By setting up the network equations in different formulations (*MNA*/*STA*) and algebraic post-processing of the obtained DAEs (*Compr*/*Subst*), the resulting models widely differ in their dimension and sparsity. While the *STA* models consist of a large number of equations (each of low complexity), the *Compr* and *Subst* models are of much lower dimension but contain highly complex expressions. Due to the extraordinary expression complexity, the *Subst* model type only converged in one of the presented examples (*cfcamp_mos1*).

The simulation results are shown in Table 4.8. While the number of time steps and Newton iterations do not differ at all, the CPU time significantly varies over the basically equivalent model formulations. Figure 4.7 visualizes the CPU time needed for the different model types of *multiplier* and *opamp741*. As indicated in the chart, the most compact formulation (*Compr*) achieves the best simulation performance. The distribution of solving/loading does not scale over the dimension of the models.

In Figure 4.8, the results for the *cfcamp_mos1* example are shown. While the upper three model types perform as expected from the previously shown examples, the *Subst* model type performs badly, although it is the most compact model formulation. According to its number of equations it should have performed much better. While in the other cases the dimension of the models dominates the simulation performance, the complexity of the contained expressions dominates in this example. Section 4.6 will investigate this phenomenon further and provide a metric for estimating the complexity of expressions. It is likely that a break-even point between the contradicting properties of the number of equations and their complexity exists.

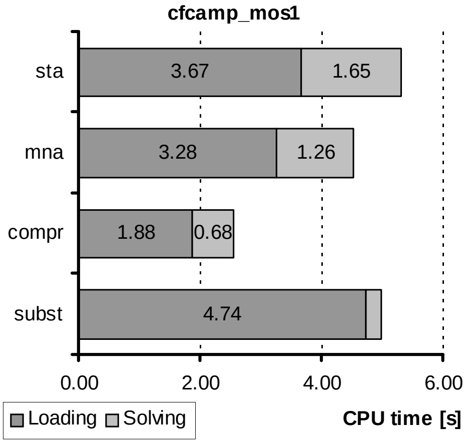


Figure 4.8: T_{tran} for Different Model Formulations (*cfcamp_mos1*)

In order to analyze the proportionality between model dimension and CPU time in more detail, statistics on multiple examples with different model formulations have been generated. These statistics purely focus on the internal processing time per iteration. They include *emitter*, *multiplier*, *sqrt*, *opamp741*, and *cfcamp_mos1* each modeled in *MNA*, *STA*, and *Compr* model style. As the examples use completely different simulation setups, models, and testbenches, a normalized metric for the CPU time was applied. Therefore, the CPU time for each simulation was normalized to CPU time per iteration:

$$T_{tran/iter} = \frac{T_{tran}}{N_{iter}}$$

Thus, any influence of the simulation time, convergence, and time step control has been eliminated. Figure 4.9 shows the coherence between the dimension of the used models and the CPU time (separately for loading, solving, and total CPU time). The CPU times scale with nearly quadratic complexity over the dimension of this selection of completely different models. This result shows that the simulation performance greatly depends on the dimension of the DAEs. Any internal differences like different equation types and different evaluation complexity do currently not influence the simulation performance observably. Hence, (4.7) is suited to estimate the CPU time for any TML model:

$$T_{Tran} = 30\text{ ns} \cdot Dim^{2,265} \cdot N_{iter} \tag{4.7}$$

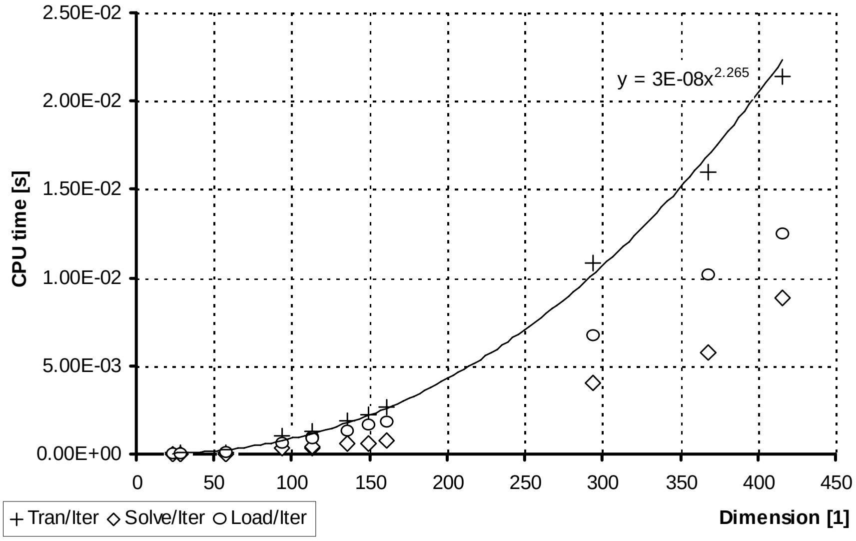


Figure 4.9: Scaling of $T_{tran/iter}$ over Dim for Various Models and Formulations

The reason for this strong dependency on the dimension of the simulated problems is the fact, that *Titan* did not apply sparse algorithms for behavioral models. As the dimension of handwritten models is typically very low, there was no need for introducing a sparse handling for the behavioral models. However, an approach to provide sparse solvers was taken and will be discussed within the following section.

For models of such high dimension as analyzed within this scope, a sparse handling is indispensable. As long as no sparse algorithms are available, the only objective to speed up the behavioral simulation is to find a model formulation as compact as possible while still enabling convergence and not exceeding a maximum expression complexity. The computational complexity of the symbolic expressions does not play an important role in this case. The CPU time for the function evaluation is completely hidden by the dominating influence of the dense data structure and processing.

Table 4.9 gives an overview of the proportionalities of different fractions of the CPU time for dense and sparse algorithms. The complexity of the expressions needed to evaluate the RHS and the Jacobian matrix is indicated by the number of evaluation cycles $N_{EvalCycles}$ which will be discussed in more detail below. In general, all CPU times linearly depend on the number of Newton iterations performed during simulation. Note that all components of the simulation have to be evaluated with the same number of iterations[1]. Hence, an apparently “non-relevant” behavioral model with poor convergence may dramatically decrease

---

[1] Multi-level Newton-Raphson methods solve this issue [34]

the over-all simulation performance of a large simulation setup by requiring a large number of iterations. The same is true for time steps.

Table 4.9: Proportionalities of CPU Time and Model Characteristics

	Dense Algorithms	***Sparse Algorithms***
Solving	$T_{solve} \sim N_{iter}, Dim^2 \ldots \frac{Dim^3}{3}$	$T_{solve} \sim N_{iter}, Dim^{1,2\ldots1,5}, Spa$
Function Evaluation	$T_{func} \sim N_{iter}, Dim, N_{EvalCycles}$	$T_{func} \sim N_{iter}, Dim, N_{EvalCycles}$
Jacobian Evaluation	$T_{jacob} \sim N_{iter}, Dim^2, N_{EvalCycles}$	$T_{jacob} \sim N_{iter}, N_{nonzero}, N_{EvalCycles}$
Checks, Stamping	$T_{checks}, T_{stamp} \sim N_{iter}, Dim^2$	$T_{checks}, T_{stamp} \sim N_{iter}, N_{nonzero}$

When using dense algorithms the main contributors scale polynomially over the dimension, whereas sparse algorithms provide an efficient method to reduce this proportionality to a close to linear scaling. The exponent in the sparse case depends on the sparsity and conditioning of the Jacobian matrix, and typically varies from 1.2 to 1.5. For the processing steps during loading, the relevant characteristic is the number of non-zero entries in the Jacobian matrix, which can be calculated as follows:

$$N_{nonzero} = (1 - Spa) \cdot Dim^2$$

4.5 Performance of Linear Solvers

Linear solvers are based on a trade-off between performance, accuracy, and numerical robustness. As already discussed in Section 3.5, *Titan* uses different solvers for the subsystems resulting from the structural description (by direct inspection of the netlist elements) and the subsystem of the model equations. By default, the dense *LAPACK* solver was used for behavioral models as they were usually of low dimension but numerically critical to handle. The other solvers are optionally available. The intention of this section is to demonstrate a comparison of the performance of the available solvers. The simulation results will be exemplarily demonstrated by the *emitter* model with a low dimension of only 19 and the *opamp741* model, which is of reasonable high dimension (368). Table 4.10 lists the performance measurements for both examples simulated with the three available solvers.

Table 4.10: Distribution of the CPU Time for Different Linear Solvers

Example	***Solver***	N_{iter}	$N_{iter/step}$	T_{tran}	T_{load}	T_{solve}	S_{solve}
emitter	*LAPACK*	16023	2.32	0.6 s	0.34 s	0.13 s	1
				100 %	55.9 %	20.7 %	
	Titan	15841	2.3	0.5 s	0.32 s	0.03 s	4.33
				100 %	64.2 %	6.4 %	
	MUMPS	16037	2.32	1.9 s	0.41 s	1.35 s	0.096
				100 %	21.4 %	70.9 %	
opamp741	*LAPACK*	4856	3.49	77.52 s	49.36 s	27.84 s	1
				100 %	63.7 %	35.9 %	
	Titan	6467	4.65	65.82 s	64.79 s	0.7 s	39.77
				100 %	98.4 %	1.1 %	
	MUMPS	4856	3.49	59.01 s	52.34 s	6.36 s	4.38
				100 %	88.7 %	10.8 %	

For the *emitter* model the choice of the linear solver does not influence the convergence of the simulation. As the model is of low dimension, the *Titan* solver speeds up the solving process by a factor of (only) 4.33. The application of the *MUMPS* solver significantly reduces the simulation performance (presumably by the overhead caused by initializing the solver and needlessly performing dynamic pivoting). Unlike in the *emitter* example, the solving performance for the *opamp741* shows a substantial improvement of a factor of 39.77 (*Titan* solver) respectively 4.4 (*MUMPS* solver). Due to an unfavorable pivoting of the Jacobian matrix, the *Titan* solver needed 6467 instead of 4856 iterations, which in turn decreased the loading performance. Hence, the *MUMPS* solver would be the optimal solution for the *opamp741*, but would be of major disadvantage for the *emitter* circuit. It is worth mentioning that the ratio between loading and solving (98 % to 1 %) for the *opamp741* simulated with the *Titan* solver indicates that there was still a problem during loading. Other simulations confirm these results.

The robustness of the *Titan* solver in combination with a sub-optimal ordering of the model equations did cause severe convergence problems for some applications. This could be solved for future applications by performing a sufficient preordering of the models' equations to ensure a favorable conditioning of the Jacobian matrix. In general, the application of a sparse solver is essential for systems of high dimension. Should convergence problems occur, the *MUMPS* solver is the preferred solution. In terms of performance, the preconditioning of the models (to improve robustness) and application of the standard *Titan* solver would most likely lead to the best results. The default *LAPACK* solver is not suited for behavioral models of higher dimension. Here, a sparse solver (*Titan* or *MUMPS*) should be applied.

4.6 Loading Performance

While the simulation efficiency of the solving process could be resolved by applying an appropriate sparse linear solver, the loading process (especially in combination with the *Titan* solver) consumes nearly the complete CPU time (98 %). Research proved that the loading process of the compiled behavioral models was not realized in a sparse manner. In order to eliminate any side effects and to only measure the performance in terms of loading and linear solving two types of linear networks were set up:

- *Chain networks* of resistors with each node additionally connected to ground (see Figure 4.10) → tridiagonal Jacobian matrix, high sparsity
- *Complete networks* with each node connected to each other by a resistor (see Figure 4.11) → fully populated Jacobian matrix, no sparsity

As the networks are static linear, the simulation performance is not influenced by dynamic effects or linearization issues. The Jacobian matrices of netlist and behavioral simulation are identical (apart from the additional port currents for the behavioral models, which are negligible).

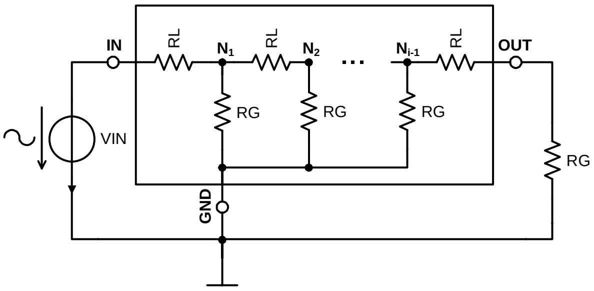


Figure 4.10: Schematic for *Chain Networks*

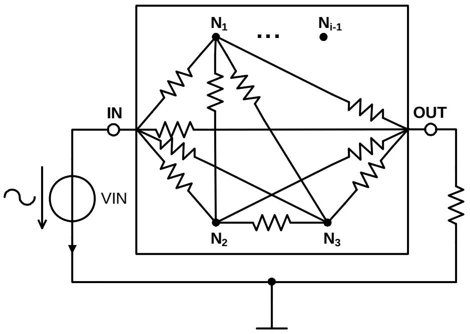


Figure 4.11: Schematic for *Complete Networks*

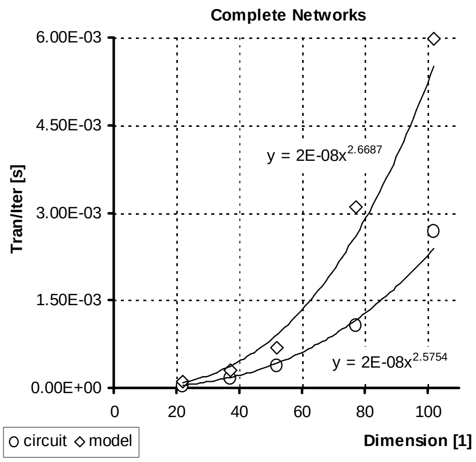


Figure 4.12: Scaling of $T_{tran/iter}$ over Dim for the *Complete Networks*

For the complete networks all benefits from sparse handling strategies are disabled as the matrices are (nearly) fully populated. The scaling of the simulation performance over dimension and sparsity can be observed by varying the number of nodes (20 to 100) of both network types. As indicated by the previous analyses, the *Titan* solver was applied to enhance the solving performance. Despite the linear nature of the problems, transient simulations with limited step size and sinusoidal input sources have been performed to achieve a large number of iterations (for accurate CPU time measurements).

Figure 4.12 shows the CPU time per iteration over the dimension of the complete networks and their corresponding behavioral models. As expected, the CPU time scales with a potential function of an exponent larger than 2. Although sparse algorithms do not affect this network type and the linearized systems are (nearly) identical, the behavioral simulations are by a factor of 2 slower than the corresponding netlist-based simulations. As Figure 4.13 verifies, this overhead in behavioral simulation is solely caused by the loading process, whereas solving is of exactly the same performance.

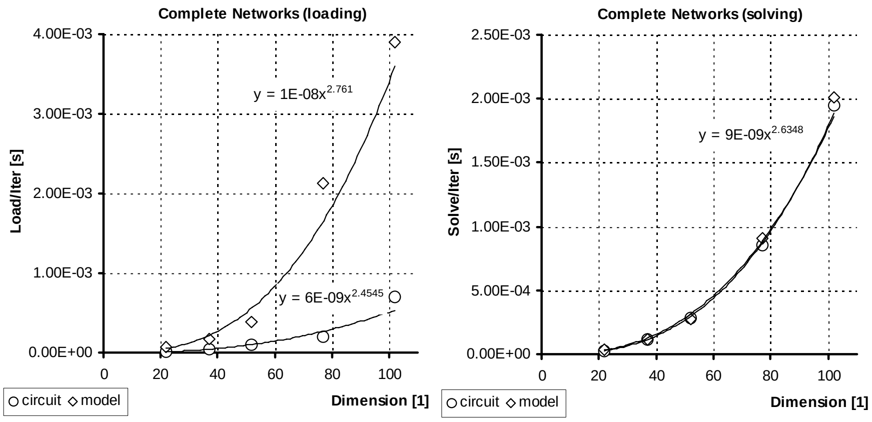


Figure 4.13: Scaling of $T_{load/iter}$ and $T_{solve/iter}$ over Dim for the *Complete Networks*

As far as chain networks are concerned, the overhead of the behavioral simulation is much more serious as Figure 4.14 shows. Due to efficient sparse algorithms the CPU time of the netlist-based simulation scales close to linear. In contrast to that, the behavioral simulation scales with an exponent of 1.6 causing a rapidly increasing discrepancy between both simulation types. Comparing the distribution between loading and solving (see Figure 4.15) evidences the application of the *Titan* solver: the solving process scales linearly and the discrepancy between both types is acceptable.

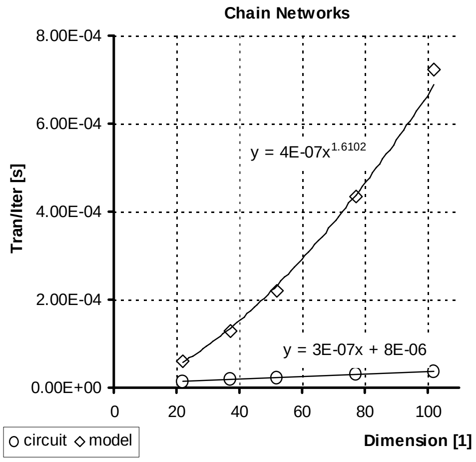


Figure 4.14: Scaling of $T_{tran/iter}$ over Dim for the *Chain Networks*

The loading process shows dramatic inefficiency. For the behavioral models, the exponent of the potential fitting function is 1.73. As the sparsity of the models is not utilized within this realization, a major improvement of the loading performance by introducing a sparse processing should be expected. Some effective measures to reduce the overhead to a minimum will be presented in Section 5.2.

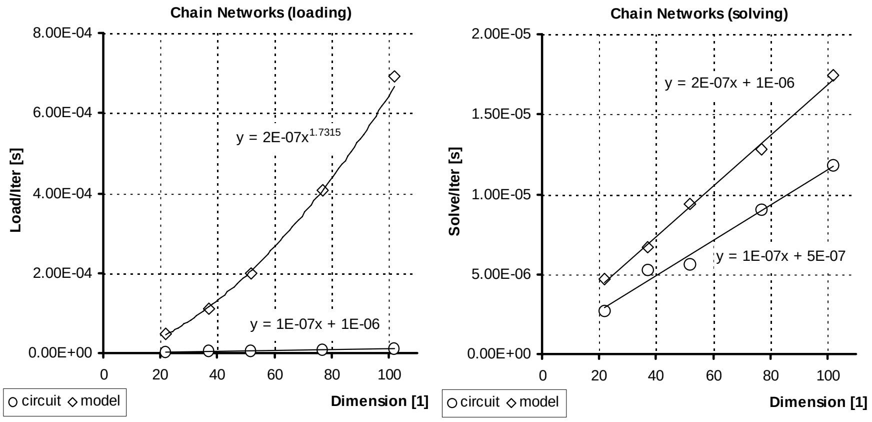


Figure 4.15: Scaling of $T_{load/iter}$ and $T_{solve/iter}$ over Dim for the *Chain Networks*

4.7 Expression Evaluation

The loading process primarily consists of a large number of expression evaluations in order to determine the residual as well as the Jacobian matrix for the next iteration of the linear solver. The spent CPU time within these processes (T_{func} and T_{jacob}) is influenced by the clock cycles required for the evaluation of the nonlinear expressions and the time to access the necessary data within the memory. While it is relatively easy to predict the computational effort for the expression evaluation itself, memory accesses are highly dependent on data structures and memory organization. Above all, data locality is of great importance as it determines the cache miss rate. The miss penalty for data that has to be fetched from the main memory instead of being available within the cache is (depending on the computer architecture) 10 to 16 times higher than a successful cache access. Unfortunately, the cache miss rate is hard to predict. The realization of the sparse data structure for the loading process (cf. Section 5.2) will demonstrate the importance of data locality.

Two metrics have been introduced in order to estimate the computational complexity of a system of DAE: the number of clock cycles for function evaluation ($N_{EvalCycles}$) and the number of memory accesses ($N_{MemAccess}$). The instruction counts IC_i for all contained arithmetic functions of the equation set as well as the instruction count for the symbolic Jacobian matrix were determined within the model generator based on the function `Cost` [70]. Some basic expression optimizations that would be performed by a compiler (strength reduction, cf. [1]) have already been taken into account within [70] (e.g. $x^2 \rightarrow x \cdot x$, $x^y \rightarrow e^{y \cdot \ln(x)}$). Afterwards, the sum of the instruction counts for each function weighted by an estimated cycles per instruction figure CPI_i (cf. Table 4.11) has been determined by (4.8). The resulting figure $N_{EvalCycles}$ does not take into account CPU architecture specific properties and non-ideal behavior of instruction level parallelism (ILP, cf. [30]). It should therefore only be considered as a rough indicator for the evaluation complexity.

$$N_{EvalCycles} \approx \sum_{i=1}^{n} IC_i \cdot CPI_i \tag{4.8}$$

Table 4.11: Cycles per Instruction

Instruction	***CPI***
addition, subtraction, multiplication	1
division, square root	4
exponential function, logarithm, sinus, cosine, etc.	8
power	17

The number of memory accesses (to the data memory) during the expression evaluation was derived from the total count of the referenced variables and parameters within the equations and their Jacobian matrix. No distinction was made between read and write accesses.

(4.9) shows the (measurable) influencing factors for the expression evaluation. There are additional dependencies on computer architecture, cache miss rate, pipelining, etc. that can neither be measured nor estimated easily.

$$T_{func} + T_{jacob} \sim N_{EvalCycles}, N_{MemAccess}, N_{iters}, \ldots \tag{4.9}$$

In order to estimate the influence of the expression evaluation on the loading performance two model types will be compared, the unsimplified models in MNA formulation and their corresponding models with all parameters replaced by their numerical default values. Thereby, the expression complexity changes remarkably as some subexpressions, that only consist of parameters, can be pre-evaluated within the modeling tool instead of being evaluated during simulation. At the same time, the structure and numerical properties of the DAE system do not change. As mentioned within the previous sections, the missing sparse handling during loading heavily influences the loading process. As this effect dominates the loading process, the influence of the expression complexity cannot be clearly diagnosed. Therefore, the statistics were processed with the sparse implementation that will be presented within Section 5.2.

Table 4.12 lists the estimated clock cycles $N_{EvalCycles}$ for the function and Jacobian evaluation as well as the estimated number of memory accesses $N_{MemAccess}$ and their respective reduction by replacing the parameters with their numerical values for three examples. The two right-most columns display the achieved speed-up in the residual's and Jacobian's evaluation. Apparently, the removal of the parameters efficiently reduces the expression complexity and the number of necessary memory accesses (by roughly 40 %). The reduced complexity of the expressions positively affects the simulation performance. It results in a speed-up of roughly 25 % for the function and the Jacobian evaluation. The discrepancy to the optimal speed-up of 40 % most likely originates from influencing factors of the computer architecture that could not be taken into account. The good correspondence between CPU time and expression complexity in this experiment is evidence for considering the function evaluation for further optimization.

Table 4.12: Complexity of the Expression Evaluation

Example	***Parameters***	$N_{EvalCycles}$	***Ratio***	$N_{MemAccess}$	***Ratio***	S_{func}	S_{jacob}
multiplier	104	7946	68.2 %	2846	51.4 %	1.23	1.16
	0	5419		1462			
cfcamp_mos1	155	7300	62 %	4203	63.4 %	1.25	1.33
	0	4532		2667			
opamp741	354	17085	64.5 %	7136	56.2 %	1.22	1.34
	0	11020		4012			

4.8 Comparison of Commercial Simulators

It is not in the scope of this research to compare different simulators. Nevertheless, a comparison with three major commercial simulators was made to ensure that the presented results and performance problems are not a *Titan* specific problem. For legal reasons all results will be presented anonymously. As *Titan* is probably the most famous moon of Saturn, three more satellites of Saturn will be used as pseudonyms for those simulators:

- *Dione* - a circuit simulator with Verilog-A support
- *Rhea* - a mixed-signal simulator supporting VHDL-AMS and Verilog-AMS
- *Thetys* - another mixed-signal simulator supporting VHDL-AMS and Verilog-AMS

Results simulated with different simulators must not be absolutely compared as it is generally impossible to guarantee fair conditions among different simulators. Therefore, each simulator was relatively compared to itself by determining the slow-down between its behavioral and circuit simulation. The accuracy options have been chosen to achieve similar settings as the *Titan* defaults (which are reasonably accurate). All measurements have been extracted from the simulator's log files (apart from the number of iterations that were not reported by all simulators). In order to be able to evaluate the convergence for those simulators a Verilog-A model measuring the number of iterations was used (see Appendix A.8).

The basic analyses were performed under the same conditions as for *Titan* in Section 4.2. For *Rhea* and *Thetys*, the model with the better simulation performance (VHDL-AMS or Verilog-A) was used within the statistics. Figure 4.16 shows the slow-down factors of the behavioral simulation compared to the circuit simulation of each simulator for the examples *cfcamp_mos1* and *opamp741*. The factors have been calculated from three different measures, the absolute CPU time spent for the transient analyses, the CPU time normalized to the number of time steps, and finally the normalized CPU time per iteration. Even though the

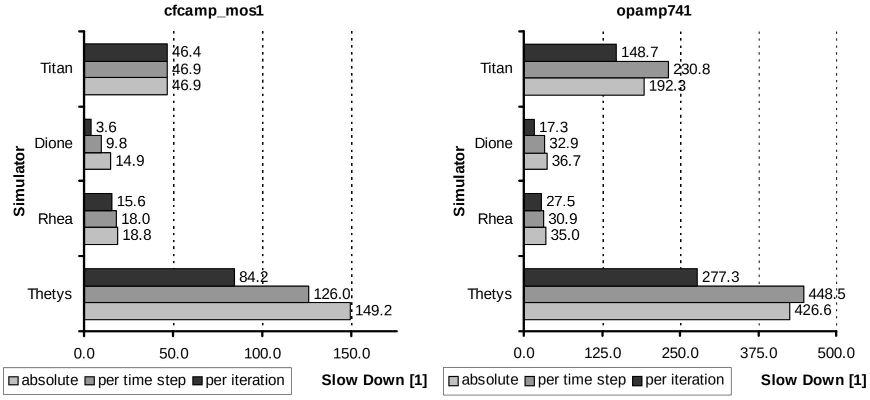


Figure 4.16: Simulator Comparison for *cfcamp_mos1* and *opamp741*

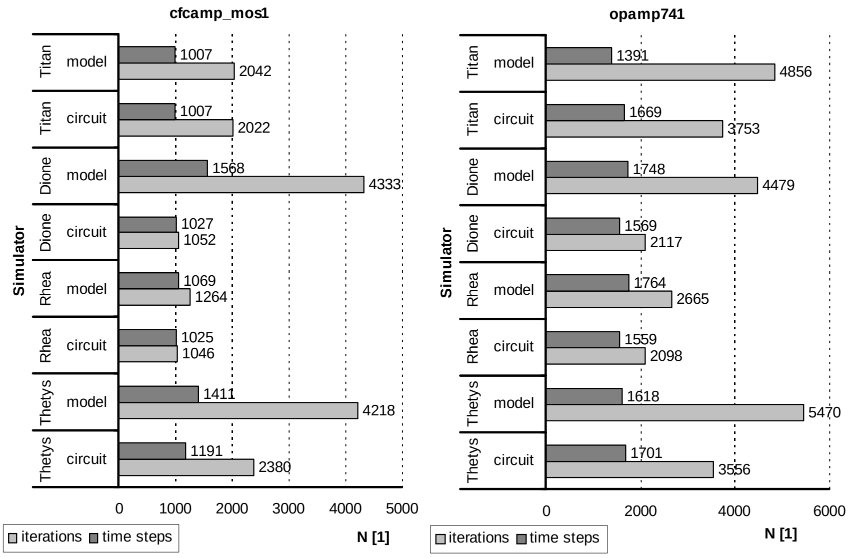


Figure 4.17: Convergence Comparison for *cfcamp_mos1* (left) and *opamp741* (right)

simulators show a great variety of different performances, no simulator reaches the expectations of being below five times slower than its netlist-based simulation. Moreover, the performance problem seems to increase for all simulators with a rising dimension of the models (as further analyses will demonstrate). It is worth mentioning that *Titan* was still measured using its default dense algorithms and solver for simulating the models at this state.

By comparing the number of necessary time steps in Figure 4.17, a rather similar behavior in time-step control can be seen. *Dione* and *Thetys* suffer from serious convergence problems in behavioral simulation for the used models. As presented for *Titan* in Section 4.6, these simulators were also analyzed about their scaling over the problem dimension. Although VHDL-AMS would have been the better solution to model the linear networks, *Rhea* had to be measured with Verilog-A models due to large memory consumption during the VHDL-AMS models' compilation (resulting in a compiler error). Figure 4.18 (dense) and 4.19 (sparse) give a qualitative overview of the simulation results for the analyzed simulators. For the complete networks, the netlist-based simulations show remarkable differences between the compared simulators. The performance of the behavioral models is significantly worse than that of the netlist-based simulations.

The analysis of the chain networks reveals that all simulators use sparse techniques (linear scaling). The performance of the circuit simulations is similar for all simulators. Still, the overhead in processing the behavioral model is considerably large for *Dione* and *Rhea*. The gradient of the linear approximation functions for those simulators differs by a factor of 8 and 6.66 respectively compared to the netlist-based simulation. *Thetys* has a very small overhead between both simulation modes. Its behavioral simulation performance is nearly as

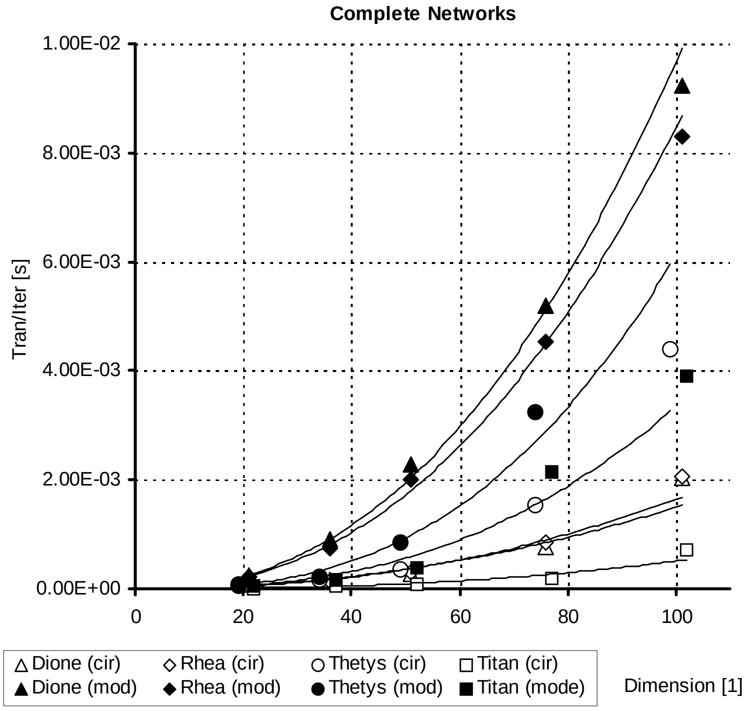


Figure 4.18: Simulator Comparison on the
Scaling of $T_{tran/iter}$ over Dim for the *Complete Networks*

good as the performance of the netlist-based simulations. *Titan* was not included in this analysis as it has shown a super-linear scaling in previous analysis due to the missing sparse handling of the behavioral models.

The significant overhead between the netlist-based simulation and the corresponding behavioral simulation is a characteristic all simulators have in common. This discrepancy can only be caused by the internal processing of the models within the loading process. Other differences between the simulation modes like convergence, time-step control, and linear solving issues (including sparse algorithms) have been eliminated by the experiment's setup. It is highly desirable in order to reduce this inefficiency to enhance the behavioral simulation performance. Apart from the relatively high dimension of the models, their structure and their contained equations represent one of the most basic models possible. For any more complicated model the performance problems would most probably get worse. For a more detailed overview of the corresponding charts and fitting functions please refer to Appendix C.1.

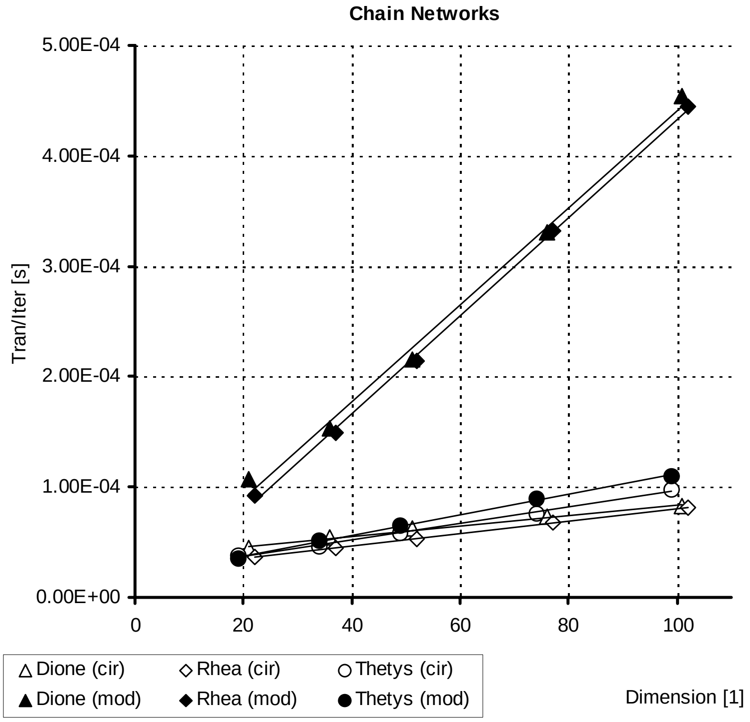


Figure 4.19: Simulator Comparison on
the Scaling of $T_{tran/iter}$ over Dim for the *Chain Networks*

4.9 Taking Advantage from Sequential Equations

A promising approach to reduce the dimension of a linearized system that has to be solved during simulation is to take advantage of sequential equations (see Section 2.1 for definition and example). As device models typically contain a high ratio of explicit equations it is obvious to model those equations in a manner that they do not have to be solved with iterative methods. Therefore, the modeling language has to support some kind of procedural evaluation (as discussed in Section 2.4) and the simulator has to deal efficiently with this method.

So far, all model equations have been modeled simultaneously and hence had to be solved simultaneously. As *Analog Insydes* supports sequential equations and Verilog-A offers useful possibilities to model the sequential equations (using procedural assignments), the advantage of solving these equations procedurally will be analyzed within this section. Unfortunately, TML and *Titan* did not provide the possibility to generate and simulate behavioral models with sequential structure. This issue has been solved and will be presented within Chapter 5. For now, the analyses have been performed by using the simulator *Dione* and Verilog-A models. The information which equations can be handled sequentially is provided by the symbolic device models.

Table 4.13: Results of Sequential Simulations with *Dione*

Example	***Type***	Dim	N_{SeqEqs}	N_{SimEqs}	N_{iter}	N_{step}	T_{tran}
cfcamp (MOS1)	Circuit	30	n/a	n/a	1052	1027	0.137 s
	Seq. Model	69	87	49	2142	1053	0.639 s
	Sim. Model	156	0	136	4333	1568	2.044 s
opamp741	Circuit	57	n/a	n/a	2117	1569	0.188 s
	Seq. Model	243	177	127	4646	1786	4.878 s
	Sim. Model	420	0	304	4479	1748	6.891 s
cfcamp (BSIM3)	Circuit	20	n/a	n/a	5195	4417	0.75 s
	Seq. Model	190	1205	123	8160	3691	63 s
	Sim. Model	1395	0	1328	(no convergence)		

Table 4.13 shows the number of (effective) sequential and simultaneous equations as well as the simulation results for three examples with three different simulation types each: the netlist-based simulation (circuit), the behavioral model with use of sequential equations (seq. model), and the simultaneous behavioral model (sim. model). The dimension of the linear system that is processed by the simulator kernel is the sum of the simultaneous equations, equations resulting from netlist elements of the testbench, and auxiliary variables (as discussed in Section 4.1).

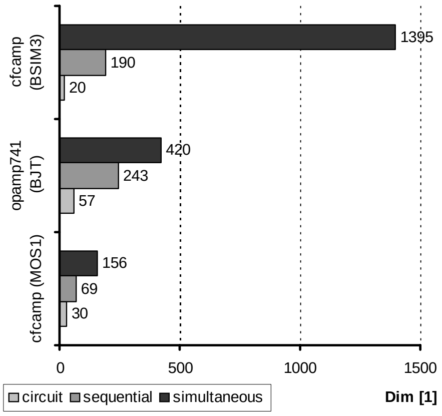


Figure 4.20: Dimension of the Linear Systems

Figure 4.20 shows the dimensions of the resulting linear systems. By handling a large portion of the equations sequentially, the dimension was effectively reduced, but is still significantly larger than for the netlist-based simulation. As can be seen from the number of iterations and time steps, handling equations sequentially (instead of simultaneously) improved convergence and time-step control for *cfcamp_mos1*, but led only to minor changes in terms of convergence for the *opamp741*.

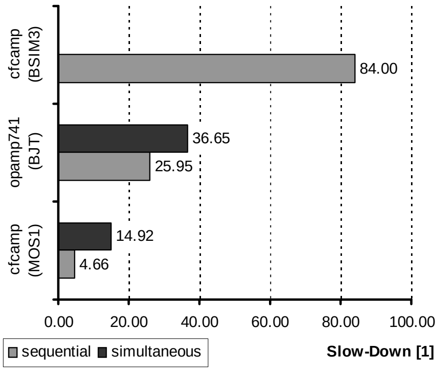


Figure 4.21: Simulation Performance of the Sequential Models

As shown in Figure 4.21, the simulation performance increased by the introduction of the sequential equations, but is still far from being competitive to the netlist-based simulation. For large examples like the *cfcamp* (modeled with BSIM3), no convergence could be achieved without sequential equations. Even with consideration of sequential equations, the performance is very low (slow-down of factor 84).

Modeling and solving explicit equations in a sequential manner typically improves convergence and reduces the CPU time (approximately proportionally to the reduced dimension of the linear system). The CPU time is mainly determined by the number of simultaneous equations whereas the number of sequential equations is only of secondary effect. Therefore, it is advantageous to formulate as many equations as possible in an explicit form (see Section 6.1 for an optimization strategy).

5 Compilation of Analytical Behavioral Models

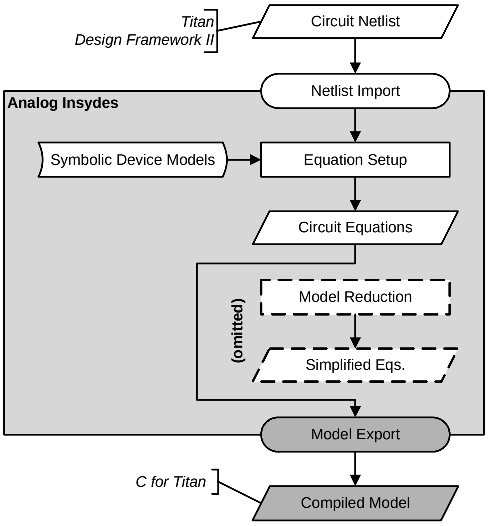


Figure 5.1: Bottom-Up Modeling Process with Compiled Model Generation

Within this chapter, enhancements to increase the behavioral simulation efficiency of *Titan* will be presented. Above all, the model compilation has been effectively improved whereas enhancements within the simulator kernel (linear solver, time-step control, etc.) have been of lower priority.

The general architecture of (most) behavioral simulators has been presented in Section 3.5. Unfortunately, only little information on specific details of commercial simulators has been published (except for the Language Reference Manuals of the AHDLs). Therefore, no comparison of the algorithms applied to *Titan* and commercial tools can be drawn. Previous experiments led to the assumption that no sufficient optimization of complex model equations is performed within commercially available model compilers. These compilers seem to rather focus on translating an AHDL into their intermediate language without performing optimizations. While the code generation will be addressed within this chapter, possible optimizations of the DAEs will be presented in Chapter 6. Optimizations as well as model compilation have been integrated into *Analog Insydes* and can thus be combined modularly. Figure 5.1 shows the modeling flow, highlighting the new model compilation for *Titan*.

As already mentioned, the compilation of analytical behavioral models has many analogies with compiling device models. Within device model compilers, device models realized in an AHDL are translated for a simulator specific interface (typically C/C++). Approaches to achieve a high-performance compiled model are quite similar. Different from (behavioral) model compilers, the objective of optimizing the model realization is of high priority within device model compilers, as it is essential to generate a model that is of at least equal performance as a manual implementation. Device model compilers' performances close to the targeted "human-optimized" implementation have been reported in [13, 76, 80].

The problems with compiling analytical behavioral models are of a more general nature. Typically, these models are of much higher dimension and have to be processed completely automatically. In contrast to that, device model compilation is more specialized (e.g. requires admittance formulation, uses physical and topological knowledge, etc.). It is often based on hand-optimized Verilog-A models, and may be accompanied by manual enhancements of the resulting model. Experiments with the *adms* compiler [44] revealed that this strategy is (not yet) applicable to general DAE-based behavioral models.

The intention of the enhancements applied to the *Titan* AMS environment was to improve both the simulation performance and the numerical robustness. Main objectives were

- the realization of a sparse loading,
- a new high-performance handling of sequential equations,
- the avoidance of restrictions resulting from modeling languages, and
- the reduction of the overhead of the former model compiler (as only very limited features of TML were needed).

The latter aspect addresses a specialized compilation method for the analytical models to achieve a close and direct interaction between the compiled models and the simulator kernel, which will be described within Section 5.3.

5.1 Tuning Simulator Options for Performance

Before presenting any enhancements to the simulator, some basic issues that might be influenced by simulator options will be addressed. The numerical solution of DAE systems is particularly sensitive to parameters influencing the behavior of the linear solver, the convergence criteria, and the integration methods. In [39], a comprehensive discussion of the most important simulator options is presented. Within this context, only some brief information on how to enhance simulation performance will be given.

The most effective method to reduce the CPU time of simulations is to keep the number of Newton iterations N_{iter} as small as possible, because nearly all contributors to the CPU time scale over N_{iter}. Therefore, it is important to avoid unnecessary large numbers of time steps as each time step requires at least one (more likely two) Newton iterations. Main causes for this might be

- a small time-step limit,
- a low iteration limit per time-step,
- over-accurate integration tolerances,
- unphysical fast signal edges, or
- break points (e.g. resulting from fine-granular PWL-sources, break-statements, or synchronization issues with digital components in mixed-signal simulations).

However, reducing the number of time steps not necessarily reduces the CPU time in a linear way: Small time-steps typically improve the convergence within each step (as the solution for the time step is closer to the previous solution). Taking fewer time steps might consequently impair convergence.

Other important aspects influencing the number of Newton iterations are accuracy options and tolerances of models. Choosing too restrictive accuracy options might result in serious convergence problems or even non-convergence. Careless or unintentional loosening of tolerances holds the risk of wrong simulation results and non-convergence. The default simulator settings might be too liberal for very sensitive circuits (e.g. bandgap). A general advice on accuracy options cannot be given. In doubt, special attention should be paid to the corresponding simulator options and tolerances specified within the model. The tolerances in behavioral models should be selected according to the physical meaning of the model's variables.

Saving values during simulation should also be handled with care. As the file I/O to store the waveforms during simulation requires a considerable amount of time, only variables of particular interest should be selected to save. Saving current values should also be used carefully as monitoring a current not only results in a longer processing time but, more important, in additional equations within the linear system. This results from the fact that the branch current to monitor is most likely not an unknown of the linear system (in MNA formulation), and therefore an additional voltage source with zero voltage is introduced within the network to obtain the necessary current variable.

For *Titan*, the linear solver should be changed to one of the sparse solvers (as discussed in Section 4.5). This improves the simulation performance for models of "higher dimension" (from approximately 10 equations) significantly.

5.2 Sparse Loading

For complex behavioral models resulting in Jacobian matrices of high dimension but with a low ratio of non-zero entries, sparse loading becomes a serious issue (for details see Section 4.6). Above all, the handling of the Jacobian matrix (storage, evaluation, copying) turned out to be worth an increased effort to utilize sparsity. As *Titan* generates a fully symbolic Jacobian matrix by automated derivation of the model's equations, a large number of complex expressions to determine the Jacobian's non-zeros is necessary. Nevertheless, the CPU time is not dominated by the evaluation of these highly complex nonlinear expressions. In fact, cache effects dramatically slow down the performance during the expression evaluation. These cache effects are caused by a low data locality due to the dense storage of the matrix. The sparse realization avoids processing and storage of structural zero entries of the Jacobian matrix. Moreover, storing the Jacobian matrix in a sparse data structure results in an improved data locality and effectively reduces the cache miss rate.

Saad [68] provides a good introduction to sparse storages and methods. The realization of a so-called coordinate data structure and the corresponding changes in the processing of the Jacobian matrix efficiently increases the loading performance. This very basic sparse matrix format stores only the non-zeros and their index within the matrix. All operations that have previously been performed based on the complete matrix may easily be adapted to this data structure. Figure 5.2 visualizes both matrix storage formats for a small example. The memory consumptions:

$$Mem_{dense} = Dim^2 \cdot 64\ \text{Bit}$$

$$Mem_{COO} = N_{nonzero} \cdot (64\ \text{Bit} + 2 \cdot 16\ \text{Bit})$$

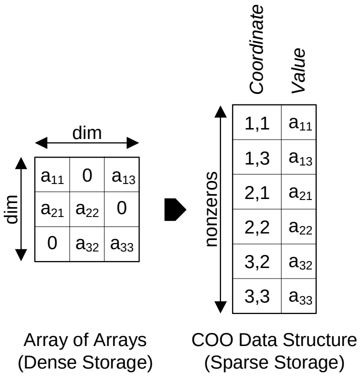


Figure 5.2: COO Data Structure

For matrices of typical dimensions (like depicted in Figure 5.3 for the *opamp741*), the effect on memory consumption and data locality is enormous. This matrix contains 1214 nonzeros at a dimension of 317 (sparsity of 98.8 %, memory reduced from 785 kB to 14 kB). Although very basic, the COO format is absolutely sufficient as an intermediate storage within the loading process as neither fill-ins nor reorderings are necessary. Thanks to its simplicity, the handling of this sparse data structure results in a minimal effort for initialization and administrative overhead.

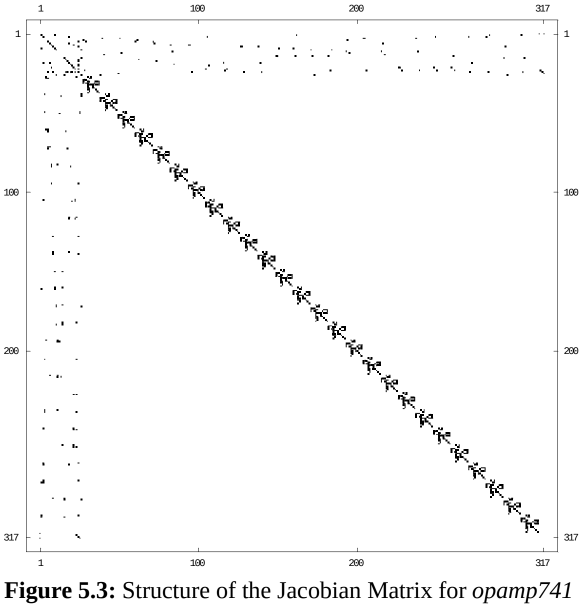


Figure 5.3: Structure of the Jacobian Matrix for *opamp741*

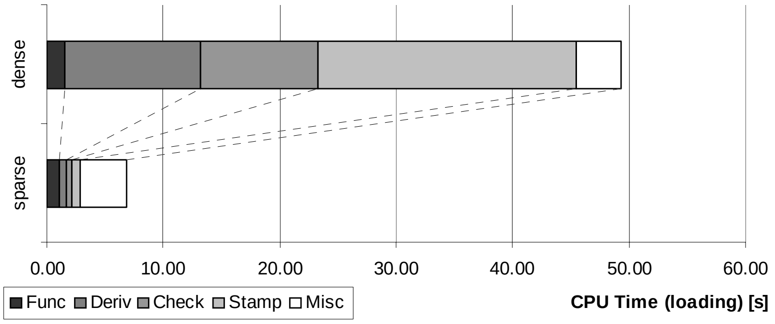


Figure 5.4: CPU Time for Loading Using the new Sparse Loading (*opamp741*)

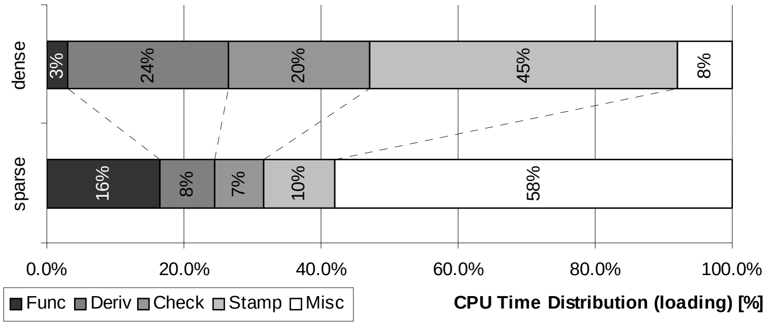


Figure 5.5: Distribution of the CPU Time During Loading

Table 4.6 on Page 47 contains the preceding profiling results for the loading process of the *opamp741* example. By repeating the analysis with the sparse loading method a significant improvement of the performance could be observed. Figure 5.4 shows the improvement of the sparse processing during loading. The evaluation of the Jacobian matrix, any checks performed on the Jacobian matrix, and the process of copying the intermediate Jacobian to a shared memory for the linear solver have been sped up by individual factors of 20 to 30. The complete loading process was sped up by a factor of 7. Merely the function evaluation for determining the residual (Func) and the processing overhead (Misc) were not affected. Hence, the distribution of the CPU time changed as shown in Figure 5.5. The processing overhead now clearly dominates the loading process. Applying the sparse solver and realization of the sparse loading resulted in a total speed-up of a factor of 10 for the *opamp741*. Table C.1 in Appendix C.2 contains the corresponding profiling results.

Figure 5.6 (left) depicts the CPU time per Newton iteration over the dimension of the chain networks (see Section 4.6) analyzed with the new sparse loading implementation. The CPU time for the behavioral simulation is now of linear complexity, too. Unfortunately, the ratio

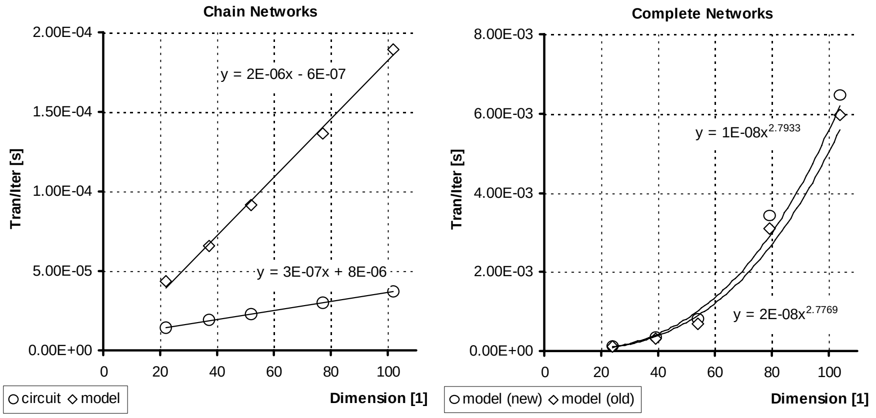


Figure 5.6: Performance of Chain Networks with Sparse Algorithms (left) and Overhead of Sparse Loading for Dense Matrices (right) with TML Compiler

of the gradients of the fitting functions between behavioral and netlist-based simulation is still significantly larger than one (6.66). This means that the performance of the behavioral simulation, even in the best case of same convergence, is of at least a factor of roughly 7 slower than its netlist-based equivalent. The main reason for this is the loading process (with a gradient ratio of 16) while the ratio of 1.55 for the solving process is acceptable (for the corresponding charts refer to Appendix C.2.4). Thus, the total discrepancy between both simulation modes caused by overhead within the loading of behavioral models will still be relevant for models of higher dimension.

Figure 5.6 (right) shows the behavioral simulations for the complete networks with and without sparse loading. As sparse loading has no effect on the complete networks, the difference between both graphs represents the additional overhead by initializing and processing the sparse data structure which is negligible. Hence, the improved loading algorithm should be advantageous even for Jacobian matrices of very low dimension or sparsity, and may therefore be used as default loading mechanism.

5.3 Concepts for a New Model Compiler

Fundamental prerequisites for the efficient processing of high-dimensional models are the application of sparse storage and algorithms as well as a direct interface to the simulator kernel. As the previous section has demonstrated, the best performance that could be achieved under optimal conditions with the standard modeling interface (TML) of *Titan* would still be of at least a factor of 6.66 slower than a netlist-based simulation of equal dimension. Hence, a more efficient and specialized approach to achieve the targeted simulation efficiency was indispensable. Main objectives were the reduction of the overhead resulting from the generality of the TML compiler and a more direct interface to the simulator kernel. Therefore, a specialized model compiler to generate high-performance models for the so-called Z-Element Model Specification (ZMS) was developed. This new C-based interface was integrated into *Titan* to provide a modern and flexible interface as a basis for future device models.

In order to be able to generate highly specialized compiled models for the analytical behavioral models and to avoid any restrictions of the AHDLs, code generation was realized within *Analog Insydes*. Here, a system of DAEs processed within the modeling tool can be exported directly to a C-model for *Titan*, compiled by a standard C-compiler, and subsequently dynamically linked with the simulator kernel. Figure 5.7 shows the architecture of the model generation and compilation. The interaction between the model and the simulator kernel is realized through a shared memory.

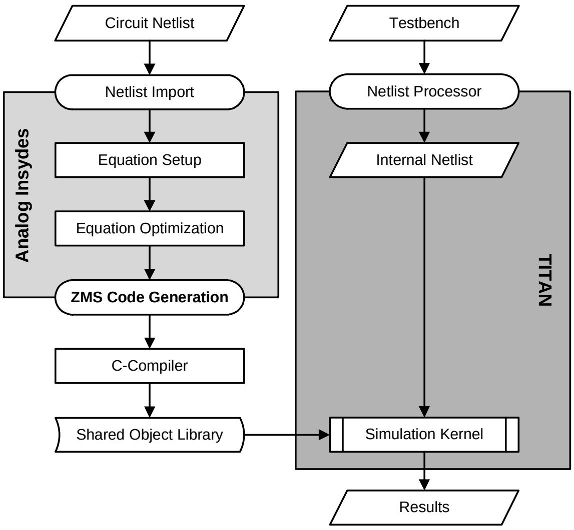


Figure 5.7: Architecture for the Model Compilation (ZMS)

As the interface was still in an early phase of development (outside the scope of this work), some features of *Titan* (like the *Titan* solver) have not yet been adapted to be compliant with ZMS. Therefore, all simulations of compiled models had to be performed with the *MUMPS* or *LAPACK* solver (instead of the more efficient *Titan* solver). Unfortunately, this reduced the performance of the solving process. This limitation is only a matter of development effort. Hence, another major improvement of the linear solver's performance can be expected by using the *Titan* solver. As the loading process (which is in the responsibility of the model compiler) does not interfere with the *Titan* solver, the presented results can easily be adapted to the most performant solver.

Within the following sections the key features of the new compiler [99, 102] and the generated models will be discussed. Figure 5.8 shows a summary of the realized features with basic requirements at the bottom of the pyramid and specialized processings on the top. The basis for all models compilers is to be at least capable of handling simultaneous DAE systems. In order to enhance convergence, limiting and damping methods as well as initial values are well-known methods. Tolerances are essential for determining the accuracy of the model, but are also related to convergence and simulation performance. A sequential handling was introduced in order to reduce dimension and simulation time [101]. Finally, several optimization strategies aim at increasing the simulation efficiency. In Chapter 6, such strategies and algorithms to optimize the equations before generating the C-based model will be introduced and applied in combination with the compiled model generation.

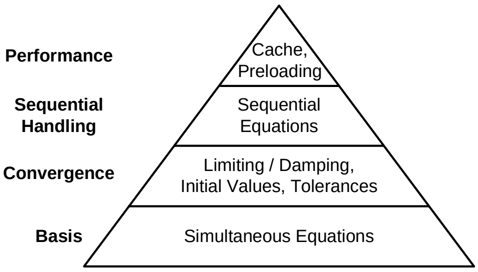


Figure 5.8: Feature Pyramid for ZMS Compiler

5.4 Compiling Simultaneous DAEs

Basically, all compiled models have to perform very similar process steps, the main difference being the efficiency of their realization. The models' tasks are to

- provide structural information (connectivity, topology, matrix structure, equation and variable types, etc.),
- handle parameters (default values, parameter changes, range checks),
- provide initial values for the variables,
- perform numerical integration,
- provide and update the Jacobian matrix and the residual,
- prevent and handle numerical problems (floating point exceptions),
- define options and tolerances for the solver (natures),
- check for convergence.

During the initialization, the model provides the simulator kernel with structural information. This information is used to manage common data structures, to process netlist entries, and to perform topology checks. Parameter settings from the netlist to configure the model are stored within the shared memory. Furthermore, the model provides the simulator kernel with initial values for the variables. All other process steps have to be performed within each iteration. The most important and time-consuming task is the evaluation of the Jacobian matrix and the residual for Newton's method.

Titan solves a system of DAEs by iteratively applying

$$\boldsymbol{J} \cdot \Delta\boldsymbol{x} = (\boldsymbol{J}_{stat} + \alpha\boldsymbol{J}_{dyn}) \cdot \Delta\boldsymbol{x} = -\boldsymbol{f}(\boldsymbol{x})\,. \tag{5.1}$$

The static and dynamic Jacobian matrices are determined within the model compilation by calculating the partial derivatives of the DAE system:

$$\boldsymbol{J}_{stat} = \begin{bmatrix} \dfrac{\partial \boldsymbol{f}_{diff}}{\partial \boldsymbol{x}_{diff}} & \dfrac{\partial \boldsymbol{f}_{diff}}{\partial \boldsymbol{x}_{algebr}} \\ \dfrac{\partial \boldsymbol{f}_{algebr}}{\partial \boldsymbol{x}_{diff}} & \dfrac{\partial \boldsymbol{f}_{algebr}}{\partial \boldsymbol{x}_{algebr}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{J}_{stat,\,11} & \boldsymbol{J}_{stat,\,12} \\ \boldsymbol{J}_{stat,\,21} & \boldsymbol{J}_{stat,\,22} \end{bmatrix} \tag{5.2a}$$

$$\boldsymbol{J}_{dyn} = \begin{bmatrix} \dfrac{\partial \boldsymbol{f}_{diff}}{\partial \dot{\boldsymbol{x}}_{diff}} & \dfrac{\partial \boldsymbol{f}_{diff}}{\partial \dot{\boldsymbol{x}}_{algebr}} \\ \dfrac{\partial \boldsymbol{f}_{algebr}}{\partial \dot{\boldsymbol{x}}_{diff}} & \dfrac{\partial \boldsymbol{f}_{algebr}}{\partial \dot{\boldsymbol{x}}_{algebr}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{J}_{dyn,\,11} & 0 \\ 0 & 0 \end{bmatrix} \tag{5.2b}$$

As the compiler is based on the *Mathematica* algebra system, the symbolic derivation is available without the additional effort of implementing automatic derivation methods. The resulting matrices are determined by evaluating the symbolic matrices with the actual solution vector. They are stored in coordinate storage form within the shared memory. The residual calculation requires the evaluation of the DAE system with the actual Newton solution. Prior to that, the necessary dynamic values are determined through numerical integration, which is performed via a call to the simulator kernel's numerical integration function. Time step length, integration method, and storage of previous values of differential variables are handled by the simulator kernel.

5.5 Compiling Sequential DAEs

In common circuit simulators, the majority of equations contained in a device model is solved internally in the device model in order to compose a compact stamp that is inserted into the simulator's Jacobian matrix. The model-internal equations are presolved in a procedural manner without applying iterative methods. The main intention is to keep the dimension of the linear system as low as possible. The concept of sequential equations is used to preprocess behavioral models in a similar way and thereby reduce the dimension of the linear system.

Modeling DAE systems with sequential structure provides an effective measure to improve performance and robustness of the model (compared to a fully simultaneous processing). The explicit formulation and procedural order of the sequential subsystem provide the means to solve the sequential equations directly and locally from the previous solution of the simultaneous variables. Thus, the sequential variables are less sensitive to numerical problems, they do not have to be solved by a linear solver, and they do not cause a residual. As soon as the sequential variables are solved, the residual for the simultaneous equations can easily be determined by using the solution vector of the sequential variables. The conventional Newton method is applied in order to solve the simultaneous equations. Thus, the reduced dimension of the linear system enhances the solver's performance.

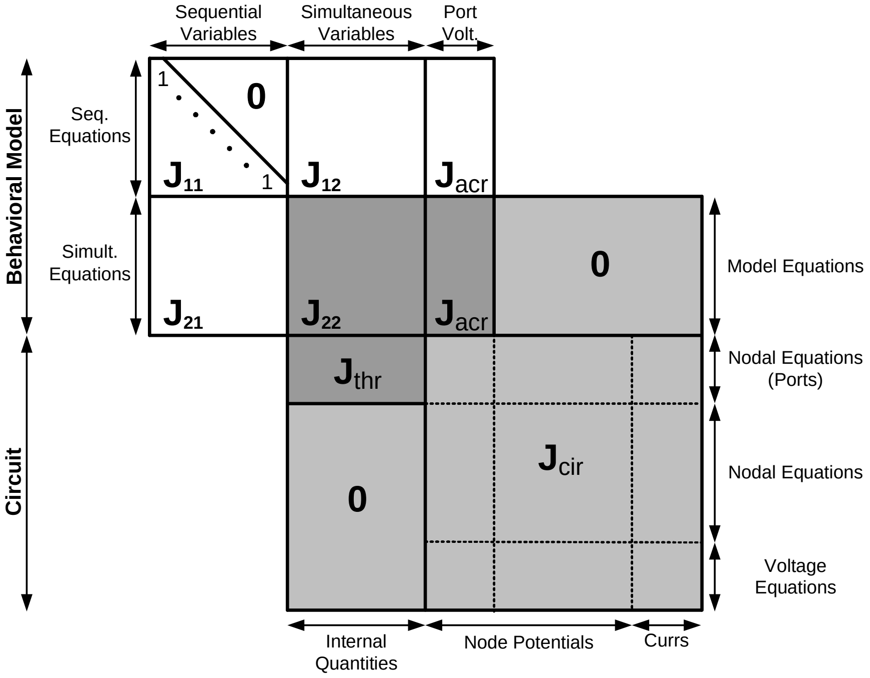


Figure 5.9: Matrix Structure of a Sequential DAE System for *Titan*

As a consequence of the approach to solve the sequential equations locally, changes in Newton’s method are necessary: applying the chain rule during the determination of the Jacobian matrix for the simultaneous equations in particular is time-consuming. As the structure of the Jacobian matrix is known to have a lower diagonal block with unit elements at the main diagonal (due to the sequential equation structure), this beneficial structure was used to achieve an efficient processing of the chain rule.

Figure 5.9 visualizes the structure of the Jacobian matrix for a *Titan* simulation including a model of sequential DAE structure. The gray parts of the matrix show the resulting Jacobian matrix that will be used within the linear solver whereas the white part will be locally solved within the model. The dark gray area represents the stamp of the behavioral model. The submatrix $\boldsymbol{J}_{cir}$ results from the netlist elements of the testbench and is set up by direct inspection from device models. The model equations (sequential as well as simultaneous ones) result in the submatrix $\boldsymbol{J}$ where $\boldsymbol{J}_{11}$ is of lower diagonal structure. Finally, the submatrices $\boldsymbol{J}_{acr}$ (node potentials of model ports) and $\boldsymbol{J}_{thr}$ (port currents contributing to nodal equations) represent the connectivity between both subsystems.

Subsequently, a sequential Newton’s method is described that takes advantage of the sequential structure of a DAE system. Let $\boldsymbol{f}(\boldsymbol{y}, \boldsymbol{x}, t) = \boldsymbol{0}$ be a DAE system with sequential structure. The vector of unknowns consists of the sequential variables $\boldsymbol{y}$ and the simultaneous variables $\boldsymbol{x}$. Setting up the linearized system yields the following equation system:

$$\boldsymbol{J} \cdot \Delta \begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{x} \end{bmatrix} = \begin{bmatrix} \boldsymbol{J}_{11} & \boldsymbol{J}_{12} \\ \boldsymbol{J}_{21} & \boldsymbol{J}_{22} \end{bmatrix} \cdot \Delta \begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{x} \end{bmatrix} = -\begin{bmatrix} \boldsymbol{f}_{seq}(\boldsymbol{y}, \boldsymbol{x}) \\ \boldsymbol{f}_{sim}(\boldsymbol{y}, \boldsymbol{x}) \end{bmatrix} \tag{5.3}$$

Due to the lower diagonal structure of $\boldsymbol{J}_{11}$, it is possible to solve the sequential subsystem for $\Delta \boldsymbol{y}$ in an explicit form ($\boldsymbol{J}_{11}$ is regular and can be inverted efficiently due to its diagonal structure). The Newton correction for the sequential variables yields

$$\Delta \boldsymbol{y} = -\boldsymbol{J}_{11}^{-1}(\boldsymbol{f}_{seq}(\boldsymbol{y}, \boldsymbol{x}) + \boldsymbol{J}_{12}\Delta \boldsymbol{x}) . \tag{5.4}$$

By substituting $\Delta \boldsymbol{y}$ into the simultaneous subsystem a linear system of reduced dimension is achieved. It has to be solved for the Newton correction of the simultaneous variables $\Delta \boldsymbol{x}$ *:*

$$-\boldsymbol{J}_{21}\boldsymbol{J}_{11}^{-1}(\boldsymbol{f}_{seq}(\boldsymbol{y}, \boldsymbol{x}) + \boldsymbol{J}_{12}\Delta \boldsymbol{x}) + \boldsymbol{J}_{22}\Delta \boldsymbol{x} = -\boldsymbol{f}_{sim}(\boldsymbol{y}, \boldsymbol{x}) \tag{5.5a}$$

$$(\boldsymbol{J}_{22} - \boldsymbol{J}_{21}\boldsymbol{J}_{11}^{-1}\boldsymbol{J}_{12})\Delta \boldsymbol{x} = -\boldsymbol{f}_{sim}(\boldsymbol{y}, \boldsymbol{x}) + \boldsymbol{J}_{21}\boldsymbol{J}_{11}^{-1}\boldsymbol{f}_{seq}(\boldsymbol{y}, \boldsymbol{x}) \tag{5.5b}$$

As the sequential variables $\boldsymbol{y}$ can be determined through direct solution of the sequential equations with the previous solution vector of $\boldsymbol{x}$ through

$$y_i = f_{seq,\, i}(y_1 \ldots y_{i-1}, \boldsymbol{x}) \quad \text{for} \quad i = 1 \ldots n \ , \tag{5.6}$$

their residual $\boldsymbol{f}_{seq}(\boldsymbol{y}, \boldsymbol{x})$ is zero. Hence the residual of the reduced system can easily be determined with the knowledge of $\boldsymbol{y}$. Consequently, Newton's method has to be applied to solve

$$\boldsymbol{J}'\Delta \boldsymbol{x} = -\boldsymbol{f}_{sim}(\boldsymbol{x}) \tag{5.7}$$

with the reduced Jacobian matrix $\boldsymbol{J}'$ obtained by the following Schur complement:

$$\boldsymbol{J}' = \boldsymbol{J}_{22} - \boldsymbol{J}_{21}\boldsymbol{J}_{11}^{-1}\boldsymbol{J}_{12} \tag{5.8}$$

$\boldsymbol{J}'$ represents the Jacobian matrix for the simultaneous subsystem taking into account additional contributions to the original Jacobian matrix resulting from the evaluation of the chain rule for the sequential equations.

Function Evaluation

The function evaluation within the sequential Newton method basically requires two process steps. The evaluation of the sequential equations with the previous solution of the simultaneous variables yields the solution vector for the sequential variables. The residual for the simultaneous equations has to be determined using the previous simultaneous solution and the current solution of the sequential equations.

However, the evaluation of the sequential equations contains differential variables that require a numerical integration before the equations can be processed. The evaluation of the (simultaneous) residual might require another numerical integration. Calling the integration function twice within one iteration implies the risk of implicit second order derivatives that would result in a major inconsistency of Newton’s method. In order to prevent this problem, the sequential subsystem itself is partitioned into a static and a dynamic subsystem. Definition 5.1 provides the algorithm for the partitioning.

Definition 5.1 - Partitioning Sequential Variables into Algebraic/Differential Variables

As stated in Definition 2.3, the sequential variables $\boldsymbol{y}$ *of a first-order DAE system with sequential structure can be partitioned into algebraic (*$\boldsymbol{y}_{algebr}$*) and differential sequential variables (*$\boldsymbol{y}_{diff}$*) and corresponding equations. The algorithm is initialized by* $\boldsymbol{y}_{diff} = \varnothing$ *. By applying*

$$\boldsymbol{y}_{algebr} = \boldsymbol{y} \cap \boldsymbol{y}_{diff}$$

$$\boldsymbol{y}_{diff} = \{\boldsymbol{y}_{algebr} | \boldsymbol{f}_{seq}(\boldsymbol{y}_{diff}, \boldsymbol{x}, \dot{\boldsymbol{x}})\}$$

in a fixed point iteration, the required partitioning for the sequential variables is obtained. All sequential variables that are defined by a sequential equation containing at least one differential simultaneous variable or a previously defined differential sequential variable are handled as differential sequential variables. As each sequential variable corresponds to a sequential equation, the partitioning for the equations is implicitly achieved.

■

The partitioning ensures that an expression does not contain more than one differentiation for each variable, either within the sequential or the simultaneous equations. The function evaluation is performed as shown below:

$$\boldsymbol{y}_{algebr} = \boldsymbol{f}_{seq,\,algebr}(\boldsymbol{y}_{algebr}, \boldsymbol{x}) \tag{5.9a}$$

$$(\dot{\boldsymbol{y}}_{algebr}, \dot{\boldsymbol{x}}) \approx NumericalIntegration(\boldsymbol{y}_{algebr}, \boldsymbol{x}) \tag{5.9b}$$

$$\boldsymbol{y}_{diff} = \boldsymbol{f}_{seq,\,diff}(\boldsymbol{y}_{algebr}, \dot{\boldsymbol{y}}_{algebr}, \boldsymbol{y}_{diff}, \boldsymbol{x}, \dot{\boldsymbol{x}}) \tag{5.9c}$$

$$RHS = -\boldsymbol{f}_{sim}(\boldsymbol{y}_{algebr}, \dot{\boldsymbol{y}}_{algebr}, \boldsymbol{y}_{diff}, \boldsymbol{x}, \dot{\boldsymbol{x}}) \tag{5.9d}$$

The numerical integration is now required only once and second order derivatives are impossible to achieve. The residual for solving the simultaneous equations using the linear solver is computed using the previously calculated solution for the sequential variables.

Determination of the Reduced Jacobian Matrix

The Schur complement is the main reason for overhead when utilizing the sequential structure of the DAE system. Therefore, it has to be implemented efficiently in order to reduce the additional effort within the loading process. Its implementation will be discussed in detail as it is one of the key features of the new compilation approach.

The computation of the Schur complement makes use of so-called structural matrices (see Definition 5.2 and Figure 5.10). A structural matrix contains references by index to the structural non-zero elements of a symbolic matrix. It will be used to simplify the processing of the Schur complement.

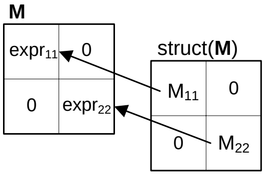


Figure 5.10: Structural Matrix

Definition 5.2 - Structural Matrix

The structural matrix $\boldsymbol{S}$ *is derived from a matrix of symbolic expressions* $\boldsymbol{M}$ *by*

$$\boldsymbol{S} = struct(\boldsymbol{M}) \ \ with$$

$$struct : \begin{cases} \boldsymbol{M}[i,j] \notin \{0\} \rightarrow [i,j] \\ \qquad\qquad 0 \rightarrow 0 \end{cases}$$

■

There are several solutions to evaluate the chain rule during the Jacobian matrix setup. The resulting $\boldsymbol{J'}$ matrix is the same for all approaches. Nevertheless, the implementation significantly determines their efficiency:

- **Symbolic Schur Complement:** All sequential variables within the simultaneous equations are substituted by their determining sequential equations. Subsequently, a fully symbolic $\boldsymbol{J'}$ matrix is set up by symbolic derivation. This approach results in an expression set of tremendous complexity with a large degree of redundancy. The complexity of the expressions alone disables this procedure even for low dimensional DAE systems.
- **Numerical Schur Complement:** The full Jacobian matrix $\boldsymbol{J}$ is set up by symbolic derivation during the model compilation (as for both following methods). After evaluating this matrix numerically within each iteration, the reduced Jacobian $\boldsymbol{J'}$ is calculated from Equation (5.8). This approach requires one matrix inversion and two matrix multiplications without utilizing the structural knowledge of the problem (high sparsity, lower diagonal structure).
- **Semi-Symbolic Schur Complement:** This method combines the strengths of both numerical and symbolic processing. Therefore, the structural Jacobian matrix $\boldsymbol{S} = struct(\boldsymbol{J})$ is set up and the Schur complement is calculated symbolically from $\boldsymbol{S}$ to derive $\boldsymbol{S'}$. Finally, evaluating $\boldsymbol{J}$ and $\boldsymbol{S'}$ results in the reduced Jacobian matrix $\boldsymbol{J'}$. Hence, the Schur complement takes advantage of structural zeros within $\boldsymbol{J}$. Unfortunately, the necessary symbolic matrix operations still result in an extraordinary high complexity (disabling the approach even for medium size matrices).

- **Semi-Symbolic Schur Complement by an Elimination Method:** In order to reduce the complexity of the previous approach, a method with low redundancy (no repetition of performed calculations) and maximal usage of structural properties of the matrices (use of structural zeros and ones) is proposed. Instead of calculating the Schur complement based on $\boldsymbol{S}$, the submatrix $\boldsymbol{S}_{21} = struct(\boldsymbol{J}_{21})$ is eliminated with the diagonal elements of submatrix $\boldsymbol{S}_{11} = struct(\boldsymbol{J}_{11})$ that are structurally one. The performed elimination steps are "recorded" and represent a procedurally evaluated transformation to calculate $\boldsymbol{S}'$ from the submatrices of $\boldsymbol{S}$. Thus, the high complexity of a fully symbolic $\boldsymbol{S}'$ matrix can be avoided. An advantage of the transformation process is that it contains a large number of very simple expressions (instead of a small number of highly complex expressions for the previous method).

Due to its advantageous properties, the latter approach was chosen. For simplicity, the separation of $\boldsymbol{J}$ into static and dynamic portions has been neglected so far. In fact, the Jacobian matrix results from:

$$\boldsymbol{J} = \boldsymbol{J}_{stat} + \alpha\boldsymbol{J}_{dyn} = \begin{bmatrix} \frac{\partial \boldsymbol{f}_{seq}}{\partial \boldsymbol{x}_{seq}} & \frac{\partial \boldsymbol{f}_{seq}}{\partial \boldsymbol{x}_{sim}} \\ \frac{\partial \boldsymbol{f}_{sim}}{\partial \boldsymbol{x}_{seq}} & \frac{\partial \boldsymbol{f}_{sim}}{\partial \boldsymbol{x}_{sim}} \end{bmatrix} + \alpha \begin{bmatrix} \frac{\partial \boldsymbol{f}_{seq}}{\partial \dot{\boldsymbol{x}}_{seq}} & \frac{\partial \boldsymbol{f}_{seq}}{\partial \dot{\boldsymbol{x}}_{sim}} \\ \frac{\partial \boldsymbol{f}_{sim}}{\partial \dot{\boldsymbol{x}}_{seq}} & \frac{\partial \boldsymbol{f}_{sim}}{\partial \dot{\boldsymbol{x}}_{sim}} \end{bmatrix} \tag{5.10}$$

The Schur complement has to be computed for the complete Jacobian matrix $\boldsymbol{J}$, but the resulting model is supposed to return $\boldsymbol{J}_{stat}$ and $\boldsymbol{J}_{dyn}$ separately. Therefore, the structural matrices are set up for both the static and dynamic Jacobian matrices:

$$\boldsymbol{S}_{stat} = struct(\boldsymbol{J}_{stat}) \tag{5.11a}$$

$$\boldsymbol{S}_{dyn} = struct(\boldsymbol{J}_{dyn}) \tag{5.11b}$$

Before processing the Schur complement, the structural matrix $\boldsymbol{S}$ is initialized by

$$\boldsymbol{S} = \boldsymbol{S}_{stat} + \alpha\boldsymbol{S}_{dyn} \tag{5.12}$$

with the symbol α representing the integration variable. After calculating the Schur complement

$$\boldsymbol{S}' = \boldsymbol{S}_{22} - \boldsymbol{S}_{21}\boldsymbol{S}_{11}^{-1}\boldsymbol{S}_{12}, \tag{5.13}$$

the resulting matrix is partitioned into static and dynamic contributions as follows:

$$\boldsymbol{S}'_{stat} = \boldsymbol{S}'|_{\alpha = 0} \tag{5.14a}$$

$$\boldsymbol{S}'_{dyn} = \boldsymbol{S}'|_{\alpha = 1} - \boldsymbol{S}'_{stat} \tag{5.14b}$$

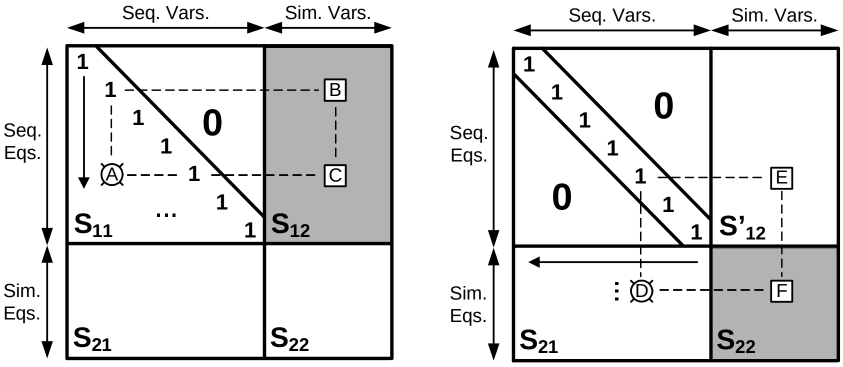


Figure 5.11: Stage 1 (left) and Stage 2 (right) of the Elimination Process

The Schur complement is performed by a two-stage elimination process. Within the first step, the submatrix $\boldsymbol{S}_{11}$ is eliminated columnwise using the diagonal elements to achieve an identity matrix. The elimination starts with the first column. During the processing, the submatrix $\boldsymbol{S}_{12}$ is gradually changed to $\boldsymbol{S}'_{12}$. Figure 5.11 (left) exemplarily shows a single step of the proposed elimination method. In order to eliminate the entry A, an elimination step is added to the transformation rules:

$$C = C - AB \tag{5.15}$$

During the elimination, fill-ins (former structural zero elements) are generated within the submatrix $\boldsymbol{S}_{12}$. Within a second stage, the entries of the submatrix $\boldsymbol{S}_{21}$ are eliminated using the derived identity matrix. Figure 5.11 (right) shows the elimination scheme for this stage. The elimination is performed row wise and results in the desired chain rule contributions within submatrix $\boldsymbol{S}_{22}$. The fill-ins resulting from this elimination process have to be taken into account to initialize data structures correctly and to provide consistent structural information to the simulator kernel. Finally, the gray part of the matrix in Figure 5.11 (right) is provided as reduced Jacobian matrix. The elimination typically results in a large number of transformation steps to perform. Still, a single elimination step is of very limited complexity.

Example 5.1: Schur Complement for Jacobian Matrix of the Foucault Pendulum

This example demonstrates the different methods to calculate the Schur complement. As the dimension of the Foucault pendulum's DAE system is pretty low, the example visualizes the methods but does not make the complexity problems visible. The Jacobian matrix (5.16) was previously derived in Example 3.1 (Page 33).

The numerical values within the example are determined by calculating the first Newton iteration of the DC analysis. The used values are:

`y1 → 0., y2 → -0.4905, y3 → 0., x1 → 0.5, x2 → 0., x3 → 0., x4 → 0.` (initial values)

$\mathtt{g} \to 9.81,\ \mathtt{l} \to 10,\ \mathtt{w} \to 0.1,\ \mathtt{lam} \to \frac{\pi}{4}$ (design point, parameter values)

$$\left(\begin{array}{ccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & -2\,\mathtt{w} \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -\frac{\mathtt{Sin}[\mathtt{lam}]}{\mathtt{l}} & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline -\mathtt{Cos}[\mathtt{lam}] & 1 & -\mathtt{x1}[\mathtt{t}] & \frac{\mathtt{g}}{\mathtt{l}} - \mathtt{y3}[\mathtt{t}] & 0 & 0 & 0 \\ 0 & 0 & -\mathtt{x2}[\mathtt{t}] & 0 & \frac{\mathtt{g}}{\mathtt{l}} - \mathtt{y3}[\mathtt{t}] & 2\,\mathtt{w}\,\mathtt{Cos}[\mathtt{lam}] & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right) \tag{5.16}$$

Symbolic Schur Complement

For the symbolic Schur complement, the linearized system is determined directly from the fully simultaneous system. This equation system is achieved by substituting all occurrences of a sequential variable by its symbolic definition within the sequential equation. Afterwards, the resulting simultaneous DAE system is derived for its variables to generate the static Jacobian matrix (5.17).

$$\begin{pmatrix} \frac{\mathtt{g}}{\mathtt{l}} - \frac{2\mathtt{w}\mathtt{Sin}[\mathtt{lam}]\,\mathtt{x4}[\mathtt{t}]}{\mathtt{l}} & 0 & 0 & -2\,\mathtt{w}\,\mathtt{Cos}[\mathtt{lam}] - \frac{2\mathtt{w}\mathtt{Sin}[\mathtt{lam}]\,\mathtt{x1}[\mathtt{t}]}{\mathtt{l}} \\ 0 & \frac{\mathtt{g}}{\mathtt{l}} - \frac{2\mathtt{w}\mathtt{Sin}[\mathtt{lam}]\,\mathtt{x4}[\mathtt{t}]}{\mathtt{l}} & 2\,\mathtt{w}\,\mathtt{Cos}[\mathtt{lam}] & -\frac{2\mathtt{w}\mathtt{Sin}[\mathtt{lam}]\,\mathtt{x2}[\mathtt{t}]}{\mathtt{l}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.17}$$

Compared to the Jacobian matrix of the sequential DAE system (5.16), (5.17) contains a considerable amount of redundant expressions. This effect becomes more and more important with an increasing dimension of the DAE system. In order to determine the numerical result, the reduced symbolic Jacobian matrix (5.17) is evaluated with the numerical values resulting in (5.18).

$$\begin{pmatrix} 0.981 & 0. & 0. & -0.148492 \\ 0. & 0.981 & 0.141421 & 0. \\ 0. & 0. & 1. & 0. \\ 0. & 0. & 0. & 1. \end{pmatrix} \tag{5.18}$$

Numerical Schur Complement

For the numerical method, (5.16) is evaluated with the numerical values before calculating the Schur complement. The resulting Jacobian matrix (5.19) is of block lower-triangular structure as the first three equations are sequential equations.

$$\left(\begin{array}{ccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & -0.2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{10\sqrt{2}} & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline -\frac{1}{\sqrt{2}} & 1 & -0.5 & 0.981 & 0 & 0 & 0 \\ 0 & 0 & 0. & 0 & 0.981 & 0.141421 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right) \tag{5.19}$$

Subsequently, the Schur complement is calculated from the submatrices of (5.19):

$$\begin{pmatrix} 0.981 & 0 & 0 & 0 \\ 0 & 0.981 & 0.141421 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} - \begin{pmatrix} -\frac{1}{\sqrt{2}} & 1 & -0.5 \\ 0 & 0 & 0. \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{10\sqrt{2}} & 0 & 1 \end{pmatrix}^{-1} \cdot \begin{pmatrix} 0 & 0 & 0 & -0.2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0.981 & 0. & 0. & -0.148492 \\ 0. & 0.981 & 0.141421 & 0. \\ 0. & 0. & 1. & 0. \\ 0. & 0. & 0. & 1. \end{pmatrix} \tag{5.20}$$

The resulting Jacobian (5.20) is the same again. For matrices of higher dimension, the matrix inverse as well as the matrix multiplications are problematic in terms of computational effort.

Semi-Symbolic Schur Complement

Combining both numerical and symbolic approaches, the semi-symbolic Schur complement is based on the structural matrix of the symbolic Jacobian matrix (5.16) of the sequential DAE system. The structural matrix (5.21) consists of references to the numerical values that have previously been calculated as shown in (5.19). For better readability, the indexing has been printed using standard matrix indices. In fact, the matrices are stored in COO matrix format containing only the non-zero entries of the matrix.

$$\left(\begin{array}{ccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & G[1, 7] \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ G[3, 1] & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline G[4, 1] & G[4, 2] & G[4, 3] & G[4, 4] & 0 & 0 & 0 \\ 0 & 0 & G[5, 3] & 0 & G[5, 5] & G[5, 6] & 0 \\ 0 & 0 & 0 & 0 & 0 & G[6, 6] & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & G[7, 7] \end{array}\right) \tag{5.21}$$

In this method, the Schur complement is calculated by symbolically processing the submatrices of (5.21):

$$\begin{pmatrix} G[4, 4] & 0 & 0 & 0 \\ 0 & G[5, 5] & G[5, 6] & 0 \\ 0 & 0 & G[6, 6] & 0 \\ 0 & 0 & 0 & G[7, 7] \end{pmatrix} -$$

$$\begin{pmatrix} G[4, 1] & G[4, 2] & G[4, 3] \\ 0 & 0 & G[5, 3] \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ G[3, 1] & 0 & 1 \end{pmatrix}^{-1} \cdot \begin{pmatrix} 0 & 0 & 0 & G[1, 7] \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} G[4, 4] & 0 & 0 & -G[1, 7]\,(G[4, 1] - G[3, 1]\,G[4, 3]) \\ 0 & G[5, 5] & G[5, 6] & G[1, 7]\,G[3, 1]\,G[5, 3] \\ 0 & 0 & G[6, 6] & 0 \\ 0 & 0 & 0 & G[7, 7] \end{pmatrix} \tag{5.22}$$

The resulting Schur complement of the structural matrix (5.22) is subsequently evaluated using the numerical values of (5.21), and leads to the same result. The entries within the last column insinuate the growing complexity of the expressions within this method. Furthermore, the multiplication of G[1,7] and G[3,1] is performed twice.

Semi-Symbolic Schur Complement by Elimination

Finally, the preferred method presents the semi-symbolic Schur complement by elimination. The starting point is the structural matrix (5.21). The processing aims at transforming it into a new matrix with the lower-left submatrix being eliminated to zeros. Afterwards, the transformed lower-right submatrix is evaluated with the numerical values in order to achieve the numerical Jacobian matrix of the simultaneous subsystem.

(5.23) shows the sequence of elimination steps that are necessary to transform the matrix. At first, the entry G[3,1] is eliminated in order to achieve a unity matrix within the upper-left submatrix. Afterwards, the entries of the lower-left matrix are eliminated resulting in fill-ins in G[4,7] and G[5,7].

$$
\begin{aligned}
&G[3, 7] == -G[1, 7]\, G[3, 1] \\
&G[4, 7] == -G[1, 7]\, G[4, 1] \\
&G[4, 7] == -G[3, 7]\, G[4, 3] + G[4, 7] \\
&G[5, 7] == -G[3, 7]\, G[5, 3]
\end{aligned}
\tag{5.23}
$$

As it can be seen within the elimination sequence, the multiplication of G[1,7] and G[3,1] is now only performed once to determine G[3,7]. Subsequently, G[3,7] is used twice to determine other matrix entries. Thus, the chain rule was evaluated with very low effort.

■

Supplementary Performance Enhancements

Further enhancements to the performance of the Schur complement realization have been achieved through strategies known from compiler design: constant propagation, constant folding, and pre-evaluation of loop-invariant expressions (see [1] for details on compiler design). Since these optimizations have already been achieved on a relatively high level of abstraction (instead of relying on the C-compilations optimization), even structural changes within the matrices can be taken into account.

Constant propagation identifies constants assigned to a variable and removes the corresponding variable by replacing all occurrences of the variable with the constant’s value (saving evaluation time and memory). As the Jacobian matrix $\boldsymbol{J}$ contains a relatively large number of constant non-zero entries (especially ones) and simple expressions containing only parameters (e.g. $1/R$), these entries have been recursively propagated into the transformation rules of the Schur complement and the $\boldsymbol{J}_{stat}$ or $\boldsymbol{J}_{dyn}$ matrices. Combined with constant folding (pre-evaluation of expressions containing only constants) the Schur complement can be simplified efficiently.

Afterwards, the pre-evaluation of loop-invariant expressions saves unnecessary frequent evaluations of expressions within a loop. This strategy is adapted as so-called preloading within simulators. The constant entries of the reduced Jacobian matrices $\boldsymbol{J}'_{stat}$ and $\boldsymbol{J}'_{dyn}$ are loaded only once within the initialization of the model instead of being processed repeatedly within each iteration. Furthermore, there is typically a remarkable amount of sequential equations which can also be pre-evaluated instead of being recalculated within each iteration. These sequential equations that contain solely constants and parameters are only evaluated within the initial iteration of each analysis. Thus, the computational effort within each iteration is reduced.

As the expression evaluation for simultaneous equations and the reduced Jacobian matrices do not contain any interdependencies, they can be evaluated in any order. Hence, the order of the data within the memory should be optimally adapted to the expression evaluation with regard to cache effects. Storing data within arrays restrains the C-compiler from optimizing the memory allocation with respect to expression evaluation and cache effects. In some cases, an array structure is inevitable (storage of vectors and matrices within the shared memory). Local data (e.g. the internal Jacobian matrix) is preferably held in separate variables although the matrix structure would typically advise an array storage to achieve optimal results from C-compiler optimizations.

5.6 Improving Convergence

Tolerances

Tolerances are of fundamental importance to the accuracy and the convergence of Newton's method since they determine the level of accuracy to which the linearized systems are solved. During simulation, tolerances for equations and variables are used to check for convergence (refer to Section 3.1). In netlist-based simulations, the tolerances can relatively easily be specified, since the linear system typically consists of nodal equations (sums of currents) and a few voltage equations (voltage sources, inductors) only. For behavioral models, the tolerances are specified by the user and should match the physical meaning of a variable or equation (e.g. charge, force, temperature).

Determining tolerances in general DAEs without prior knowledge of the physical meaning of variables is not possible without loss of generality. For future applications, the *Analog Insydes* based modeling flow should be extended by a general strategy to keep track of tolerances for all contained symbols and equations. As the DAEs are set up from a topology and device models, the physical meaning can be provided by the device models and stored within an additional data structure. This information has to be managed and updated whenever an equation is subject to algebraic processing. Therefore, reformulations, simplifications carried out by model reduction, substitutions, and many other processings do not only have to change the DAE system but they also have to update the information on tolerances of the equations.

Without keeping this additional physical information from device models, the proposed consequent handling of tolerances is not yet available. Compromises and heuristics have to be applied to determine reasonable accuracy settings. Hence, internal naming conventions for the variables in *Analog Insydes* have been used to (heuristically) determine the physical meaning of variables.

Table 5.1: Tolerances used within Compiled Models

Tolerance	***absolute***	***relative***
Residual	10^{-6}	10^{-5}
Variables	10^{-6}	10^{-5}
Differential Variables	10^{-2}	10^{-5}

Table 5.1 summarizes the tolerance settings that have been used for the C-based models. Due to the general nature of the problem (to find reasonable tolerances for a general DAE system) these settings are a good trade-off.

Initial Values

Inappropriate initial values of the variables are a common source of convergence problems in DC analysis. As Newton's method converges better in proximity of the solution, a good and consistent initial value is of major importance for DC analysis. Possible problems in DC analysis are non-convergence, numerical problems (e.g. division by zero), and convergence towards a “wrong” operating point (if multiple solutions are available). By default, simulators start from zero, if no user-specified initial value (e.g. nodeset) is available. Especially for equation sets that are not (manually) optimized for convergence, thus not internally preventing numerical problems (e.g. by case differentiation), bad initial values are a serious issue.

The numerical problems result from nonlinear operations causing floating point exceptions during the evaluation (e.g. division by zero, infinity, not a number). Unfortunately, once a single variable is assigned a floating exception value, the exceptions propagate through the equation system causing irrecoverable inconsistencies. Therefore, it is essential to avoid any floating exceptions (e.g. by limiting functions, damping, initial values). A special case of such floating exceptions is division by zero. It primarily happens within the first DC iteration (due to variables initialized with zero) and can not be handled by limiting functions. Damping strategies are not suitable either as a workaround of this problem, as they depend on a previous successful solution, which is not applicable within the first DC iteration. In any subsequent iteration, division by zero is most unlikely to be a problem since variables are very unlikely to reach an exact value of zero within Newton's method[1].

Consistent initial values for the internal variables of a model provide an effective measure to avoid this problem. They prevent numerical problems and support convergence by starting the iteration close to the expected solution. An additional problem that occurs with DC convergence in behavioral simulation is the low effectiveness of homotopy methods for large behavioral models. As most homotopy methods are based on topological information (e.g.

---

[1] Zero values during the iterative solution might result from conditional statements or functions like e.g. max(), min(), sign(), unitstep(). These functions should be used with care within models and (if possible) return a value close to zero instead of an exact zero.

slowly ramping up sources, inserting capacitors at nodes) they have limited effect on behavioral models. The methods can only influence the ports of a model but not the internal equation set.

Within the model compiler consistent initial values are determined by using the DC operating point available within *Analog Insydes*. Thus, initial values for most variables are available and are applied within the model initialization. These initial values are a DC solution to the simulation, provided that the testbench the model is integrated in is the same as for the modeling process. If this precondition is not met, the values provide at least some basic consistency between the variables. Any variable without an initial value is initialized with a positive random number close to zero instead of exactly zero. The random values also avoid zeros resulting from differences of variables that were initialized with the same value (e.g. $(V_1 - V_2)^{-1}$).

Limiting Functions

The intention of limiting functions is to prevent floating point exceptions by local correction of functions that could cause such problems. The main objective is to extend the domain of nonlinear functions to the complete range of possible arguments in order to prevent domain exceptions (e.g. square root or logarithm of negative numbers). Furthermore, the co-domain of the limited functions should not exceed the possible range of floating point numbers. Thus, most of the root causes for floating exceptions within the model evaluation can be prevented.

The original functions are replaced by the limiting functions. They return limited values when being used outside of their area of validity. The limited value must not be accepted as a solution for the function. The intention is rather to overcome short-term problems and to force the limiting functions to converge back to the area of validity. In order to avoid false convergence, the model activates a limiting flag when using a limiting function outside its area of validity. This flag instructs the simulator kernel to continue the iteration until no limiting functions are active anymore.

Due to the fact that behavioral models may contain automatically generated or manually created equation sets, limiting functions for behavioral models are more important and more complicated as for device models. Within device models, limiting functions are typically applied with regard to physical knowledge of the device (e.g. pn-junction limiting), which is impossible for general models. Furthermore, device models are optimized for convergence by experts whereas behavioral models may also have been created by unexperienced users.

Important issues related to the definition of limiting functions are:

- Determining the area of validity and the bounds for the activation of the limiting function
- Determining a simple and sufficient continuation function (mostly linear)
- Creating convergence back to the area of validity
- Taking care of the continuity of the piece-wise defined limiting functions (values and, if possible, first-order derivatives)

- Limiting the derivatives of the limiting function (to keep function and derivative consistent)
- Avoiding to return exact zero numbers (also for derivatives)

Typical (double precision) floating point numbers have a range of approximately 10^{-308} to 10^{308} (depending on architecture and number format). Numbers with an absolute value smaller than 10^{-308} (except 0) cause an underflow, numbers of an absolute value greater than 10^{308} result in overflows. Apart from the theoretically possible range of floating point numbers, values of high orders of magnitude result in serious problems for linear solvers due to ill-conditioned systems. Therefore, the numerics of the problem advise to limit functions already at considerably lower orders of magnitude (e.g. 10^{30}). All nonlinear functions should cover the whole domain and have a co-domain that does not exceed the predefined maximum value. Problematic functions in terms of the domain are square root and logarithm. In terms of overflow/underflow especially exponential and power functions are critical (exp, pow).

Local damping strategies are a special case of hard-coded limiting functions. These functions adaptively determine properties (e.g. bounds, gradient, value) of the limiting functions by past values of the function's arguments. Thus, a more specific and efficient limiting with regard to the function's application is possible. As this strategy would require the storage of past values of the expressions that are used as arguments for all of the concerned functions, it is not efficiently applicable for behavioral modeling.

5.7 Results

The performance of the models generated by the new ZMS-based compiler will be presented within this section. Due to the missing integration of *Titan*'s default sparse solver for the ZMS interface, all following simulations have been performed with the *MUMPS* solver. As this solver's intention is to process systems of much higher dimension, it is of suboptimal performance for these simulations (although it is much more efficient than the default *LAPACK* solver). As previously mentioned, the ZMS interface should be enhanced to be compatible with the *Titan* solver that has shown to achieve a high solving performance (refer to Section 4.5). In order to evaluate the efficiency of the model compilation, the presented statistics and charts will focus on the CPU time for the loading process. The overall performance for future applications with the *Titan* solver can be preestimated by adding the presented loading performance and the solving performance that was achieved using the TML-compilation and the *Titan* solver.

First of all, the analyses of CPU time over model dimension for the linear networks have been repeated with the compiled models in order to check the sparse loading and to estimate the overhead between both simulation modes. Figure 5.12 shows the loading performance for the complete and the chain networks. The charts for total and solving performance can be found in Appendix C.2. The loading performance of the chain networks was efficiently improved to a ratio of only 1.2. Thus, the overhead was reduced to only 20 % (compared to a factor of 16 for the sparse loading using TML). For the complete networks, the loading is even faster than for the equivalent netlist-based simulations (due to the efficient preloading

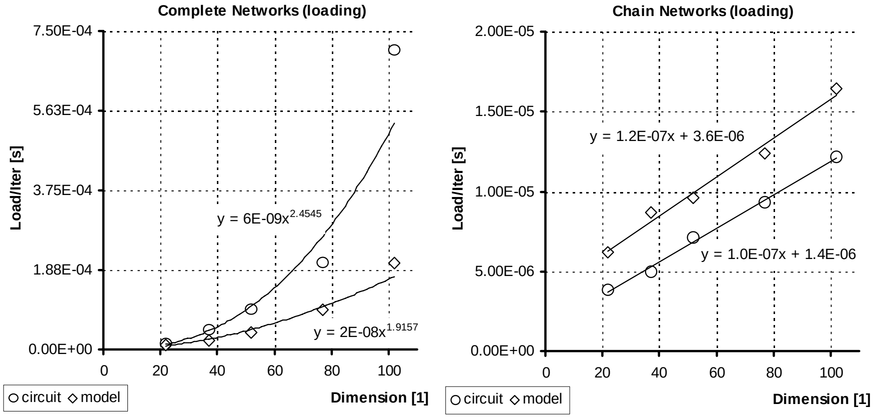


Figure 5.12: Loading Performance for Complete (left) and Chain Networks (right) for *Titan* with ZMS-Based Models

applied within the compiled models). As the network is completely linear, all entries of the Jacobian matrix can be preloaded within the model initialization. Hence, the model evaluation within each iteration merely consists of the residual calculation.

In Table 5.2, the performance of the compiled models is compared to the TML-based model compilation (already including sparse loading). As the dimension indicates, the local solving of sequential equations significantly reduces the dimension but does not achieve dimensions competitive to the netlist-based simulation. This problem will be addressed by optimization methods presented within the next chapter. The loading performance (T_{load}) could be efficiently enhanced by the ZMS compilation. The slow-down (S_{load}^{-1}) of the CPU time for the loading process demonstrates that the processing within the compiled models is only a factor of 2 to 3 slower than the circuit simulation (for the Gummel-Poon and MOS Level 1 based models).

Figure 5.13 visualizes the slow-down factors for these models compared to the TML-based simulation. The compiled models achieved major speed-ups between 3 and 12. Compared to the performance that was originally reached by TML models without sparse loading (slow-down of 210 for *opamp741*), the CPU time was improved by factors of 10 to 70. Considering the sequential structure, it is now also possible to simulate models based on the symbolic BSIM3 model. Due to the prevention of numerical problems (floating point exceptions) through the sequential equations, convergence could be achieved for the highly complex *cfcamp* and *nand2* examples. Indeed, the slow-down for these models is still considerably high (*cfcamp*: 55, *nand*: 28).

Table 5.2: Performance Measurements (ZMS Compilation)

Example	***Type***	Dim	$N_{iter/step}$	T_{load}	S_{load}^{-1}
multiplier	Circuit	28	2.0	0.046 s	1
	Seq. Model (ZMS)	47	2.95	0.159 s	3.46
	Model (ZMS)	95	2.97	0.178 s	3.87
	Model (TML)	113	2.37	0.634 s	15.85
opamp741	Circuit	58	2.25	0.178 s	1
	Seq. Model (ZMS)	137	3.05	0.502 s	2.82
	Model (ZMS)	314	3.1	0.568 s	3.19
	Model (TML)	368	3.49	5.805 s	32.61
cfcamp_mos1	Circuit	23	2.01	0.069 s	1
	Seq. Model (ZMS)	62	3.24	0.134 s	1.94
	Model (ZMS)	149	3.71	0.188 s	2.72
	Model (TML)	149	2.03	0.49 s	7.1
cfcamp	Circuit	21	2.12	0.469 s	1
	Seq. Model (ZMS)	83	2.87	26.02 s	55.48
nand2	Circuit	11	2.09	0.075 s	1
	Seq. Model (ZMS)	27	3.25	2.092 s	27.89

Comparing the results of the simultaneous and the sequential processing by the ZMS-compiler, the speed-up by applying the sequential solving method does not yet look satisfying. Chapter 6 will show, that the advantages of the sequential Newton method will have a distinct effect on the simulation performance as soon as they are used in conjunction with DAE optimization techniques.

The presented work aimed at the integration of a high-performance model compilation into the symbolic analysis toolbox. Compared to TML-based models, it has significantly enhanced the simulation performance to a level that is already quite competitive to netlist-based simulations (slow-down for *opamp741* reduced from 210 to 3). Above all, the efficient handling of the equation systems led to a major speed-up. By applying a local sequential solving method, the dimension of the linearized equation system has been reduced significantly. The realized sequential Newton method enables several optimizations of the DAE systems that would have been ineffective without this specialized processing of the equation system.

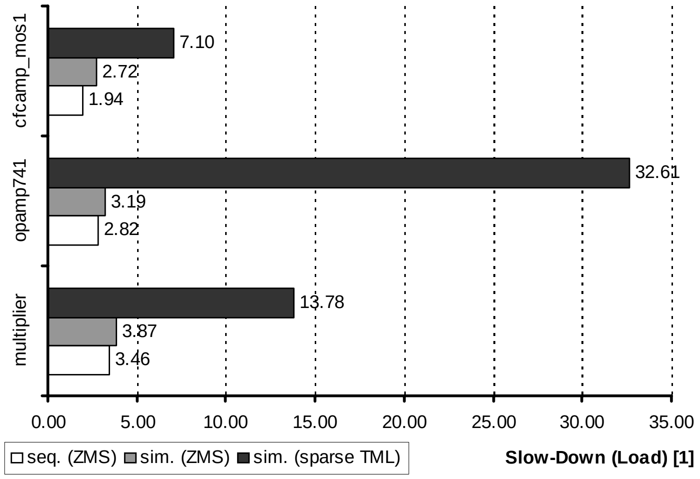


Figure 5.13: Slow-Down Factors for the Loading Process (ZMS Compilation)

6 Optimization of DAEs for Numerical Methods

As demonstrated in Chapter 4, the formulation of behavioral models determines the simulation performance to a large degree. Within this chapter, algorithms to optimize the model efficiency by algebraic transformations of the DAE systems will be presented. Figure 6.1 shows the bottom-up modeling flow including the model optimization. The term “optimization” within this context does not refer to the mathematical meaning of optimization methods, but to its meaning within computer science and compiler design, improving the efficiency of a system. The intention is to automatically optimize the equation sets through algebraic reformulation (semantics-preserving transformations) with respect to the simulation algorithms. All algorithms have been integrated into *Analog Insydes* and can be applied to general DAEs. Thus, a flexible and modular application of the equation optimization is possible.

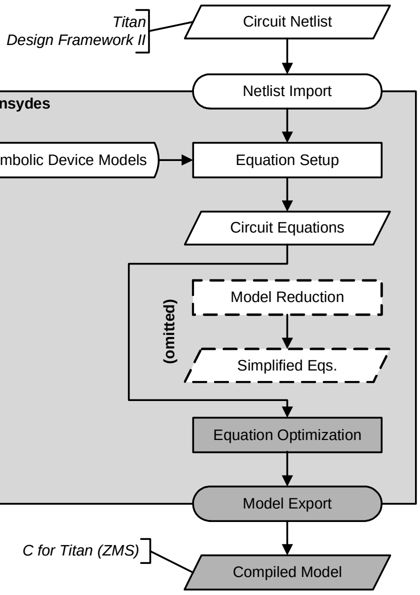


Figure 6.1: Bottom-Up Modeling Process with Model Optimization

Compiler Optimizations

Unlike in C-compilers, the reformulations are performed on a high abstraction level and are supported by an algebra system. Therefore, the variety of possible optimizations is much larger. A global optimization of the equation set is possible (e.g. structural changes of the equation system). Nevertheless, the strategies are very similar to those known from compiler design. [1] provides a comprehensive introduction to compilers and optimization strategies. Typical compilers perform machine-independent and machine-dependent code optimizations within a two-stage approach. The latter can only be performed with a detailed knowledge of the target platform in order to take advantage of its strengths. Similarly, a distinction between simulator-dependent (Chapter 5) and simulator-independent optimizations (Chapter 6). The following list contains a description of code optimizations that can be applied in a generalized form to DAE systems (refer to [1] for details):

- **Data-flow analysis** – Performing analyses of the interdependencies between expressions to optimize for parallelism and optimal evaluation order. A similar approach has been applied in order to identify sequential equations (for procedural evaluation) within a simultaneous equation set.
- **Common-subexpression elimination** – Recognition and pre-evaluation of repeatedly used subexpressions in order to avoid redundant computational effort. This optimization was applied to DAEs to reduce their complexity.
- **Copy propagation** – Recognition of constant assignments (e.g. $x = 1$) and copy statements (e.g. $x = y$), removal of the assignment, and propagation of the assigned value. Within DAEs, redundant equations can be removed by the same means.
- **Strength reduction** – Replacement of an expensive operation by an equivalent cheaper one (e.g. $x^2 \rightarrow x \cdot x$). Optimizations of this type could be performed directly on the (high) level of DAEs, but it is left to the C-compilation as it can easily be carried out there.
- **Inline expansion** – Replacement of a function call to a “simple” function by inserting the function’s contents. Thus, the overhead of the function call is avoided and data locality can be improved. Similarly, sequential equations cause overhead when being processed. If the sequential equation itself is “cheap” to evaluate, it is advantageous to substitute all occurrences of the assigned sequential variable by the sequential equation.
- **Dead-code elimination** – Removal of expressions which compute values that are never used.
- **Algebraic simplification** – Application of algebraic properties to simplify expressions (e.g. algebraic identities $x + 0 \rightarrow x$, $x \cdot 1 \rightarrow x$). This optimization is implicitly achieved by the used algebra system. For implementational reasons, a special treatment of algebraic simplifications within if-statements was necessary (due to `HoldRest` attribute of Mathematica).
- **Constant folding** – Pre-evaluation of constant expressions at compilation time to save computational effort at runtime. This optimization is also covered by the algebra system that automatically evaluates constant expressions. Again, if-statements require the function `FoldConstants` (due to `HoldRest` attribute of Mathematica).
- **Loop-invariant expressions** – Evaluating loop-invariant expressions outside the loop. This kind of optimization was used in a generalized way within Chapter 5 (preloading of constant entries of the Jacobian matrix).

In addition to the optimizations applied during model generation, the C-compiler’s optimizations are applied during the compilation of the C-based model and have turned out to achieve further performance improvements. These optimizations probably speed-up the execution by an optimized evaluation order (instruction level parallelism) and improved storage mechanisms (like cache optimization and register allocation).

Similar approaches to the new optimization methods performed within *Analog Insydes* are:

- **Modelica modeling language** – This object-oriented modeling language for multi-physics systems provides optimizations dealing with semi-symbolic systems [20, 51, 54]. The equations are automatically reformulated and hence the application is very convenient for users.
- **(Behavioral) model compilers** – Performing the optimizations within the model compiler would be the most obvious and user-friendly method. Unfortunately, current versions of commercial model compilers seem to either apply no optimizations or they do not efficiently deal with complex analytical models (refer to Section 4.8).
- **Device model compilers** – The efficient compilation of device models requires a high level of optimization. The device model compiler MCAST for instance, has been reported to provide various optimization techniques [80] whereas ADMS [44] does not.

As *Analog Insydes* supports the model export for a wide variety of languages (VHDL-AMS, Verilog-A, MAST, ZMS), the optimizations have been proven to be efficient for different simulators. Optimizing models with respect to a specific simulator would yield the best results but requires a specific knowledge of the target simulator's algorithms and interferes with the paradigm of simulator-independent models. These simulator-dependent optimizations therefore have to be performed within the model compilers (as presented for *Titan* within the previous chapter).

The overall target of the DAE optimizations focuses on two properties of the equation set with regard to the necessary simulation effort: the achievement of a DAE system of sequential structure that consists of a minimal number of simultaneous equations and is of low redundancy.

As discussed in Section 4.4, different (equivalent) formulations for network equations are available. Within the proposed bottom-up modeling flow (cf. Figure 6.1), the network equations are the basis for the model optimization and code generation. Hence, it is highly desirable to choose an initial formulation of the network equations that is advantageous for the following process steps. Due to its compact formulation, the MNA setup is the preferred solution. STA unnecessarily enlarges the dimension of the equation set whereas a compressed equation set is of disadvantage due to redundant expressions.

Section 6.4 will present an example for the equation optimization. The results of the optimization strategies will be summarized within Section 6.5. For the usage of the new *Analog Insydes* functions please refer to Appendix B.2.

6.1 Recognition of Sequential Equations

In [58], an algorithm for taking advantage of sequential equations was presented. The recognition was done during the MNA-like set up of the circuit equations using a bottom-up modeling strategy. Within this section, a general algorithm to identify sequential equations from a system of DAEs will be presented [94]. This approach is also suitable for manual modeling and is more general in terms of recognition. It was realized through a new function `IdentifySequentialEquations`.

The algorithm's objective is to determine a partitioning and ordering of the equations of a system of DAEs being compliant with Definition 2.3 (on Page 13) and having as many sequential equations as possible. An additional criterion for a good partitioning is to handle as many nonlinear equations as possible in a procedural way. This helps to prevent convergence problems during the iteration of the remaining simultaneous system.

The resulting Jacobian matrix of the DAE system is of block lower-triangular (BLT) structure as visualized in Figure 6.2. The dimension of the lower-triangular submatrix J_{11} is typically significantly larger than the dimension of J_{22}. The modeling language Modelica also provides an algorithm for BLT-transformation of nonlinear equation systems [20] that originates from [18]. In [63], an algorithm for block triangularization of a sparse matrix for direct methods was presented. The identification process for DAEs does not only affect the reordering of the matrix, but also the reformulation of the equations and ensuring the solvability of the resulting DAEs.

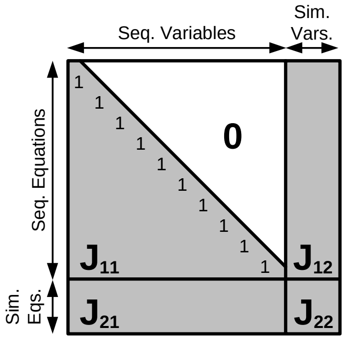


Figure 6.2: Block Lower-Triangular Matrix Structure

During the identification process, the equations and the variables of the original system are successively permuted and separated into sequential and simultaneous subsets. Figure 6.3 shows the flow chart for the identification of sequential equations. Initially, the algorithm starts with all equations and variables unclassified. Optionally, it is possible to start with a user-defined initial set of simultaneous variables simplifying the identification process and/or an initial set of sequential equations which will be kept during the processing. As it is advantageous to identify as many nonlinear equations as possible, the unclassified equations are preordered giving the nonlinear equations a higher priority.

The algorithm itself is based on the analysis of dependency matrices having the same structure as the Jacobian matrix and indicating for each equation which variables are referenced. The identification searches the set of unclassified equations recursively for additional sequential equations. Therefore, the equations have to fulfil three criteria:

- to fit into the lower-diagonal block structure,
- to be explicitly solvable for the corresponding sequential variable,
- and not to result in (implicit) second-order derivatives.

For the first criterion, the dependencies of the equation are checked. In order to be compliant with the lower-diagonal structure, the equation must neither contain any unclassified variables (except the one, it will be solved for) nor derivatives of unclassified variables. For ensuring that a new sequential equation is explicitly solvable for its sequential variable a pattern-matching method is applied. For practical reasons (convergence) it has been restrict-

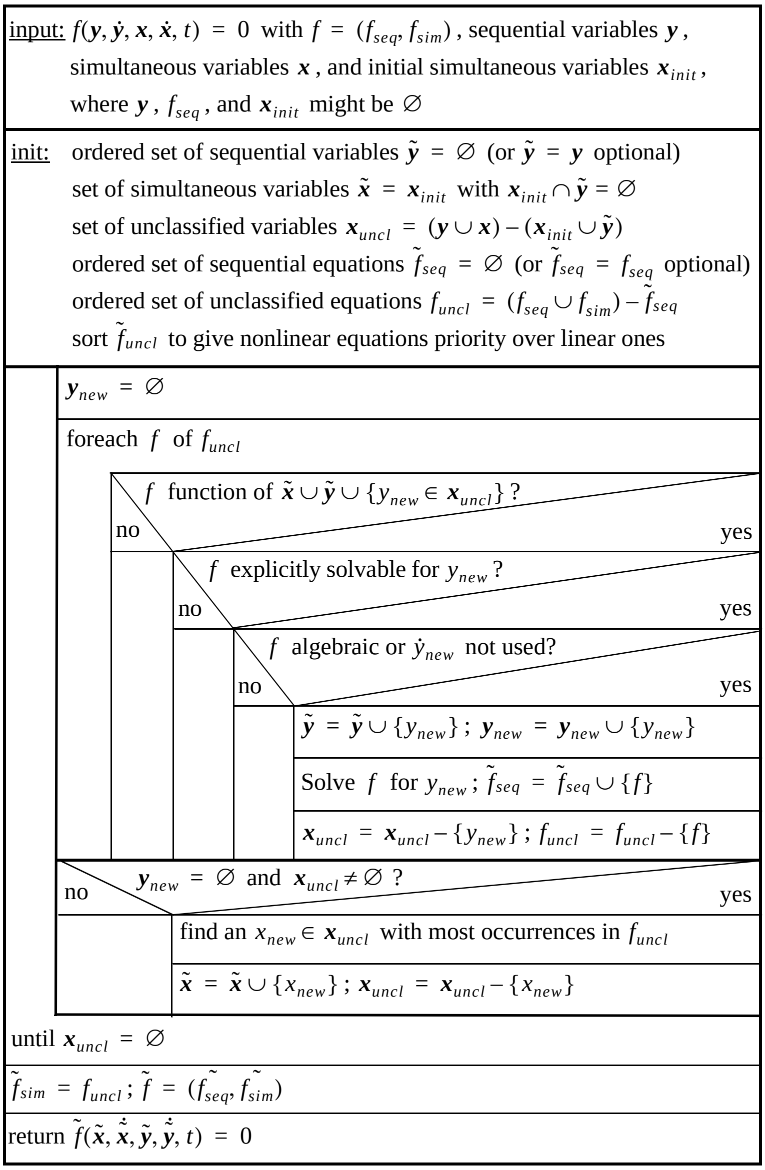


Figure 6.3: Flow Chart of the Identification of Sequential Equations

ed to apply the symbolic solving only in linear and weakly nonlinear contexts. Optionally, it can also be applied to symbolically solve nonlinear equations (as long as their solution is unique). Finally, the new sequential equation must not imply second-order derivatives. Therefore, it is checked to be either an algebraic equation or that its sequential variable does not appear in a dynamic context. Equations meeting all requirements are symbolically solved to be of explicit form and subsequently added to the sequential subsystem.

The identification process is executed in a fixed-point iteration until no new sequential variables are found. Unless all variables have been classified, it is necessary to declare one of the remaining variables as simultaneous variable, as this will almost always allow to solve some more unclassified equations sequentially. Every time the identification loop is stuck, the algorithm heuristically determines a variable to be kept simultaneously. The larger the number of dependent unclassified equations for a variable, the better is this variable suited to be treated simultaneously since this reduces the number of unknowns for as many remaining equations as possible. Afterwards, another identification cycle starts.

Once all variables have been classified, the identification process terminates. All remaining equations are declared to be simultaneous equations. Example 6.1 illustrates the identification process step by step for an arbitrary system of nine equations (numbered 1 to 9) depending on the variables A to I.

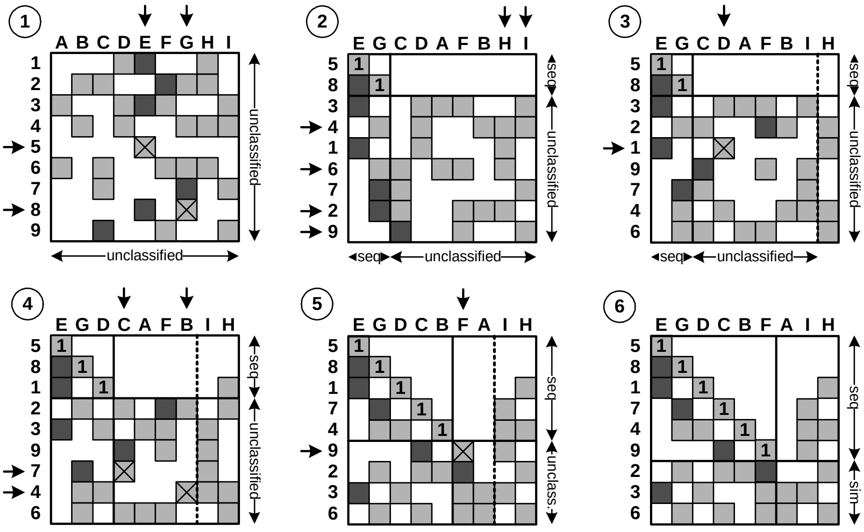


Figure 6.4: Example for the Sequential Identification Algorithm

Example 6.1: Identification of Sequential Equations

Figure 6.4 illustrates the algorithm in six steps. Each diagram represents the dependency matrix of the equations on the corresponding variables. Light gray boxes represent linear nonzero entries of the Jacobian matrix whereas dark gray boxes mark nonlinear dependencies. The white boxes depict structural zero entries. The arrows indicate rows or columns which have been identified to be swapped within the next process step. The used equations and sets of variables are visualized at the right and bottom sides of the matrices. The crosses indicate entries to be identified as sequential variables. These entries are moved to the correct position by row and column swapping and the corresponding equation is reformulated into an explicit formulation. Hence, the diagonal entry becomes one (ensuring good pivot elements).

Within the example, six equations and variables have been identified to be sequentially solvable (see Diagram 6). To emphasize some special behavior, two transitions will be discussed in detail: In Diagram 2, the algorithm is stuck since no equation depends on less than two unclassified variables. In order to solve this problem, one variable with as many dependencies on unclassified equations is declared to be solved simultaneously. Choosing variable H as simultaneous variable enables the algorithm to identify Equation 1 with variable D. At the stage of Diagram 5, both Equation 9 or 2 could be used as sequential equations for variable F. As Equation 2 contains F in a nonlinear context, it is more useful to select Equation 9.

■

A future aspect related to the discussed BLT transformation is the automatic partitioning of the sequential equations into multiple lightly coupled subsystems. This structure would be of advantage to further improve the Schur-complement or to even parallelize the evaluation of the subsystems.

6.2 Common Subexpression Elimination

Reducing the redundancy within DAEs through common subexpression elimination [94] is another promising approach. In order to apply common subexpression elimination (CSE) to general DAEs, two major process steps are performed:

- Recognition of common subexpressions within the DAEs
- Substitution by additional sequential equations and variables

The recognition of common subexpressions is based on abstract syntax trees (AST) [1]. An AST is a finite, labeled, directed tree where the internal nodes are labeled by operators, and the leaf nodes represent the operands (in this case variables, parameters, and constants). They are used as intermediate data structures in parsers to represent the syntactical hierarchy of source code. Figure 6.5 shows an exemplary expression tree for the following equation:

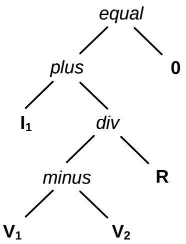


Figure 6.5: AST Example

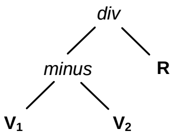
$$I_1 + \frac{V_1 - V_2}{R} = 0 \tag{6.1}$$

The flow chart for the CSE algorithm is shown in Figure 6.6. In an initial step, an abstract syntax tree for each equation within the DAE system is set up. The resulting forest F of syntax trees is the basis for the recognition of common subexpressions. As there is no canonical form for nonlinear expressions, and since the recognition only matches identical subtrees E_{com} within F, some basic algebraic transformations are applied in order to (heuristically) normalize the expressions and thereby enhance the recognition rate. All operands of commutative operations are alphabetically sorted with the first operand normalized to a positive sign.

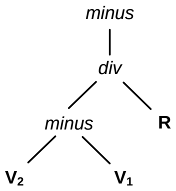
Example 6.2: Normalization of Subexpressions

Consider as an example the expressions E_1 and E_2:

$E_1 : \frac{V_1 - V_2}{R}$ with



and $E_2 : -\frac{V_2 - V_1}{R}$ with



Mathematically, both expressions are equivalent. However, their syntax trees are not identical, thus disabling the CSE. The normalization reformulates E_2 to the form of E_1.

■

After performing the discussed normalization, the algorithm extracts all subtrees E_{sub} of a user-defined minimal depth out of F. Eliminating subexpressions of very low depth (e.g. 2) causes more overhead by handling the additional equations than it saves effort during evaluation. Therefore, the default value for the minimal depth is 3. A duplicate search in E_{sub} yields the common subexpressions E_{com} of the DAE system.

Instead of eliminating these subexpressions directly and recursively proceeding the CSE (which would yield a large number of very simple expressions), the expressions E_{com} are iteratively expanded by one level of hierarchy, unless the number of their occurrences within F decreases. Thus, a reasonable number of subexpressions of maximal depth is recognized without compromising efficiency.

The second step is the substitution. For each expression of E_{com} an additional sequential equation and a new sequential variable are introduced. Subsequently, the occurrences of the subexpressions are substituted by the newly introduced variable. In order to keep the sequential structure of the DAEs, each new sequential equation is inserted before its first usage within the equation set. Additional equations which have only been used within the simultaneous equations are appended at the end of the sequential equation set.

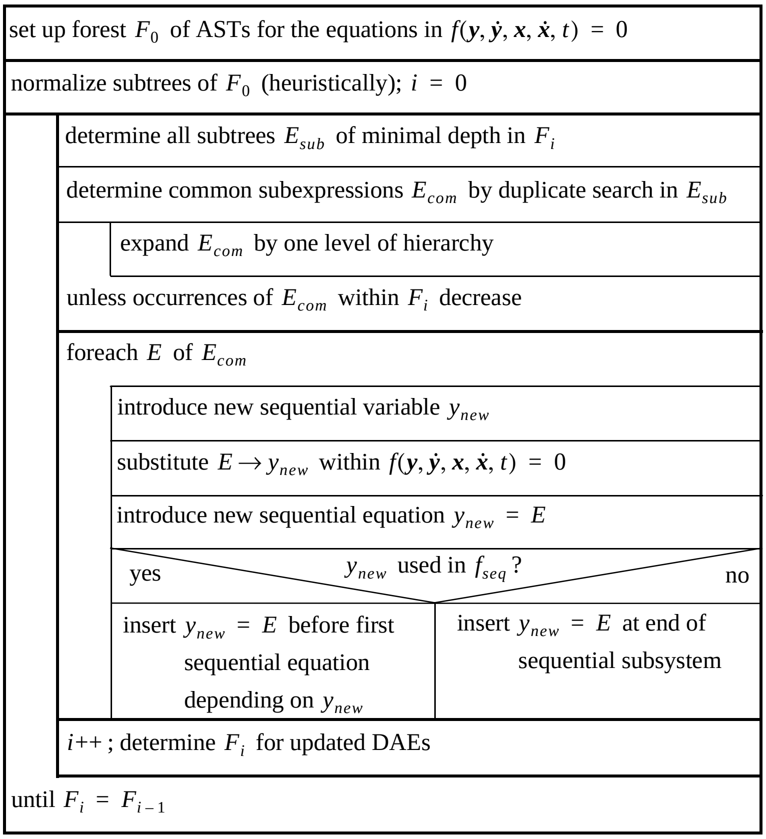


Figure 6.6: Flow Chart of the Common Subexpression Elimination

Although this strategy increases the number of sequential equations, the computational effort is reduced by avoiding multiple evaluations of common subexpressions. The elimination is performed globally (on the complete DAE system) and is applied in a fixed-point iteration to detect so-called deep subexpressions recursively (subexpressions that contain previously eliminated subexpressions).

The new *Analog Insydes* function `OptimizeCommonSubexpressions` also contains an alternative algorithm for detecting common subexpressions. The algorithm was introduced within [70] and is also suited for recognizing common subexpressions. In contrast to the previously presented algorithm, it also extracts expressions of depth 2 (e.g. $-V_1$) thus causing an unacceptable overhead. Therefore, additional postprocessing through inline expansion of cheap subexpressions is necessary.

The results of both algorithms are of similar quality with respect to the evaluation cost. While the algorithm of Sofroniou yields a large number of common subexpressions with quite low complexity each (and causes additional effort for inline expansion), the proposed algorithm leads to a reasonable number of more complex subexpressions (through expansion of the expressions).

Even though the number of sequential equations is only of secondary interest, each sequential equation causes some computational overhead (Schur-complement, storage). There is a trade-off between the caused overhead and the complexity of the sequential equation as well as the number of references to the sequential variable. If a sequential equation is cheap in terms of evaluation cost and its sequential variable is rarely used, it is more efficient to substitute the sequential variable with its defining equation (inline expansion). The intention of the function `SubstituteSequentialEquations` is to perform an inline expansion of “cheap” sequential equations. It is based on three criteria: the depth of the sequential equation, its evaluation cost, and the number of references to the sequential variable. Any sequential equation that violates one of the requirements, is removed through inline expansion.

Up to now, CSE has only been applied to DAE systems to enhance the function evaluation. When generating compiled models, another possibility in order to further increase the models’ efficiency arises. Even if all common subexpressions within an equation set have been eliminated before the model generation, the Jacobian matrix of the DAE contains most likely new common subexpressions. These common subexpressions can not be prevented when generating AHDL models, since there is no possibility to influence the internal Jacobian matrix. In contrast to that, the generation of compiled models offers the possibility to perform the CSE on both the DAE system as well as on its Jacobian matrix. This feature was integrated into the ZMS model generation. Thus, redundancy resulting from common subexpressions during the loading process can be reduced efficiently. Furthermore, the evaluation of the function as well as the Jacobian matrix entries are nested such that each equation, its derivatives, and its common subexpressions are evaluated in a sequence. Thus, data locality is increased and evaluation performance enhanced.

A specialized application of CSE is the recognition and extraction of loop-invariant expressions. As these expressions only contain constants and parameters they do not have to be evaluated within each iteration. In fact, such loop-invariant expressions can be calculated once within the models’ initialization phase and can be reused within subsequent iterations. This feature can be used within all AHDLs and compiled models that support an initialization function[1]. A slightly modified version of the CSE algorithm (integrated into the same function) allows to extract such loop-invariant expressions into additional sequential equations. Therefore, the criteria for the recognition have been modified: The expressions do not necessarily have to be common expressions, a single appearance is sufficient, and they must not contain any variables. During the model generation sequential equations consisting only of constants and parameters are moved to the initialization function.

---

[1] e.g. for Verilog-A: @(initial_step)

6.3 Elimination of Redundant Equations

Superfluous equations and variables within the DAEs are further causes for redundancy. Two cases have to be distinguished:

- Trivial equations that equate two symbols (copy/constant propagation)
- Unused sequential equations (dead-code elimination)

While the first case happens frequently due to the MNA-setup of the DAEs, the latter is an unlikely case that will only become relevant when optimizing poorly formulated equations (relevant for import and optimization of ADHL-based models).

The recognition of trivial equations is based on pattern matching. All equations that contain only two symbols in an additive context (e.g. $var_1 = var_2$, $var_1 = -param$, $var_1 - var_2 = 0$) are checked for removal. Depending on the type of the symbols (constant, parameter, seq./sim. variable, differential variable) and the type of the equation (sequential or simultaneous), several decisions have to be made:

- Is the equation really redundant?
- Can the equation be removed?
- Which of the symbols should be removed?

After deciding to remove an equation and one of the contained variables, the corresponding “copying” variable and its derivatives are substituted with the “copied” symbol (constant, parameter, variable).

Although the problem itself appears to be trivial, there are some hazards:

- $seqvar_1 = seqvar_2$ – can only be removed, if it is a sequential equation. In a simultaneous context, it represents a simultaneous equation with both left- and right-hand side being defined by a sequential equation.
- $var_1 = param$ – parameters in *Analog Insydes* can be functions of time (e.g. for independent transient sources). In this case, var_1 can only be replaced, if it is not a dynamic variable.
- $var_1 = \dot{var}_2$ – substitution of var_1 would possibly increase the order of the DAE system.
- Variables can be protected by the user in order to prevent their substitution.

The elimination method is applied iteratively in order to eliminate multiple copy propagations, too.

Eliminating unused sequential equations performs a search for each of the sequential variables. A sequential variable that is not used in any equation (except its defining sequential equation), is removed together with its definition. The elimination of redundant equations is performed by the new *Analog Insydes* functions `RemoveTrivialEquations` and `RemoveRedundantEquations`.

6.4 Example Application

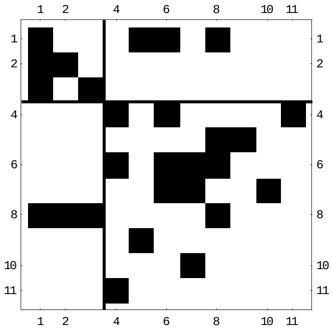


Figure 6.7: Jacobian Structure of the Diode Example (Initial)

The optimization algorithms will be illustrated by means of the diode circuit that was introduced in Example 2.2 (on Page 16). Even though the example is very simple, the most relevant optimizations can be sufficiently applied to the network equations of the diode circuit. The sequence of the optimization process is arbitrary. The optimizations may interfere with each other. Initially, the equation set consists of 3 sequential equations and 8 simultaneous equations. The evaluation cost $N_{EvalCycles}$ is approximately 288 including 129 memory accesses ($N_{MemAccess}$).

Table 6.1 summarizes the properties of the DAEs for different stages of the optimization process. Figure 6.7 visualizes the structural matrix of the initial DAE system as resulting from the MNA setup of the network equations. The system's sparsity is relatively high and the lower-diagonal subblock is only of dimension three.

Common Subexpression Elimination

The CSE algorithm is able to identify three common subexpressions resulting from resistor and capacitor branch currents that each appear within two nodal equations. The corresponding sequential equations have been appended at the end of the sequential equation set that is shown in Equation (6.2a), since they are only referenced within the simultaneous equations. All occurrences of the former common subexpressions have been substituted by the newly introduced sequential variables. Thus, the evaluation cost was reduced by 17 %.

$$\begin{aligned}
&\mathtt{vd[t]} == -\frac{\mathtt{RS\,I\$AC[t]}}{\mathtt{AREA}} + \mathtt{V\$IN[t]} - \mathtt{V\$OUT[t]}\\
&\mathtt{id[t]} == -\mathtt{AREA}\left(e^{-\frac{0.00333167\,\mathtt{q}\,(\mathtt{BV}+\mathtt{vd[t]})}{\mathtt{k}}}\,\mathtt{IBV} - \left(-1 + e^{\frac{0.00181069\,\mathtt{q}\,\mathtt{vd[t]}}{\mathtt{k}}}\right)\mathtt{IS}\right) + \mathtt{GMIN}\,\mathtt{vd[t]}\\
&\mathtt{cj[t]} == \mathtt{AREA}\,\mathtt{CJO}\,\mathtt{If}\left[\mathtt{vd[t]} < 0.25,\ \frac{1}{(1.-2.\,\mathtt{vd[t]})^{0.333}},\ 2.51926\,(0.3335 + 0.666\,\mathtt{vd[t]})\right]\\
&\mathtt{SeqExpr1[t]} == \mathtt{C0}\,(\mathtt{V\$GND'[t]} - \mathtt{V\$OUT'[t]})\\
&\mathtt{SeqExpr2[t]} == \frac{\mathtt{V\$GND[t]} - \mathtt{V\$OUT[t]}}{\mathtt{R0}}\\
&\mathtt{SeqExpr3[t]} == \frac{\mathtt{V\$OUT[t]} - \mathtt{V\$OUT2[t]}}{\mathtt{RL}}
\end{aligned} \tag{6.2a}$$

$$\begin{aligned}
&\mathtt{I\$VGND[t]} + \mathtt{SeqExpr1[t]} + \mathtt{SeqExpr2[t]} == 0\\
&\mathtt{I\$AC[t]} + \mathtt{I\$VIN[t]} == 0\\
&-\mathtt{I\$AC[t]} - \mathtt{SeqExpr1[t]} - \mathtt{SeqExpr2[t]} + \mathtt{SeqExpr3[t]} == 0\\
&\mathtt{I\$VOUT[t]} - \mathtt{SeqExpr3[t]} == 0\\
&\mathtt{I\$AC[t]} == \mathtt{id[t]} + 1.15\times 10^{-8}\,\mathtt{id'[t]} + \mathtt{cj[t]}\,\mathtt{vd'[t]}\\
&\mathtt{V\$IN[t]} == \mathtt{VIN}\\
&\mathtt{V\$OUT2[t]} == 0\\
&\mathtt{V\$GND[t]} == 0
\end{aligned} \tag{6.2b}$$

Recognition of Sequential Equations

From Equation (6.2b) it is obvious, that a major part of the simultaneous equations can be solved sequentially. Applying the BLT transformation in order to identify further sequential equations yields a reduction of six simultaneous equations. This results in the equation set (6.3a) and (6.3b).

$$\begin{array}{l}
\mathrm{V\$GND}[t] == 0 \\
\mathrm{V\$IN}[t] == \mathrm{VIN} \\
\mathrm{V\$OUT2}[t] == 0 \\
\mathrm{I\$VIN}[t] == -\mathrm{I\$AC}[t] \\
\mathrm{vd}[t] == -\frac{\mathrm{RS}\,\mathrm{I\$AC}[t]}{\mathrm{AREA}} + \mathrm{V\$IN}[t] - \mathrm{V\$OUT}[t] \\
\mathrm{SeqExpr2}[t] == \frac{\mathrm{V\$GND}[t]-\mathrm{V\$OUT}[t]}{\mathrm{R0}} \\
\mathrm{SeqExpr3}[t] == \frac{\mathrm{V\$OUT}[t]-\mathrm{V\$OUT2}[t]}{\mathrm{RL}} \\
\mathrm{id}[t] == -\mathrm{AREA}\left(e^{-\frac{0.00333167\,q\,(\mathrm{BV}+\mathrm{vd}[t])}{k}}\,\mathrm{IBV} - \left(-1 + e^{\frac{0.00181069\,q\,\mathrm{vd}[t]}{k}}\right)\mathrm{IS}\right) + \mathrm{GMIN}\,\mathrm{vd}[t] \\
\mathrm{I\$VOUT}[t] == \mathrm{SeqExpr3}[t] \\
\mathrm{cj}[t] == \mathrm{AREA}\,\mathrm{CJO}\,\mathrm{If}\left[\mathrm{vd}[t] < 0.25,\ \frac{1}{(1.-2.\,\mathrm{vd}[t])^{0.333}},\ 2.51926\,(0.3335 + 0.666\,\mathrm{vd}[t])\right] \\
\mathrm{SeqExpr1}[t] == \mathrm{C0}\,(\mathrm{V\$GND}'[t] - \mathrm{V\$OUT}'[t]) \\
\mathrm{I\$VGND}[t] == -\mathrm{SeqExpr1}[t] - \mathrm{SeqExpr2}[t]
\end{array} \tag{6.3a}$$

Only two equations (out of previously eight) have to be solved simultaneously:

$$\begin{array}{l}
-\mathrm{I\$AC}[t] - \mathrm{SeqExpr1}[t] - \mathrm{SeqExpr2}[t] + \mathrm{SeqExpr3}[t] == 0 \\
\mathrm{I\$AC}[t] == \mathrm{id}[t] + 1.15\times 10^{-8}\,\mathrm{id}'[t] + \mathrm{cj}[t]\,\mathrm{vd}'[t]
\end{array} \tag{6.3b}$$

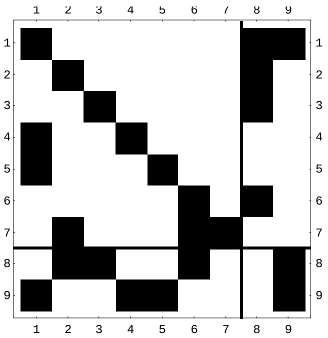


Figure 6.8: Jacobian Structure of the Diode Example (Optimized)

Removal of Trivial Equations

Still, the sequential equations (6.3a) contain several trivial equations causing redundancy. Removing these equations reduces the dimension of the sequential equation set by five. Equations (6.4a) and (6.4b) show the final stage of the optimization process for the diode example. The structure of the Jacobian matrix for the optimized DAEs is shown Figure 6.8.

Compared to the initial equation set, the resulting DAEs (last column of Table 6.1) have been reduced to a dimension of only two simultaneous equations (-75 %), the evaluation cost has been reduced by 25 %, and the number of memory accesses has been reduced by 17 %. Within the following section, the algorithms will be applied to problems of realistic dimension. For the upcoming examples, performance measurements will also be compared in order to demonstrate the enhancement of the simulation performance.

$$\begin{aligned}
&\mathrm{vd}[t] == \mathrm{VIN} - \frac{\mathrm{RS}\,\mathrm{I\$AC}[t]}{\mathrm{AREA}} - \mathrm{V\$OUT}[t] \\
&\mathrm{SeqExpr2}[t] == -\frac{\mathrm{V\$OUT}[t]}{\mathrm{R0}} \\
&\mathrm{SeqExpr3}[t] == \frac{\mathrm{V\$OUT}[t]}{\mathrm{RL}} \\
&\mathrm{id}[t] == -\mathrm{AREA}\left(e^{-\frac{0.00333167\,q\,(\mathrm{BV}+\mathrm{vd}[t])}{k}}\,\mathrm{IBV} - \left(-1 + e^{\frac{0.00181069\,q\,\mathrm{vd}[t]}{k}}\right)\mathrm{IS}\right) + \mathrm{GMIN}\,\mathrm{vd}[t] \\
&\mathrm{cj}[t] == \mathrm{AREA}\,\mathrm{CJO}\,\mathrm{If}\left[\mathrm{vd}[t] < 0.25,\ \frac{1}{(1.-2.\,\mathrm{vd}[t])^{0.333}},\ 2.51926\,(0.3335 + 0.666\,\mathrm{vd}[t])\right] \\
&\mathrm{SeqExpr1}[t] == -\mathrm{C0}\,\mathrm{V\$OUT}'[t] \\
&\mathrm{I\$VGND}[t] == -\mathrm{SeqExpr1}[t] - \mathrm{SeqExpr2}[t]
\end{aligned} \tag{6.4a}$$

$$\begin{aligned}
&-\mathrm{I\$AC}[t] - \mathrm{SeqExpr1}[t] - \mathrm{SeqExpr2}[t] + \mathrm{SeqExpr3}[t] == 0 \\
&\mathrm{I\$AC}[t] == \mathrm{id}[t] + 1.15\times 10^{-8}\,\mathrm{id}'[t] + \mathrm{cj}[t]\,\mathrm{vd}'[t]
\end{aligned} \tag{6.4b}$$

Table 6.1: Optimization Process for the diode Example

Property	***Initial***	***⇨ CSE***		***⇨ BLT***		***⇨ RTE***		***Final***	
Seq. Eqs.	3	6	(+3)	12	(+6)	7	(-5)	7	(+133 %)
Sim. Eqs.	8	8		2	(-6)	2		2	(-75 %)
$N_{EvalCycles}$	288	239	(-49)	238	(-1)	215	(-23)	215	(-25 %)
$N_{MemAccess}$	129	136	(+7)	136		106	(-30)	106	(-18 %)

6.5 Results

Within this section the results of the DAE optimizations will be presented on the basis of four example circuits. For each of the circuits, four model types in different intermediate stages of the optimization process will be compared:

- **sim** – the simultaneous model of the circuit after applying the RTE algorithm
- **seq** – the sequential model including the sequential equations resulting from the symbolic device models after applying the RTE algorithm
- **blt** – the optimized sequential model after applying the BLT algorithm
- **blt/cse** – the optimized sequential model after applying BLT and CSE algorithms

Before reviewing the simulation performance of the models, the efficiency of the optimization algorithms will be discussed on the basis of the models' characteristics. Table 6.2 summarizes the optimization results for the presented models. The key figures are the number of simultaneous equations, in order to show the efficiency of the BLT algorithm, as well as the evaluation complexity, showing the effect of the CSE.

Table 6.2: Optimization Results (Model Characteristics)

Example	***Type***	N_{SeqEqs}	N_{SimEqs}	$N_{EvalCycles}$
multiplier	blt/cse	72	25	4801
	blt	64	25	7882
	seq	48	38	7874
	sim	0	86	7946
opamp741	blt/cse	339	61	12002
	blt	243	61	16788
	seq	177	127	16895
	sim	0	304	17085
nand2	blt/cse	339	6	30889
	blt	265	6	57268
	seq	254	17	57285
	sim	0	271	57683
cfcamp	blt/cse	1707	16	153722
	blt	1375	16	275111
	seq	1313	78	275135
	sim	0	1391	277032

Figure 6.9 depicts the dimension of the resulting linear systems for the different model types. The high dimensions of the simultaneous models are very efficiently reduced by utilizing the sequential equation structures. After having applied the presented BLT algorithm, the dimension of the models is very close or even equal to the dimension that is achieved in netlist-based simulations. This provides the basis for high simulation performance. Above all, the BSIM3-based models show an enormous reduction of the model dimension by using the sequential equations (*cfcamp*: from 1396 to 21). Solving these models simultaneously would result in an enormous overhead due to the large number of equations.

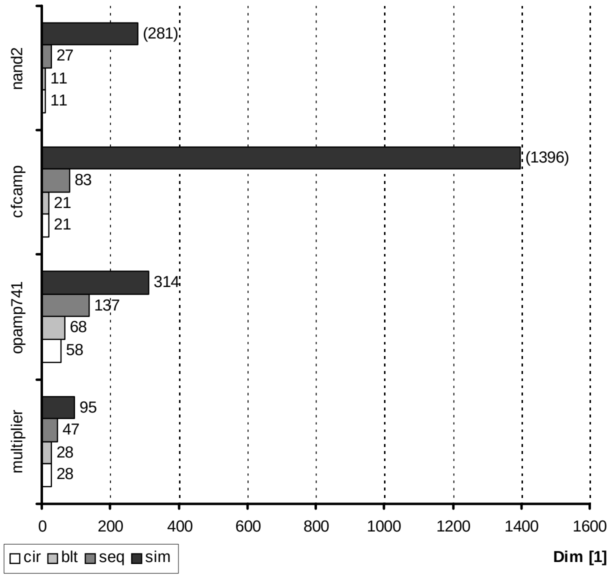


Figure 6.9: Optimization of the Dimension by Utilizing Sequential DAE Structures

The achieved reduction of redundant common subexpressions through the CSE algorithm is demonstrated in Figure 6.10. It shows the evaluation complexity before (seq) and after the application of the common subexpression elimination (blt/cse). For the Gummel-Poon based models, a reduction by 30 % to 40 % was achieved. For the BSIM3-based models, the CSE is of even higher efficiency. Their evaluation complexity was reduced by more than 45 %.

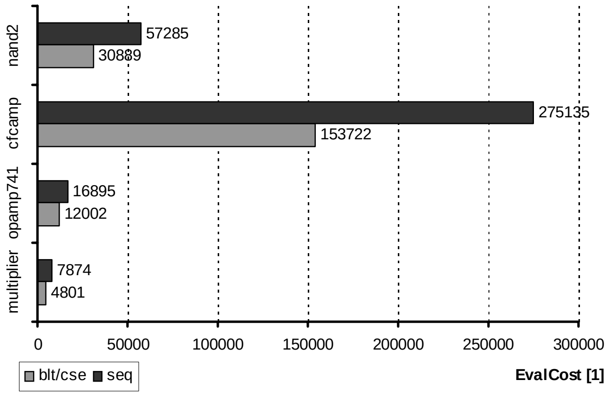


Figure 6.10: Effect of Common Subexpression Elimination on $N_{EvalCycles}$

Table 6.3: Simulation Performance of the Optimized ZMS-Models

Example	***Type***	Dim	$N_{iter/step}$	T_{load}	S_{load}^{-1}	T_{solve} [a]	S_{solve}^{-1}
multiplier	circuit	28	2.0	0.046 s	1	0.261 s	1
	blt/cse	28	2.9	0.128 s	2.78	0.333 s	1.28
	blt	28	2.9	0.152 s	3.3	0.349 s	1.34
	seq	47	2.95	0.159 s	3.46	0.57 s	2.18
	sim	58	2.97	0.178 s	3.87	1.09 s	4.18
opamp741	circuit	58	2.25	0.178 s	1	0.758 s	1
	blt/cse	68	3.05	0.339 s	1.9	1.366 s	1.8
	blt	68	3.05	0.487 s	2.74	1.342 s	1.77
	seq	137	3.05	0.502 s	2.82	2.405 s	3.17
	sim	314	3.1	0.577 s	3.24	5.058 s	6.67
nand2	circuit	11	2.09	0.075 s	1	0.206 s	1
	blt/cse	11	3.21	0.869 s	11.59	0.344 s	1.67
	blt	11	3.21	2.051 s	27.35	0.324 s	1.57
	seq	27	3.25	2.092 s	27.89	0.554 s	2.69
cfcamp	circuit	21	2.12	0.469 s	1	0.494 s	1
	blt/cse	21	2.87	6.543 s	13.95	1.197 s	2.42
	blt	21	2.87	24.67 s	52.6	1.83 s	3.7
	seq	83	2.87	26.02 s	55.48	3.707 s	7.5

a. CPU times for solving are disproportionately high due to the application of the *MUMPS* solver, cf. Section 5.3

Within Table 6.3 the simulation performance of the ZMS-based models is presented. Applying the optimization strategies improved the loading performance by 20 % to 75 % although the convergence of the models is by roughly 50 % worse compared to the circuit simulation. The performance improvements (16 % to 74 %) due to the application of CSE becomes apparent when examining the loading time. The BLT transformation efficiently speeds up the solving time of the *MUMPS* solver by 39 % to 50 %. Similar speed-ups are to be expected for future application of the *Titan* solver after its integration with the ZMS interface. Figure 6.11 depicts the loading performance of the optimized models normalized to the corresponding performance of the circuit simulation. The slow-down of the optimized models for *opamp741* was reduced to a factor of 1.9 only. For the BSIM-based models, factors of 11 to 14 have been achieved. Taking into account the large number of equations (345, 1723), the high complexity of the model equations, and the fact that no model reduction was applied, this is a great achievement.

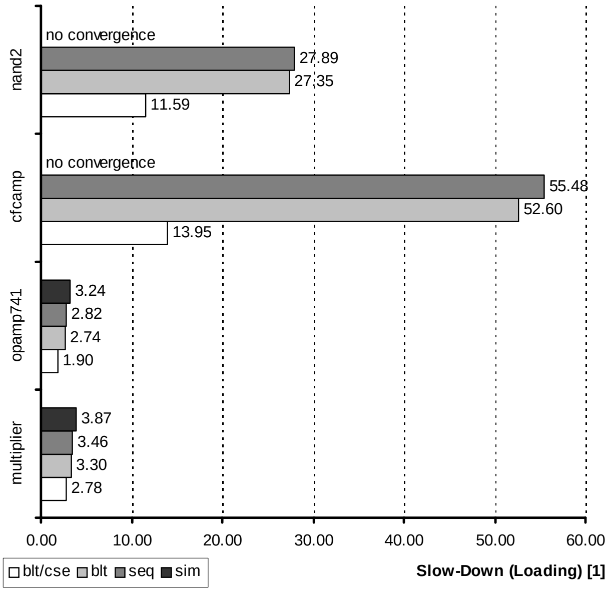


Figure 6.11: Loading Performance after Optimizations

Finally, Figure 6.12 shows the slow-down factors for the solving process of the optimized models. The proportionality of the CPU time for solving the linearized equation systems to the reduced dimension (refer to Figure 6.9) can be recognized very clearly. If the convergence of the behavioral simulations could be further improved, the solving performance would be even closer to the performance of the netlist-based simulation. Results achieved by using the optimization methods in conjunction with Verilog-A models have been published in [94, 97].

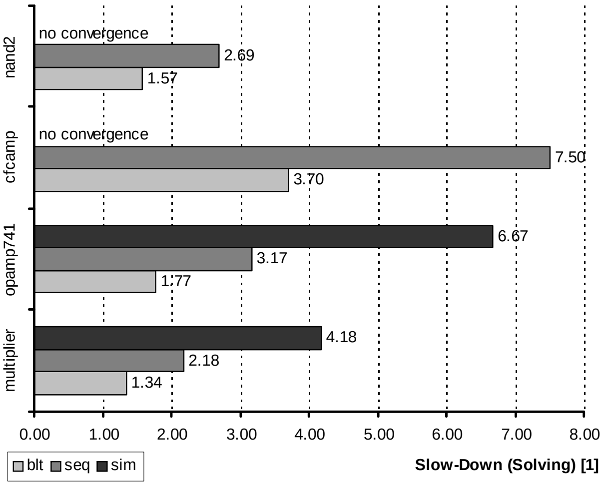


Figure 6.12: Solving Performance after Optimizations

7 Conclusion

Automated bottom-up modeling strategies are of great importance for modern structured top-down design flows. Symbolic analysis offers good opportunities for automated derivation of accurate behavioral models of analog circuit blocks. This work dealt with a bottom-up modeling flow based on the symbolic analysis toolbox *Analog Insydes*. The advantages of the tool's modeling strategy make it highly attractive for the creation of analytical behavioral models. Despite highly efficient model reduction algorithms, the simulation performance of the generated behavioral models used to be unacceptably low.

Objectives

This work aimed at performing an in-depth analysis of the behavioral simulation efficiency of the generated models using the *Titan* simulator to identify the root causes for such low performance. Furthermore, a significant enhancement of the simulation performance without further loss of accuracy was targeted in order to make the bottom-up modeling approach through symbolic analysis competitive. The basic assumption was that performance suffered from missing consideration of the applied simulation algorithms and the ability of simulators to efficiently deal with such complex behavioral models.

Problem Analysis

A detailed comparison of the performance of netlist-based and behavioral simulations was presented. It was based on unsimplified behavioral models generated without the application of model reduction. Thus, the simulation performance of the netlist-based simulation should be equal or at least very similar to the behavioral model's performance as both problems are of identical complexity. The results of the performance analyses lead to the conclusion that the simulation performance of complex analytical behavioral models is suboptimal due to inefficiencies in processing the behavioral models' equation sets. The slow-down compared to an equivalent netlist-based simulation ranges from 2.5 (for small models) up to 200 (for an operational amplifier). Currently, the speed-up achieved by the model reduction has to compensate for the initially low behavioral simulation performance of the unsimplified models. Such inefficiencies prevented the application of the modeling approach through symbolic analysis.

The analyses presented within Chapter 4 have shown that the simulation efficiency suffers from suboptimal model formulation as well as from shortcomings in the behavioral simulation algorithms due to the extraordinary high complexity of the models. In particular, the missing sparse handling for behavioral models led to a major slow-down for higher dimensional models. Furthermore, the dimension of the linearized equation systems of the models was significantly higher than that of the equivalent netlist-based simulation resulting in an increased simulation effort. Finally, the processing of the models turned out to be less efficient than in circuit simulations. Although convergence is essential for behavioral models, it is not the key to solving the discussed performance problems.

Achievements

Within Chapters 5 and 6, various approaches for improvements of the behavioral simulation performance by "simulator-friendly" modeling and major enhancements of the model compilation have been presented. These algorithms lead to significant improvements of the simulation efficiency by reformulating and restructuring the DAE systems without loss of accuracy. Most important are the identification of sequential equations and the elimination of common subexpressions. The former is very well suited to reduce the number of simultaneous equations by transforming the DAE system to a sequential structure with a maximum number of sequential equations. The latter focuses on a reduced evaluation complexity of the equation set by extracting common subexpressions. Furthermore, some strategies to reduce redundancy within the equation sets have been discussed. All optimization techniques presented so far have been integrated into an automated modeling and optimization flow. As far as the simulator is concerned, a highly efficient model compilation for *Titan* was developed. It incorporates an efficient sparse handling as well as a local solving method significantly reducing the dimension of the linearized equation system.

The improvements achieved within this work will be summarized on the basis of a representative modeling example (*opamp741*). Initially, the unsimplified behavioral model in TML had an astonishingly low simulation performance. Compared to its netlist-based counterpart,

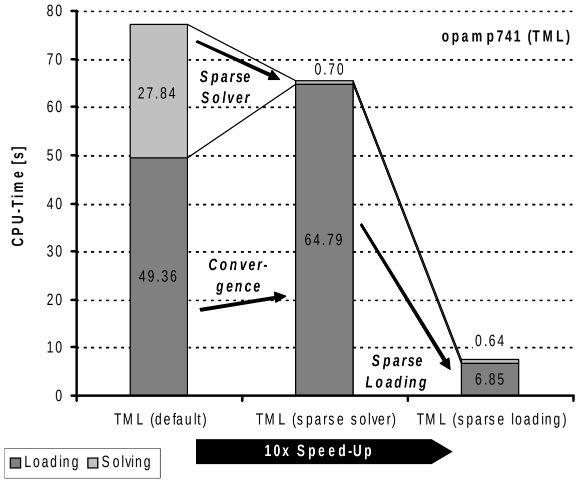


Figure 7.1: Performance Improvement through Sparse Algorithms (for *opamp741*)

the behavioral simulation performance was a factor of 192 less efficient. Figure 7.1 shows the improvements of the simulation performance that have been realized using TML models. The origin of the low performance could mostly be attributed to the missing application of sparse algorithms and sparse data structures. Through the application of the sparse *Titan* solver, the solving performance has been sped-up by a factor of 40 at the expense of inferior convergence due to pivoting problems (+30 % iterations). Additionally, the integration of a sparse data structure and processing within the *Titan* model compiler enhanced the efficiency of the loading process by a factor of 10. In total, the sparse handling for the TML models resulted in a speed-up of 10 for the *opamp741*.

As a consequence of the still unsatisfying performance, a model compiler based on a new modeling interface of *Titan* (ZMS) has been developed and integrated into *Analog Insydes*. It aims at increased processing efficiency and a more direct communication between the model and the simulator kernel. Thereby, the simulation performance has been improved further as depicted in Figure 7.2. The bar to the left of the chart represents the best performance that was achieved using TML models (cf. right bar of Figure 7.1). The simulation performance using the new ZMS-based models of the fully simultaneous DAE system (ZMS sim) of the amplifier has sped-up the loading process by a factor of 12. An unresolved issue of the ZMS-based models is the missing adaptation of the ZMS interface and the most performant linear solver (*Titan* solver). Hence, the models have been simulated using the *MUMPS* solver (dashed bars within the chart) which performed suboptimal compared to the *Titan* solver

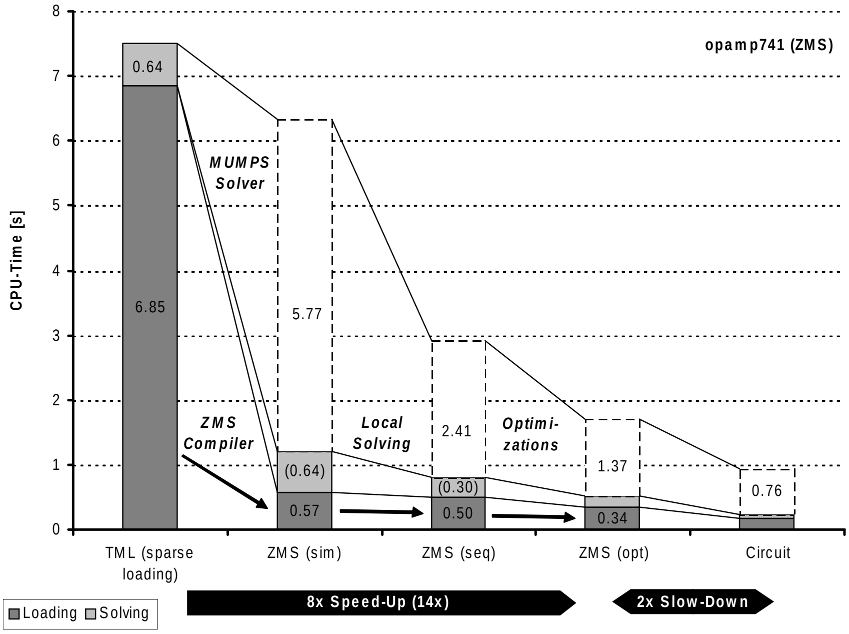


Figure 7.2: Performance Improvement through ZMS Compilation (for *opamp741*)

(cf. Section 4.5). As this is only a matter of development effort, the solving performance using the *Titan* solver has been pre-estimated (light gray contributions within the chart). For the fully simultaneous models, the solving performance has to be at least equal to the solving performance using the TML models. This is due to the fact that the resulting linearized equation systems are (nearly) identical. Based on this assumption, the simulation performance would be enhanced by a factor of 6 through the generation of compiled ZMS models. For the following levels of improvement, this basic appraisal for the *Titan* solver was scaled by the same speed-up factors as achieved for the *MUMPS* solver, being a conservative estimate.

Moreover, a local solving method was applied to efficiently reduce the dimension of the linearized equation systems to be solved by the simulator. This approach is based on utilizing the sequential structure of DAE systems. Sequential equations are solved locally within the compiled model so that only the remaining simultaneous equations have to be solved by the linear solver. Due to an efficient implementation of the necessary processing steps, this measure speeds-up the solving performance proportionally to the dimension of the simultaneous subsystem of the models' DAE systems. The loading performance is also enhanced due to the reduced amount of data, which is transferred between model and simulator kernel and increased data locality. For the *opamp741*, the local solving method doubled the simulation performance (for *MUMPS*); using the *Titan* solver a speed-up of 1.5 is likely.

Enabled by the possibility of solving sequential equations locally, several optimization strategies for DAE systems have been developed and integrated into *Analog Insydes*. They aim at improving the models' efficiency by reformulating equations and restructuring the DAE systems. The recognition of sequential equations reduces the dimension of the simultaneous subsystem and thus speeds-up simulation. Furthermore, a new algorithm for common-subexpression elimination within DAE systems reduces redundancy within the function evaluation of the model. Last but not least, particular attention was paid to data locality, efficient data structures, loop-invariant expressions, and redundant information within the equation system. Through the application of these optimization strategies, another speed-up of 42 % was achieved for the *opamp741*. These optimization strategies have also been applied to extremely complex models of up to 1700 equations based on a symbolic BSIM3 device model. For the latter, speed-ups of up to factor 4 have been achieved through the optimizations.

Finally, the models' simulation performance has been improved to a competitive level. For the *opamp741*, a slow-down of only factor 2 compared to the netlist-based simulation remains. Compared to the initial situation of a slow-down of factor 192, the efficient processing of the model equations resulted in a total speed-up of nearly 100 without reducing the model's accuracy. The model reduction algorithms can easily compensate for the remaining overhead. By combining the discussed approaches with the efficient model reduction strategy, a significant speed-up of the behavioral simulation in comparison to the circuit simulation was achieved [93]. Hence, the modeling flow may obtain increasing acceptance for automated bottom-up generation of highly efficient analytical models.

Future Aspects

Future aspects for the application of the presented model compilation could be the integration of the *Titan* solver in order to achieve competitive performance within the solving process. This requires a preordering strategy within *Titan* to guarantee the solvability of the equation system and to achieve a high level of accuracy. The pivoting algorithm has already been successfully applied to TML models and should be adaptable to ZMS-based models. With regard to model optimization, future aspects are preordering strategies for the simultaneous equations to perform a static pivoting before exporting the model. Thus, the conditioning of the DAE system could already be enhanced during model generation, reducing the risk of badly conditioned systems and numerical problems during simulation. Furthermore, Modelica provides an optimization strategy called “tearing” to decouple subsystems by introducing additional variables [21]. By using this strategy within *Analog Insydes*, the effectiveness of sequential equations could be improved due to possible parallel processing of lightly coupled subsystems.

By integrating import functionality for AHDL-based behavioral models into Analog Insydes, an even wider range of use cases would be possible. Figure 7.3 shows a proposed modeling flow visualizing some promising applications of the tool. As proof of concept, a prototypical function `ReadVerilogA` (rf. to Appendix B.3.8 for details) was realized. It was evaluated to read in a Gummel-Poon transistor model realized in Verilog-A. This function would be advantageous for extending the symbolic device model library of *Analog Insydes* by additional up-to-date device models.

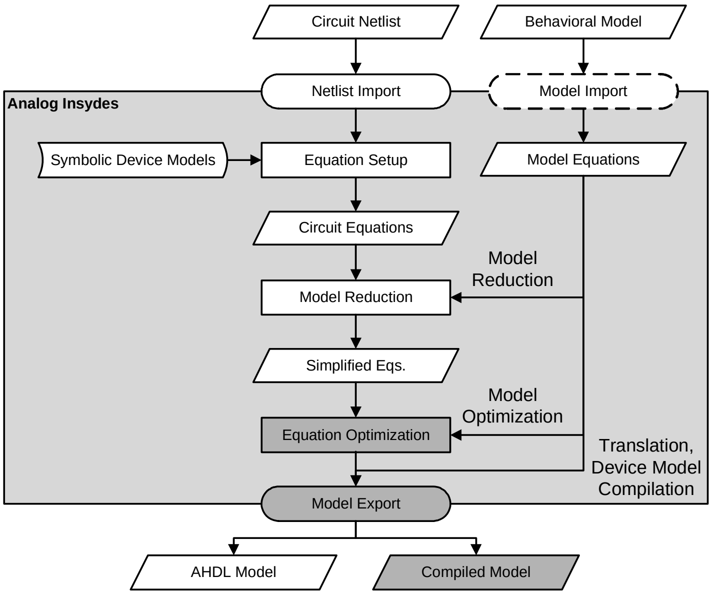


Figure 7.3: Future Applications of the Modeling Flow

The automated import of models would enable several new applications:

- **Device model compilation** – compilation of device models realized in an AHDL (typically Verilog-A) to simulator specific compiled models (e.g. ZMS for *Titan*)
- **Model optimization** – application of the optimization strategies to existing models
- **Model simplification** – application of the model reduction algorithms to DAE systems extracted from AHDL-based models
- **Model translation** – translating models from one AHDL to another (this would require extensive examination of language specific issues)

Furthermore, an extension of the general model compilation strategy for other simulators' compiled model interfaces is desirable. As this does typically not require fundamental changes within the equation processing but mainly structural and syntax changes within the generated models, such extensions are possible with limited effort. First approaches towards using Cadence CMI [91] have successfully been taken.

A Modeling Examples

A.1 cfcamp

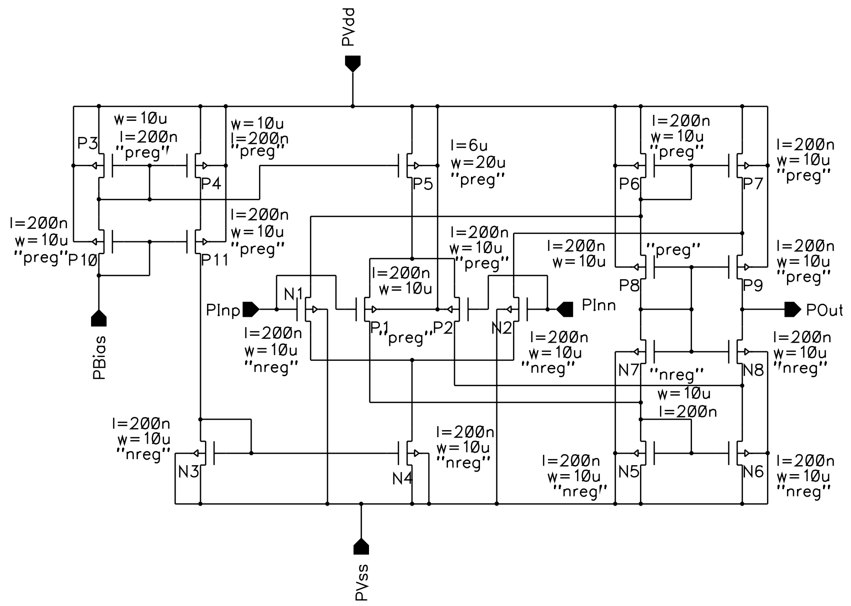


Figure A.1: Schematic of the *cfcamp* Circuit

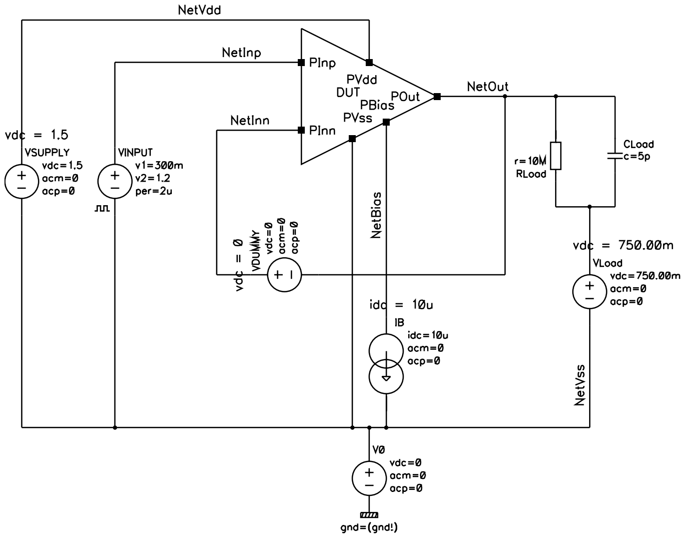


Figure A.2: Testbench for the *cfcamp* Circuit

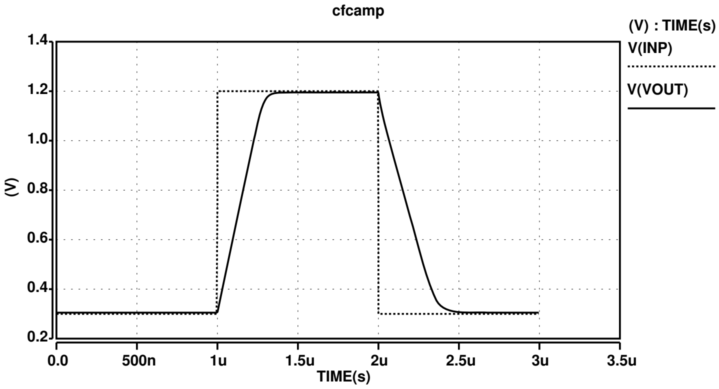


Figure A.3: Simulation Results of the *cfcamp* Circuit

A.2 diode

```
**
** TITAN Netlist for the diode circuit
**

** Model parameters for diode
.MODEL D1N4148 D(IS=2.68e-9 N=1.84 IKF=0.041 IBV=100e-6 BV=100 RS=0.6 CJO=4e-12
                   VJ=0.5 M=0.333 FC=0.5 TT=11.5e-9 XTI=3)

** Subcircuit Declaration
.SUBCKT DIODERC A C G
  D0 A C D1N4148 AREA=1
  C0 G C C='100p'
  R0 C G R='10K'
.ENDS

** DUT Instantiation
XI2 IN OUT GND DIODERC

** Testbench
VIN   IN   0    SIN (0 5 100)
VOUT  OUT2 0    DC '0' AC '0' '0'
VGND  GND  0    DC '0' AC '0' '0'
RLOAD OUT  OUT2 200Meg

** Simulation statements
.SAVE results FORMAT=scope5
.OP
.TRAN 1e-4 1
.OUTPUT TRAN LEVEL=0 V(*)

.END
```

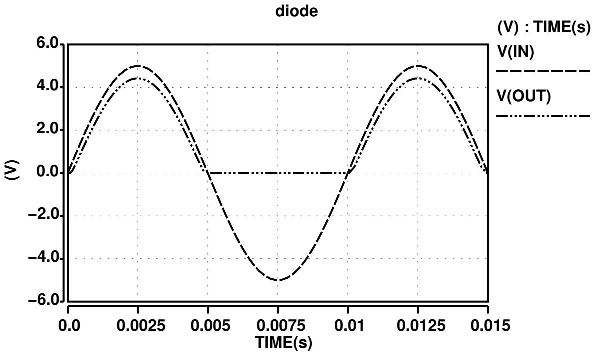


Figure A.4: Simulation Results of the *diode* Circuit

```
--
-- VHDL-AMS model of the diode circuit (entity)
--

LIBRARY ieee;
USE ieee.math_real.ALL;
USE ieee.electrical_systems.ALL;

ENTITY diodeRC IS
   GENERIC (
      AREA : REAL := 1.0;
      BV   : REAL := 100.0;
      C0   : REAL := 0.1E-9;
      CJO  : REAL := 0.4E-11;
      GMIN : REAL := 1.0E-12;
      IBV  : REAL := 0.1E-3;
      ISS  : REAL := 0.268E-8;
      R0   : REAL := 10.0E3;
      RS   : REAL := 0.6;
      k    : REAL := 0.138062E-22;
      q    : REAL := 0.160219E-18
   );
   PORT (
      TERMINAL A, C, G : ELECTRICAL
   );
END ENTITY diodeRC;
```

```
--
-- VHDL-AMS model of the diode cont’d (architecture dae)
--

ARCHITECTURE dae OF diodeRC IS

   QUANTITY VA ACROSS I$A THROUGH A;
   QUANTITY VC ACROSS I$C THROUGH C;
   QUANTITY VG ACROSS I$G THROUGH G;

   -- Simultaneous Vars
   QUANTITY V$C    : REAL;
   QUANTITY V$G    : REAL;
   QUANTITY id     : REAL;
   QUANTITY vd     : REAL;
   QUANTITY cj     : REAL;
   QUANTITY id$d1  : REAL;
   QUANTITY vd$d1  : REAL;
   QUANTITY V$C$d1 : REAL;
   QUANTITY V$G$d1 : REAL;

   -- If-functions
   FUNCTION iffunc1(
      vd : REAL) RETURN REAL IS
      VARIABLE res : REAL;
   BEGIN
      IF vd < 0.25 THEN
         res := 1.0/((1.0-2.0*vd)**(0.333));
      ELSE
         res := 0.251926E1*(0.3335+0.666*vd);
      END IF;
      RETURN res;
   END;

BEGIN

   -- Simultaneous Equations
   I$A+I$C+C0*(V$C$d1+V$G$d1)+(-V$C+V$G)/(R0) == 0.0 TOLERANCE "Current";
   I$G+C0*(V$C$d1-V$G$d1)+(V$C-V$G)/R0 == 0.0 TOLERANCE "Voltage";
   id+0.115E-7*id$d1+I$A+cj*vd$d1 == 0.0 TOLERANCE "Current";
   -VA+vd+V$C+I$A*RS/AREA == 0.0 TOLERANCE "Voltage";
   id-GMIN*vd-AREA*(ISS*(-1.0+exp(0.181069E-2*vd*q/k))
      -IBV*exp((-0.333167E-2)*(BV+vd)*q/k)) == 0.0 TOLERANCE "Current";
   cj-AREA*CJO*iffunc1(vd) == 0.0 TOLERANCE "Current";
   VC-V$C == 0.0 TOLERANCE "Voltage";
   VG-V$G == 0.0 TOLERANCE "Voltage";
   id$d1  == id'dot TOLERANCE "Current";
   vd$d1  == vd'dot TOLERANCE "Current";
   V$C$d1 == V$C'dot TOLERANCE "Current";
   V$G$d1 == V$G'dot TOLERANCE "Current";

END ARCHITECTURE dae;
```

```
//
// Verilog-A model of the diode circuit
//

`include "constants.h"
`include "discipline.h"

// Declaration for discipline DAEVar
nature daevar
   abstol = 1n;
   units = "";
   access = X;
endnature

nature daevar_flow
   abstol = 1n;
   units = "";
   access = Y;
endnature

discipline DAEVar
   potential daevar;
   flow daevar_flow;
   domain continuous;
enddiscipline

module diodeRC( A, C, G );
   inout A, C, G ;
   electrical A, C, G;

   // parameters
   parameter real AREA = 0.1E1;
   parameter real BV   = 0.1E3;
   parameter real C0   = 0.1E-9;
   parameter real CJO  = 0.4E-11;
   parameter real GMIN = 1.0E-12;
   parameter real IBV  = 0.1E-3;
   parameter real ISS  = 0.268E-8;
   parameter real R0   = 0.1E5;
   parameter real RS   = 0.6;
   parameter real k    = 0.13806226E-22;
   parameter real q    = 0.16021918E-18;

   // procedural variables
   real  vd;
   real  id;
   real  cj;

   // simultaneous variables
   DAEVar V_C;
   DAEVar V_G;
   DAEVar I_A;
   DAEVar id_d1;
   DAEVar vd_d1;
   DAEVar V_C_d1;
   DAEVar V_G_d1;
```

```
//
// Verilog-A model of the diode circuit
//

analog begin

   // Empty contributions (for topology checker)
   Y(I_A) <+ 0*X(I_A);
   Y(V_C) <+ 0*X(V_C);
   Y(V_G) <+ 0*X(V_G);

   // Procedural Equations
   vd = -RS*X(I_A)/AREA-X(V_C)+V(A);
   id = GMIN*vd+AREA*(-ISS+limexp((0.18106E-2*vd*q)/k))
        -IBV*limexp((-0.33316E-2*(BV+vd)*q)/k));
   cj = AREA*CJO*(vd < 0.25 ? 1/pow(1 -2*vd,0.333) :
                              0.251926E1*(0.3335 + 0.666*vd));

   // Simultaneous Equations
   Y(I_A)    <+ -id-0.115E-7*X(id_d1)+X(I_A)-cj*X(vd_d1);
   Y(V_C)    <+ X(V_C)-V(C);
   Y(V_G)    <+ X(V_G)-V(G);
   X(id_d1)  <+ ddt(id);
   X(vd_d1)  <+ ddt(vd);
   X(V_C_d1) <+ ddt(X(V_C));
   X(V_G_d1) <+ ddt(X(V_G));

   // Branch Equations
   I(A) <+ -X(I_A);
   I(C) <+ -X(I_A)+(X(V_C)-X(V_G))/R0+C0*(X(V_C_d1)-X(V_G_d1));
   I(G) <+ (X(V_G)-X(V_C))/R0+C0*(X(V_G_d1)-X(V_C_d1));
end
endmodule
```

A.3 emitter

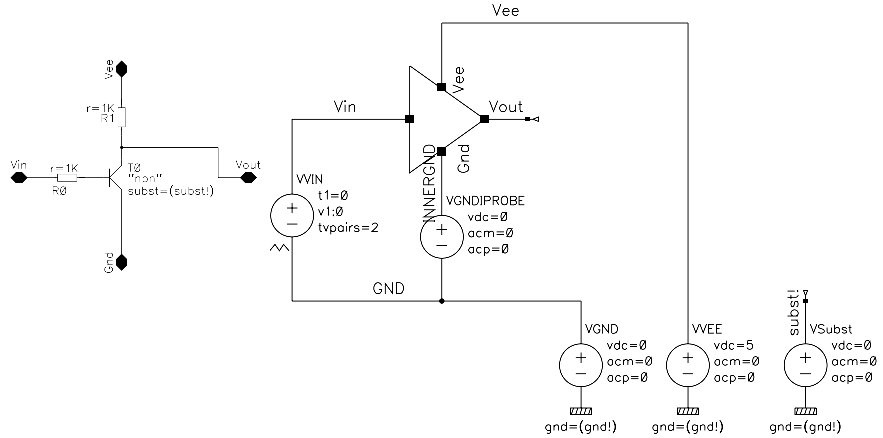


Figure A.5: Schematics of the *emitter* Circuit and its Testbench

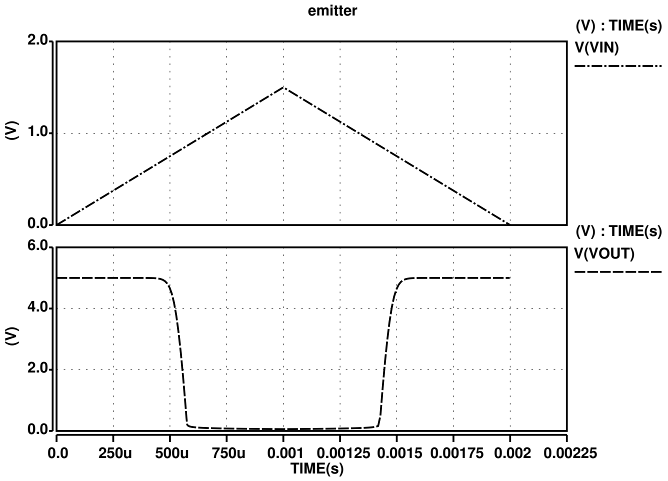


Figure A.6: Simulation Results of the *emitter* Circuit

A.4 multiplier

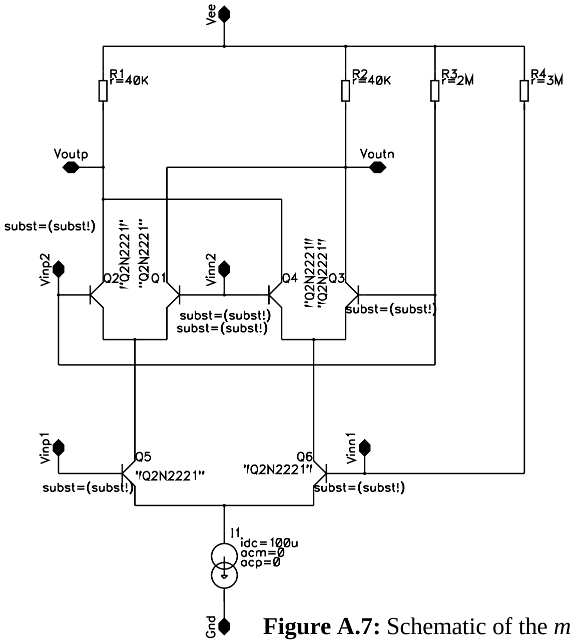


Figure A.7: Schematic of the *multiplier* Circuit

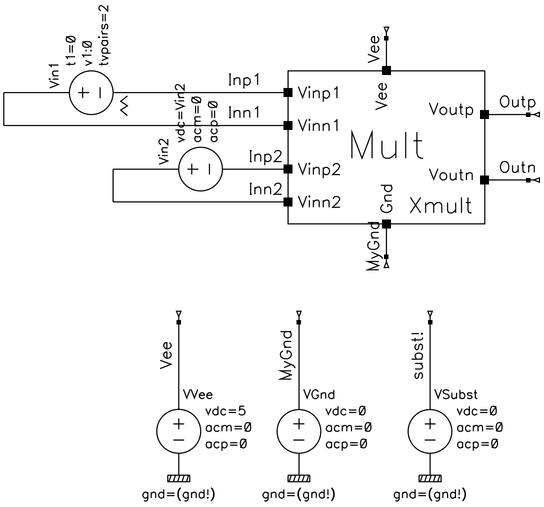


Figure A.8: Testbench for the *multiplier* Circuit

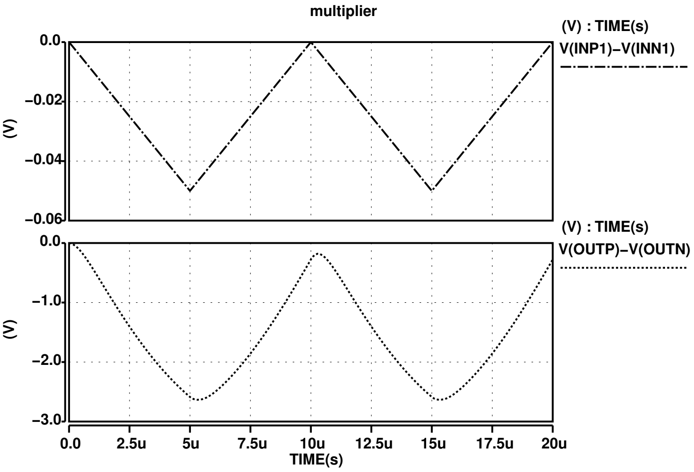


Figure A.9: Simulation Results of the *multiplier* Circuit

A.5 nand2

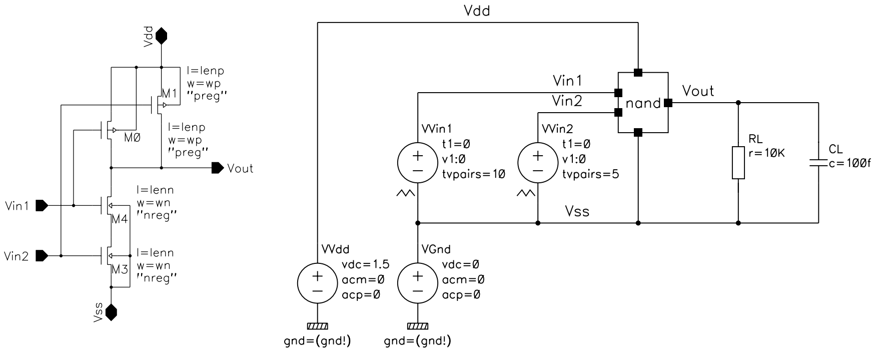


Figure A.10: Schematics of the *nand2* Circuit and its Testbench

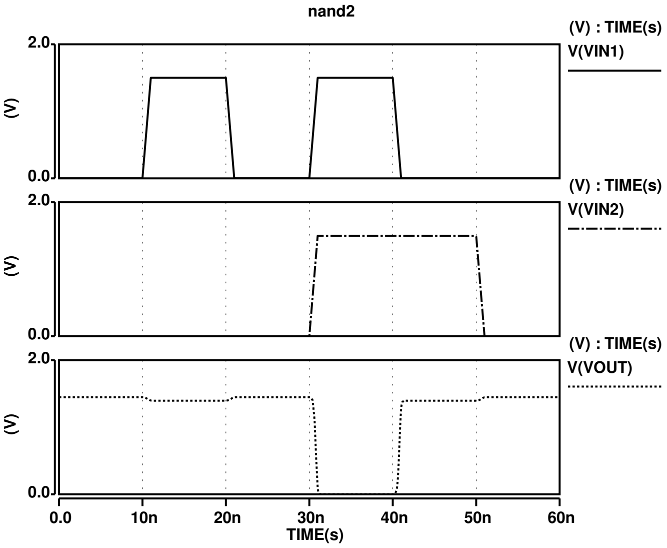


Figure A.11: Simulation Results of the *nand2* Circuit

A.6 opamp741

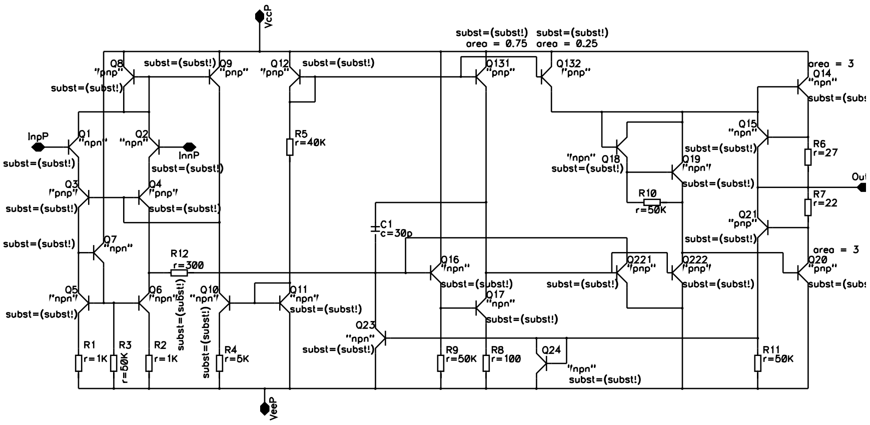


Figure A.12: Schematic of the *opamp741* Circuit

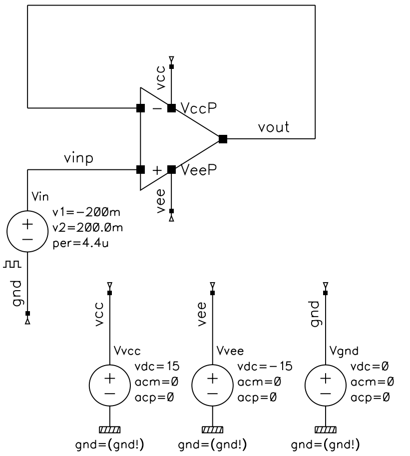


Figure A.13: Testbench for the *opamp741* Circuit

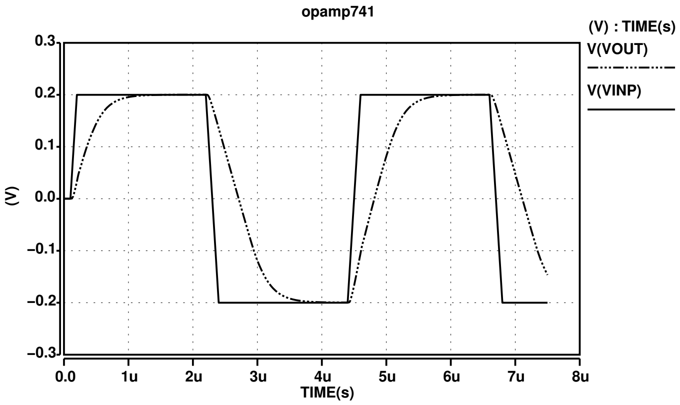


Figure A.14: Simulation Results of the *opamp741* Circuit

A.7 sqrt

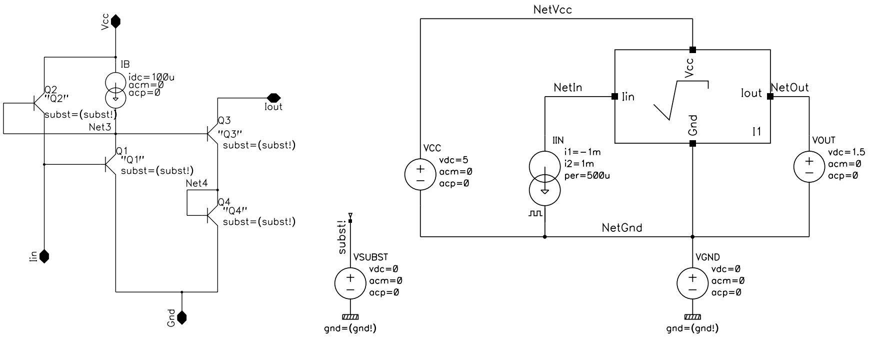


Figure A.15: Schematics of the *sqrt* Circuit and its Testbench

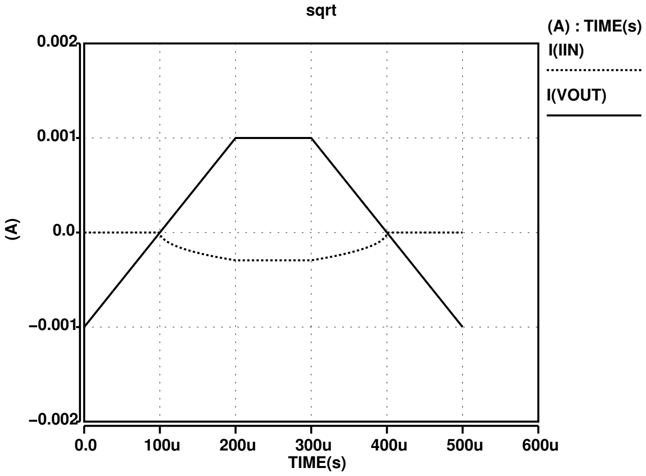


Figure A.16: Simulation Results the *sqrt* Circuit

A.8 stepmonitor

This Verilog-A model provides a basic solution for monitoring the convergence within a transient analysis for simulators supporting the $debug and $strobe functions. Including a reference of stepmonitor within the netlist (no connections) results in an additional logfile named <*netlist*>.stepmonitor containing information about the timesteps (length, absolute time, number of Newton iterations) and a final statistics (total steps, total iterations, average iterations per time step).

Note: The model extensively uses file-I/O to perform this statistics and therefore seriously slows down the simulation performance. Any performance measurements have to be performed in separate simulation runs without the stepmonitor instance.

```
//
// stepmonitor Verilog-A model
//
`include "constants.h"
`include "discipline.h"

module stepmonitor;

 integer LOGFILE, WRITEFILE, READFILE, DUMMY;   // File pointers
 integer eof;                               // EOF dependends on sim version..
 integer stepno, totaliterno;
 real    told, tstep, tabs;
 integer iters, dummy, readline, alternate;

 analog begin

  @(initial_step("tran")) begin
      eof = 0; dummy = 0;
      stepno = 0; iters = 0; totaliterno = 0;
      told = 0.0; alternate = 0; tabs = 0.0;

     // Opening files
      LOGFILE = $fopen("spectre_conv.log", "w");
      $fstrobe(LOGFILE, "=== stepmonitor logfile ===");

     // Touch the files needed for iteration debugging
      DUMMY = $fopen(".iterstep0", "w");
      $fstrobe(DUMMY, "0");
      $fclose(DUMMY);
      DUMMY = $fopen(".iterstep1", "w");
      $fstrobe(DUMMY, "0");
      $fclose(DUMMY);

      if (eof == 0) begin
         // Determine the eof-value
         READFILE = $fopen("%C", "r");
         readline = $fscanf(READFILE, "%c", dummy);
         if (readline == 0) eof = 1;                     // Spectre 5.1.
         else eof = -1;                                  // Spectre 6.1.
         $strobe("eof is: %d", eof);
         $fclose(READFILE);
      end
   end
```

```
//
// stepmonitor Verilog-A model cont’d
//

// Analysis within each timestep
   if (analysis("tran") && !analysis("ic")) begin
      // Open alternating files to determine iterations per timestep
      if (alternate <= 0) alternate=1;
      else alternate=0;

      // Determine iteration number from last file
      iters = -1;
      readline = 0;
      if (alternate <= 0)
        READFILE = $fopen(".iterstep1", "r");
      else
        READFILE = $fopen(".iterstep0", "r");

      // Count the lines within file
      while ($fscanf(READFILE, "%d", dummy) != eof) begin
         iters = iters + 1;
      end
      $fclose(READFILE);
      totaliterno = totaliterno + iters;
      $fstrobe(LOGFILE, "%d:\tabstime = %e\ttstep = %e\titers = %d", stepno,
               tabs, tstep, iters);

      // Calculate steplength and export step debugging
      stepno = stepno + 1;
      tabs = $abstime;
      tstep = tabs - told;
      told = tabs;

      // Count iterations to file
      if (alternate <= 0)
        WRITEFILE = $fopen(".iterstep0", "w");
      else
        WRITEFILE = $fopen(".iterstep1", "w");
      $fdebug(WRITEFILE, stepno);
      $fclose(WRITEFILE);
   end

   // Triggers logfile export at last timestep
   @(final_step("tran")) begin
      // Finale statistics
      $fstrobe(LOGFILE, "=== Overall Statistics ===");
      $fstrobe(LOGFILE, "Total iterations: %d", totaliterno);
      $fstrobe(LOGFILE, "Total steps:      %d", stepno);
      $fstrobe(LOGFILE, "Iterations/step:  %g", (1.0*totaliterno)/stepno);

      // Close all files
      $fclose(LOGFILE);
      $fclose(WRITEFILE);
   end
 end
endmodule
```

B Analog Insydes

This appendix provides additional information on new or enhanced functions within *Analog Insydes*. The first subsection covers topics related to model generation, the second subsection provides usage information on the DAE optimization functions, and the last subsection introduces some auxiliary functions.

B.1 Modeling Functions

B.1.1 Model Order Reduction

A DAE system

$$\boldsymbol{f}(\boldsymbol{x}_{diff}, \dot{\boldsymbol{x}}_{diff}, \boldsymbol{x}_{algebr}, t) = \boldsymbol{0}$$

can be transformed into an equivalent system

$$\tilde{\boldsymbol{f}}(\boldsymbol{x}_{diff}, \dot{\boldsymbol{x}}_{diff}, \boldsymbol{x}_{algebr}, \boldsymbol{x}_{sub}, t) = \boldsymbol{0} \text{ with}$$

$$\boldsymbol{x}_{sub} \in \mathbb{R}^{n_{diff}} \qquad \text{substitution variables}$$

by adding dummy equations $\boldsymbol{f}_{sub}$ of the form

$$\boldsymbol{f}_{sub}(\dot{\boldsymbol{x}}_{diff}, \boldsymbol{x}_{sub}) = \boldsymbol{x}_{sub} - \dot{\boldsymbol{x}}_{diff} = 0$$

and substitution of $\dot{\boldsymbol{x}}_{diff} \rightarrow \boldsymbol{x}_{sub}$ within the differential equations $\boldsymbol{f}_{diff}$. Recursive application is suited for order reduction of higher order DAE systems to first-order. The structure of the Jacobian matrix changes to

$$\tilde{\boldsymbol{J}} = \frac{\partial}{\partial \tilde{\boldsymbol{x}}}\tilde{\boldsymbol{f}} = \begin{bmatrix} \dfrac{\partial \boldsymbol{f}_{diff}}{\partial \boldsymbol{x}_{diff}} & \dfrac{\partial \boldsymbol{f}_{diff}}{\partial \boldsymbol{x}_{algebr}} & \dfrac{\partial \boldsymbol{f}_{diff}}{\partial \boldsymbol{x}_{sub}} \\ \dfrac{\partial \boldsymbol{f}_{algebr}}{\partial \boldsymbol{x}_{diff}} & \dfrac{\partial \boldsymbol{f}_{algebr}}{\partial \boldsymbol{x}_{algebr}} & \boldsymbol{0} \\ -\boldsymbol{1} & \boldsymbol{0} & \boldsymbol{1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{J}_{stat,\,11} & \boldsymbol{J}_{stat,\,12} & \boldsymbol{J}_{stat,\,13} \\ \boldsymbol{J}_{stat,\,21} & \boldsymbol{J}_{stat,\,22} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{1} \end{bmatrix} + \alpha \begin{bmatrix} \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ -\boldsymbol{1} & \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \text{ with}$$

$$\tilde{\boldsymbol{J}} \in \mathbb{R}^{2n_{diff} + n_{algebr}} \times \mathbb{R}^{2n_{diff} + n_{algebr}}.$$

The described method is realized in *Analog Insydes* through the function `ToFirstOrderSystem` and is applied to the DAE systems before the model generation.

B.1.2 WriteModel

During this work, the original `WriteModel` function has been extended and enhanced by several modeling languages (TML, ZMS, CMI) and features. Especially the idea of providing different modeling strategies for certain features as well as the generation of simulator-specific models have been integrated into the modeling function. Thus, the function is able to generate specific “flavors” of the AHDLs adapted to the simulators needs. The latter depend on supported language features as well as performance and robustness aspects. The model export is used as follows:

```
WriteModel[
    modelfile,                (* Output file for model *)
    entityname,               (* Entity / module name of model *)
    DAEObject,                (* DAEObject to model *)
    ports,                    (* List of model ports *)
    connections,              (* Connection information for ports / variables *)
    (* Mandatory “options” *)
    ModelingLanguage          -> ”VHDL-AMS”|”Verilog-A”|”TML”|”ZMS”|...,
    Simulator                 -> ”Titan”|...,

    Options                   (* Additional options*)
]
```

Table B.1: New Options of `WriteModel`

Option	*Values*	*Description*
`CSE`	`True`|`False`	Internally performs CSE on Jacobian matrix and functions (valid only for *Titan* ZMS models)
`Damping`	`True`|`False`	Replaces several nonlinear functions by damping functions (e.g. limexp, limsqrt)
`IfStrategy`	see Table B.2	Distinguishes the modeling strategy for conditional statements
`InsertTolerances`	`Automatic`| `True`|`False`	Tries to heuristically determine tolerances of variables and equations (if applicable to modeling language)
`Preloading`	`True`|`False`	Applies preloading strategies within the model export (applicable for Verilog-A and *Titan* ZMS)
`SequentialStrategy`	see Table B.3	Distinguishes the modeling strategy for sequential equations

Table B.1 provides the new options of `WriteModel` that have been introduced to generate simulator-specific models. Unfortunately, the modeling methods are limited by the support of certain language features on the simulator side. Whereas simultaneous equations can be very intuitively modeled in VHDL-AMS, the language is of major disadvantage for sequential equations. The (most desirable) simultaneous procedural statement is not yet supported by any of the (to the author available) simulators. Using analog functions to realize sequential equations requires the sequential variable to be a free quantity (unnecessarily) increasing the number of unknowns of the resulting system of equations (otherwise, derivatives of sequential variables would not be possible). This makes modeling of sequential equations utterly impossible in VHDL-AMS as it is not possible to reduce the dimension of the linear system.

In contrast to that, modeling sequential equations is very comfortable in Verilog-A. Though modeling simultaneous equations requires some effort as there is no direct representation of an equality in Verilog-A. Therefore, their modeling is realized through an indirect branch contribution. Unfortunately, this seems to significantly limit the simulation performance.

Initial values and tolerances are not supported by some simulators. In order to enable the modeling of conditional statements and sequential equations, several modeling strategies have been introduced. Tables B.2 and B.3 introduce the implemented modeling strategies.

Table B.2: Option Values for `IfStrategy`

Option Value	***Description***
`ConditionalOperator`	Use of the ternary operator to model conditional statements (applicable for Verilog-A only)
`Function`	Generation of a function for each conditional statement and subsequent calls of the function from within the equation set (applicable for MAST, Verilog-A, VHDL-AMS)
`SimultaneousCondition`	Introduction of an additional variable and equation per conditional statement, substitution of all conditional statements by the newly introduced variable (applicable for TML and VHDL-AMS)
`UnitStep`	Replacement of conditional statements by a sum of two inverse unitstep functions multiplied with either of the branch equations (applicable for TML and VHDL-AMS)

The probably most comfortable modeling strategy for conditional statements is the ternary operator of Verilog-A. Anyway, there are several restrictions on what language features might be used within the ternary operator and functions. The `SimultaneousCondition` option has the disadvantage of introducing additional variables. Finally, `UnitStep` might cause numerical problems to do “unannounced” discontinuities, but experiments have shown good convergence using this feature.

Table B.3: Option Values for `SequentialStrategy`

Option Value	*Description*
`None`	All equations are modeled simultaneously
`Function`	Generation of a function, an additional free quantity, and an simultaneous statement for each sequential equation (applicable for VHDL-AMS)
`SequentialAssignment`	Modeling of sequential equations by direct assignments (Verilog-A: procedural statement, VHDL-AMS: simultaneous procedural statement)

The modeling strategies for sequential equations are a crucial point when generating VHDL-AMS models as was described above. The option’s value `SequentialAssignment` would be the best modeling strategy, but is currently not supported by simulators. The strategy activated by the option value `Function` does not enhance the performance of the simulation as no reduction of the models’ dimension can be achieved.

B.1.3 WritePincompatibleModel

`WritePincompatibleModel` is a wrapper function for `WriteModel`. It is intended to bottom-up generate behavioral models starting from a circuit netlist. The function contains a complete behavioral modeling flow to import the circuit, extract the subblock to be modeled, setup the circuit equations, optimize them for numerical methods, reduce the complexity (a future issue), and finally export a behavioral model in one of the available analog hardware description languages. As the process allows a diversity of different modeling strategies and settings, `WritePincompatibleModel` tries to determine the best settings for your simulator automatically. The function’s usage:

```
WritePincompatibleModel[
    netlistfile,                (* Netlist to generate model from *)
    instancename,               (* Instance name of subcircuit to model*)
    modelfile,                  (* Output file for model *)
    entityname,                 (* Entity / module name of model *)
    (* Mandatory “options” *)
    ModelingLanguage            -> ”VHDL-AMS”|”Verilog-A”|”TML”|”ZMS”|...,
    Simulator                   -> ”Titan”|...,
    CircuitSimulator            -> ”Titan”|”AnalogArtist”,

    Options                     (* Additional options*)
]
```

The combination of `ModelingLanguage` and `Simulator` allows to generate models in various modeling languages and to adapt the optimization as well as modeling strategies to the specified target simulator (names not provided due to anonymous presentation).

Table B.4: Additional Options of `WritePincompatibleModel`

Option	***Values***	***Description***
`EquationFormulation`	“MNA”|”STA”	Selection of a formulation of the network equations
`CommonSubexpressions`	`True|False`	Apply CSE
`CompressModelEquations`	`True|False`	Compress model equations using `CompressEquations`
`MarkAllSimultaneous`	`True|False`	Discard all information on sequential equations (all equations modeled simultaneously)
`RemoveTrivialEquations`	`True|False`	Reduce redundancy by removing trivial equations
`SequentialIdentification`	`True|False`	Apply BLT
`SequentialSubstitution`	`True|False`	Substitutes all sequential equations
`SubstituteParameters`	`True|False`	Substitutes all parameters with their numerical default values
`ProvideInitialGuess`	`True|False`	Perform a DC analysis and provide results as initial values for the model
`Lisfile`	File name	Import the logfile of Titan to extract DC/AC values
`<Subfunction>Options`	List of options for specific subfunctions	These options allow to pass user-specified options to all of the major subfunctions within the modeling process (e.g. `WriteModelOptions`)

The option `WriteModelOptions` is particularly helpful to select a non-default modeling strategy within the bottom-up modeling flow. It enables the user to customize the model export as the provided options are passed to `WriteModel`.

B.2 DAE Optimization Functions

B.2.1 IdentifySequentialEquations

`IdentifySequentialEquations[`DAEObject, Options**`]`** performs BLT on a given DAEObject and returns an optimized DAEObject with additional sequential equations.

Table B.1: Options of `IdentifySequentialEquations`

Option	*Values*	*Description*
`ChooseMethod`	“Version1”|”Version2”	Switches between two different algorithms. Version2 is more performant.
`KeepSequentialEquations`	`True|False`	Keep or discard original sequential equations of the input DAEObject
`InitialSimultaneous-Variables`	List of variables	Keeps this variables simultaneous
`KeepPivotElementsForSimeqs`	`True|False`	Keeps pivot elements within the simultaneous equations, typically this is not necessary
`KeepDecoupledSequential-Blocks`	`True|False`	Does not recognize sequential equations that couple independent sequential blocks (supported only by Version1)

B.2.2 OptimizeCommonSubexpressions

`OptimizeCommonSubexpressions[`DAEObject, Options**`]`** performs a common subexpression elimination on the DAEObject. It finds common subexpressions and extracts these expressions into additional sequential equations to avoid unnecessary multiple evaluation of common expressions.

Table B.2: Options of `OptimizeCommonSubexpressions`

Option	*Values*	*Description*
`ChooseMethod`	“Version1”|”Version2”	Switches between two different algorithms. Version1 is more performant and yields a lower number of deeper expressions.
`ExtractParametric-Expressions`	`True|False`	Additionally extracts expressions only consisting of parameters (for preloading)
`ExpandSubexpressions`	`True|False`	Expands the found common subexpressions to maximum depth
`MinCost`	Integer (default: 3)	Minimum cost of a subexpression
`MinDepth`	Integer (default: 3)	Minimum depth of a subexpression
`MinUsage`	Integer (default: 2)	Minimum number of references to a subexpression

The latter three options may result in discarding a found expression. A common subexpression has to be compliant with all three conditions. When used together with `ExtractParametricExpressions->True`, `MinUsage` does not apply.

B.2.3 RemoveRedundantEquations

`RemoveRedundantEquations[``DAEObject`, Options**]** removes sequential equations that are not needed within the equation structure. The option `ProtectedVariables` allows to provide a list of variables that are protected from being removed.

B.2.4 RemoveTrivialEquations

`RemoveTrivialEquations[``DAEObject`, Options**]** removes redundant (trivial) equations and thereby reduces the dimension of the DAE system. The option `ProtectedVariables` allows to provide a list of variables that are protected from being substituted.

B.2.5 SubstituteSequentialEquations

`SubstituteSequentialEquations[``DAEObject`, Options**]** conditionally replaces sequential variables by their determining sequential equation. Sequential equations violating one of the conditions (options `MinCost`, `MinUsage`, `MinDepth`) are substituted. With default options, the function reduces the dimension to the number of simultaneous variables. Use carefully, as the resulting equation sets might be of enormous complexity.

Table B.3: Options of `SubstituteSequentialEquations`

Option	*Values*	*Description*
`KeepParametricExpressions`	`True`|`False`	Protects parametric expressions from being substituted
`MinCost`	Integer (default: 1)	Minimum cost of a sequential equation
`MinDepth`	Integer (default: 1)	Minimum depth of a sequential equation
`MinUsage`	Integer (default: 1)	Minimum number of references to a sequential variable
`ProtectedVariables`	List of variables	List of sequential variables that are protected from being substituted

B.3 Supplementary Functions

This section introduces some new supplementary functions related to the modeling and optimization process.

B.3.1 AlgebraicDifferentialPartitioning

`AlgebraicDifferentialPartitioning``[DAEObject]` performs a partitioning of the equation set of a DAEObject into algebraic and differential subsystems.

B.3.2 CheckDAEConsistency

`CheckDAEConsistency``[DAEObject]` performs a set of basic consistency checks to make sure a DAEObject does contain a valid DAE system. The function checks:

- Block sizes
- Equation / Variable numbers
- Structure of sequential equations (explicit formulation, lower-diagonal block)
- Independency of sequential subsystems
- First-order system

B.3.3 DAEStatistics

The function **`DAEStatistics`**`[DAEObject]` generates detailed statistics on a DAE system:

• Basic Statistics	Number of equations and variables, differential variables, etc.
• Sequential Statistics	Number of seq. and sim. vars. / eqs., sparsity, nonzeros

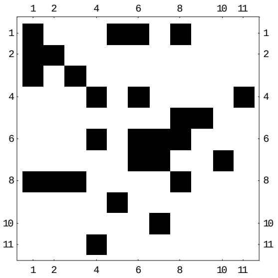
• Nonlinear Statistics	Number of function calls to nonlinear functions
• Evaluation Cost	Evaluation cost for functions and their Jacobian matrix
• Memory Accesses	Memory accesses for functions and their Jacobian matrix

B.3.4 DisplayEquations / DisplayEquationCosts

`DisplayEquations[``DAEObject`**`]`** shows a well-formatted list of the DAE system (also including the partitioning into seq. and sim. equations). **`DisplayEquationCosts`** also includes cost, depth, and usage of the equations.

B.3.5 DisplayLinearSystem / DisplayLinearSystemStructure

`DisplayLinearSystem[``DAEObject`**`]`** generates a symbolic representation of the linearized system of the DAEs (with Backward Euler applied to resolve differential variables). **`DisplayLinearSystemStructure[`**`DAEObject`, Options**`]`** returns a plot of the structural nonzero entries of the system's Jacobian matrix (as shown within the figure). The boolean options `DisplayBlockBorders` and `DisplaySeparators` allow to turn of auxiliary lines to separate the subsystems.



B.3.6 EvaluationCost

The function **`EvaluationCost[`**`DAEObject`, Options**`]`**estimates the effort necessary for the numerical evaluation of the functions resp. the Jacobian matrix of the DAEObject. The options allow to measure in different modes and display the result in different metrics. The estimated CPI figures can be customized by using the option `FunctionCosts`.

Table B.1: Options of `EvaluationCost`

Option	***Values***	***Description***
`MeasureMode`	“Function“| ”Jacobian”|”All”	Account for function only, Jacobian matrix only, or both
`Metric`	“FunctionCalls”| “Flops”|”TotalFlops”	Display results in calls per function, flops per function, or total flops
`FunctionCosts`	List of function->cost	Customize the function costs

B.3.7 OptimizeEquations

`OptimizeEquations[``DAEObject`**`]`** applies various optimization strategies to the DAE system and returns an optimized system of equations.

B.3.8 ReadVerilogA (Prototype)

The function **`ReadVerilogA[`**modelfilename**`]`** is a prototypical approach to import the equations of a Verilog-A model into *Analog Insydes* for further processing. As it does not contain a Verilog-A parser, the process relies on the *adms* model compiler [44] to convert the Verilog-A model to a *Mathematica* conform syntax. This intermediate file is afterwards imported and postprocessed to obtain a DAEObject. The postprocessing requires comprehensive reformulations of procedural statements to obtain valid sequential equations. To name only the major problems there are:

- Syntax conversion
- Unbalanced conditional statements
- Nested conditional statements
- Multiple procedural assignments to the same variable
- Redundant equations
- Conversion of branch representation to network equations

The function has been successfully applied to import a Verilog-A implementation of the Gummel-Poon model. Based on the imported equation set, a symbolic device model for *Analog Insydes* was generated. The future application of this function includes several different use cases: Generation of *Analog Insydes* device models, model translation (e.g. Verilog-A to VHDL-AMS), model optimization, as well as device model compilation (e.g. Verilog-A to *Titan* ZMS). In order to enable such promising applications, further enhancements have to be done to provide further necessary features. However, the prototypical application to the Gummel-Poon model has been the proof-of-concept for the Verilog-A import functionality.

C Additional Statistics

C.1 Loading Performance

This section contains the charts for the complete and chain network experiments for the simulators *Dione*, *Rhea*, *Thetys* (for the corresponding results for *Titan* refer to Section 4.6).

C.1.1 Dione (Verilog-A)

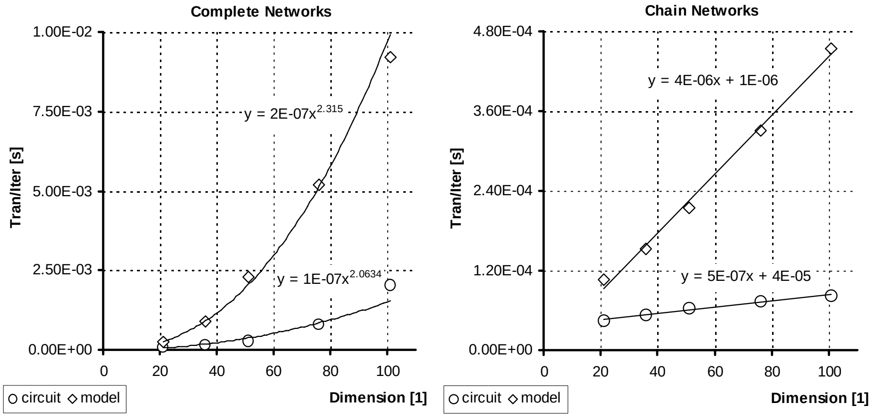


Figure C.1: CPU Time for Complete (left) and Chain Networks (right) for *Dione*

C.1.2 Rhea (Verilog-A)

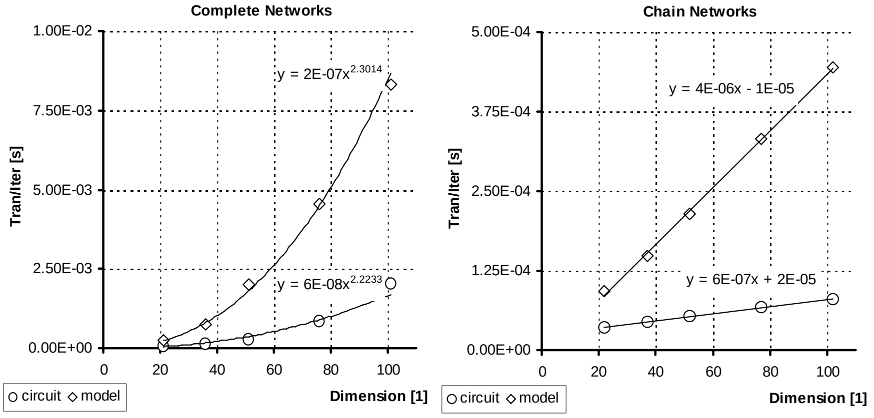


Figure C.2: CPU Time for Complete (left) and Chain Networks (right) for *Rhea*

This statistics for *Rhea* could not be performed with the VHDL-AMS front-end of the simulator as the compiler crashes due to insufficient memory for the majority of the generated models.

C.1.3 Thetys (VHDL-AMS)

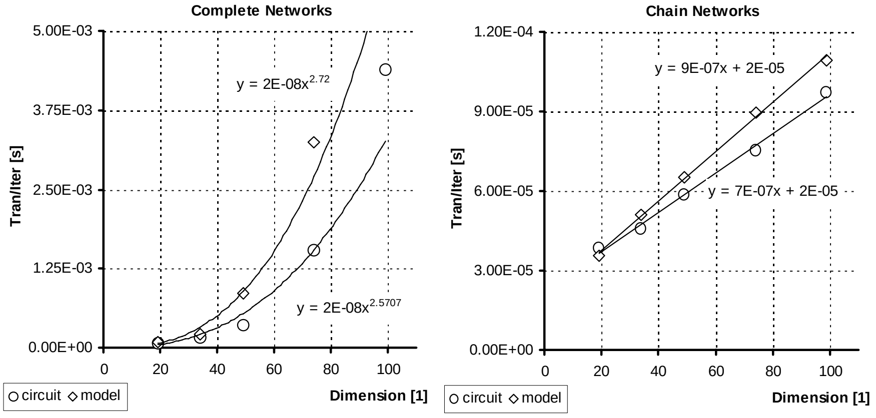


Figure C.3: CPU Time for Complete (left) and Chain Networks (right) for *Thetys*

C.2 Sparse Loading Performance

C.2.4 Titan (TML)

Table C.1: Evaluation of the Sparse Loading for *opamp741*

Example		$N_{iter/step}$	T_{tran}	S_{tran}	T_{load}	S_{load}	T_{solve}	S_{solve}
model (dense)	absolute	3.49	77.52 s	10.0	49.36 s	7.2	27.84 s	43.5
	relative	n/a	100 %		63.6 %		35.9 %	
model (sparse)	absolute	4.64	7.73 s		6.85 s		0.64 s	
	relative	n/a	100 %		88.6 %		8.3 %	

Figure C.4: Additional charts to show the distribution of the CPU time to loading and solving for the sparse implementation for *Titan*. Refer to Section 5.2 for further details.

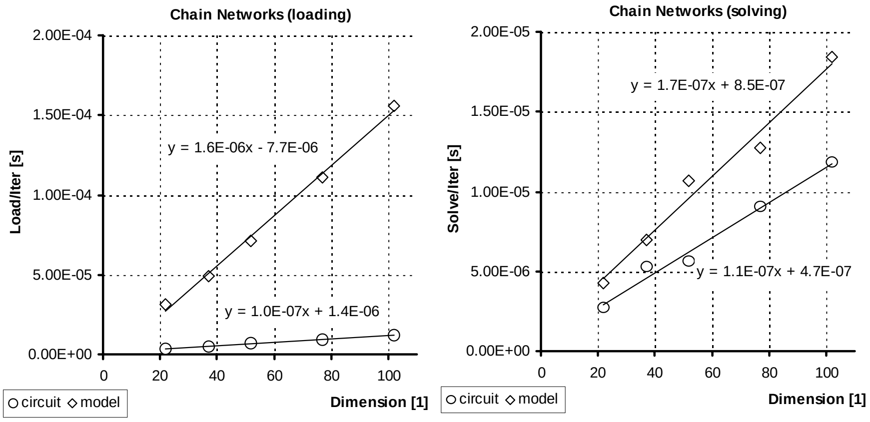


Figure C.4: CPU Time for Chain Networks with *Titan* (TML, incl. Sparse Loading). CPU time for loading (left) and solving (right)

C.2.5 Titan (ZMS)

Figure C.5 / Figure C.6: Additional charts to show the CPU time for the ZMS-based models for *Titan*. Refer to Section 5.7 for further details and the charts for the loading performance. The CPU time for the transient analysis (left chart) is dominated by the solving performance (right chart), which is suboptimal due to application of the *MUMPS* solver (missing integration for the sparse *Titan* solver).

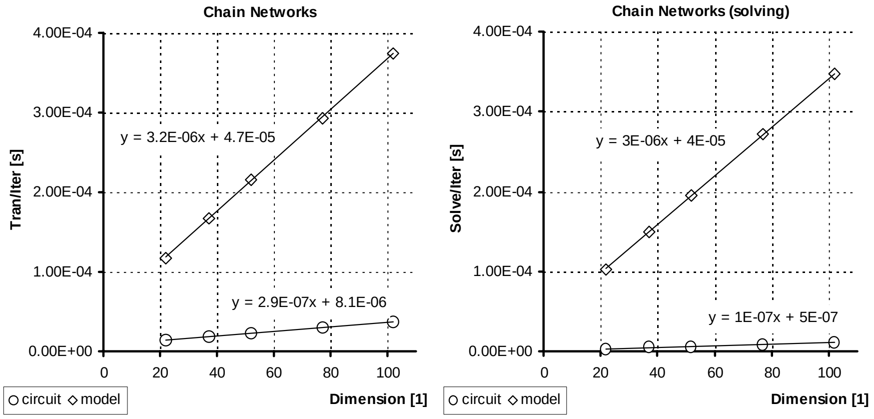


Figure C.5: CPU Time for Chain Networks with *Titan* (ZMS, MUMPS Solver) for transient analysis (left) and solving (right)

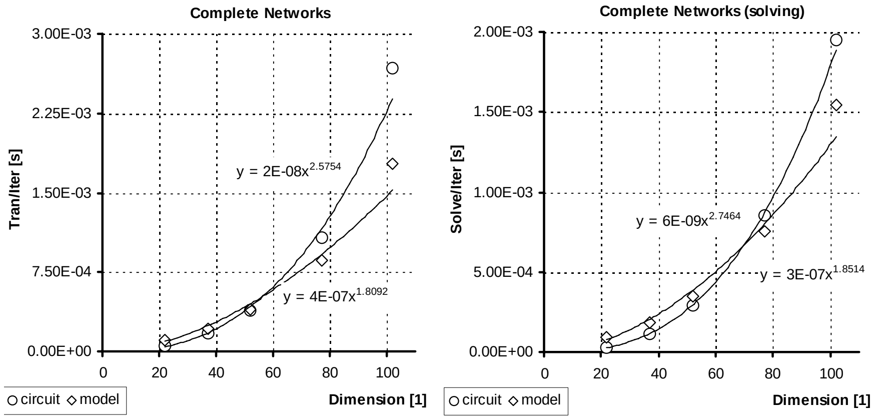


Figure C.6: CPU Time for Complete Networks with *Titan* (ZMS, MUMPS Solver) for transient analysis (left) and solving (right)

Bibliography

[1] A. Aho, M. Lam, R. Sethi, J. D. Ullman, “Compilers: Principles, Techniques, & Tools”, 2nd Edition, Pearson Education Inc., Boston, 2007

[2] Analog Insydes (Fraunhofer ITWM): `www.analog-insydes.de`

[3] C. Borchers, “Automatische Generierung von Verhaltensmodellen für nichtlineare Analogschaltungen“, Doctoral Thesis, Fortschritt-Berichte VDI Reihe 20, VDI-Verlag, 1997

[4] C. Borchers, L. Hedrich, E. Barke, “Equation-Based Behavioral Model Generation for Nonlinear Analog Circuits”, Proc. 33rd Design Automation Conference (DAC), pp. 237-240,1996

[5] C. Borchers, “Symbolic Behavioral Model Generation of Nonlinear Analog Circuits”, IEEE Trans. on Circuits and Systems II: Analog and Digital Signal Processing, Vol. 45, pp. 1362-1371, Oct 1998

[6] S. Böhme, C. Clauß, R. Jancke, P. Trappe, P. Schwarz, T. Halfmann, R. Sommer, “Modellierungsunterstützung für Mixed-Signal-Systeme durch symbolische Vereinfachung nichtlinearer Blöcke“, ITG/GMM ANALOG 2005, Feb. 2005

[7] J. Broz, T. Halfmann, R. Sommer, “Symbolische Analyse und Reduktion Multi-Physikalischer Systeme”, 6. GI/GMM/ITG-Workshop Multi-Nature Systems: Entwicklung von Systemen mit elektronischen und nichtelektronischen Komponenten, Erfurt/Germany, Feb. 2007

[8] J. Broz, C. Clauss, T. Halfmann, P. Lang, R. Martin, P. Schwarz, “Automated Symbolic Model Reduction for Mechatronical Systems”, Proc. of the IEEE International Symposium on Computer-Aided Control Systems Design, Munich/Germany, pp. 408-415, Oct. 2006

[9] G. R. Boyle, D. O. Pederson, B. M. Cohn, J. E. Solomon, “Macromodeling of Integrated Circuit Operational Amplifiers“, IEEE Journal of Solid-State Circuits, Vol. 9, No. 6, pp. 353-364, Dec. 1974

[10] G. Casinovi and A. Sangiovanni-Vincentelli, “A Macromodeling Algorithm for Analog Circuits”, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol. 10, No. 2, pp. 150-160, Feb. 1991

[11] G. Casinovi, J. Yang, “Multi-Level Simulation of Large Analog Systems Containing Behavioral Models”, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol. 13, pp. 1391-1399, Nov 1994

[12] G. Casinovi, J. Yang, “Simulation of Analog Behavioral Models”, Proc. of the IEEE Custom Integrated Circuits Conference (CICC), pp. 12.4.1-12.4.4, May 1992

[13] V. Chaudhary, M. Francis, W. Zheng, A. Mantooth, L. Lemaitre, “Automatic Generation of Compact Semiconductor Device Models using Paragon and ADMS”, IEEE International Behavioral Modeling and Simulation Workshop (BMAS), pp. 107-112, San Jose/USA, Oct. 2004

[14] L. O. Chua, P.-M. Lin, “Computer-Aided Analysis of Electronic Circuits”, Prentice-Hall Inc., 1975

[15] G. J. Coram, “How to (and how not to) Write a Compact Model in Verilog-A“, Proc. of the IEEE Intern. Behavioral Modeling and Simulation Conference (BMAS), pp. 97-106, Oct. 2004

[16] G. Denk, U. Feldmann, C. Hammer, M. Kahlert, R. Neubert, G. Reißig, A. Windisch, “Erweiterung eines Standard-Schaltungssimulators in Richtung VHDL-AMS”, ITG/GMM ANALOG 1999, Feb. 1999

[17] S. Doboli, G. Gothoskar, A. Doboli, “Piecewise-Linear Modeling of Analog Circuits Based on Model Extraction from Trained Neural Networks“, Proc. of the International Workshop on Behavioral Modeling and Simulation (BMAS), pp. 41-46, Oct. 2003

[18] I. S. Duff, A. M. Erisman, J. K. Reid, “Direct Methods for Sparse Matrices”, Oxford University Press, 1989

[19] J. Eckmüller, M. Gröpl, H. Gräb, “Hierarchical Characterization of Analog Integrated CMOS Circuits”, Proc. of the IEEE Design Automation and Test in Europe (DATE), pp. 636-643, Paris/France, Feb. 1998

[20] H. Elmqvist, M. Otter, F. E. Cellier, “Inline Integration: A New Mixed Symbolic/Numeric Approach for Solving Differential-Algebraic Equation Systems“, Proc. of the European Simulation Multiconference, pp. XXIII-XXXIV, June 1995

[21] H. Elmqvist, M. Otter, “Methods for Tearing Systems of Equations in Object-Oriented Modeling”, Proc. of the European Simulation Multiconference, Barcelona/Spain, June 1994

[22] D. Estévez Schwarz, U. Feldmann, R. März, S. Sturtzel, C. Tischendorf, “Finding Beneficial DAE Structures in Circuit Simulation”, in W. Jaeger, H.-J. Krebs (Eds): “Mathematics – Key Technology for the Future”, Springer 2002

[23] D. Estévez Schwarz, “Consistent Initialization for Index-2 Differential Algebraic Equations and its Application to Circuit Simulation”, Doctoral Thesis, Humboldt-Universität zu Berlin, 2000

[24] U. Feldmann, R. Schultz, U. A. Wever, H. Wriedt, Q. Zheng, “Algorithms for Modern Circuit Simulation”, Arch. Elektron. & Uebertragungstech., Vol. 46, pp. 274-285, No. 4, 1992

[25] N. Fröhlich, V. Glöckel, J. Fleischmann, “A New Partitioning Method for Parallel Simulation of VLSI Circuits on Transistor Level”, Proc. of the IEEE Design, Automation and Test in Europe (DATE), pp. 679-685, Paris/France, 2000

[26] C. Gathercole, H.A. Mantooth, “Pole-zero Localization: A Behavioral Modeling Approach“, Proceedings of the Fifth IEEE Intern. Workshop on Behavioral Modeling and Simulation (BMAS), pp. 59-65, Santa Rose/USA, Oct 2001

[27] G. Gielen, W. M. C. Sansen, “Symbolic Analysis for Automated Design of Analog Integrated Circuits”, Kluwer Academic Publishers, Norwell, 1991

[28] D. Grabowski, C. Grimm, E. Barke, “Semi-Symbolic Modeling and Simulation of Circuits and Systems”, Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS), Kos/Greece, May 2006

[29] D. Grabowski, M. Olbrich, E. Barke, “Analog Circuit Simulation Using Range Arithmetics”, Proc. of the 13th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 762-767, Seoul/Korea, Jan. 2008

[30] J. L. Hennessy, D. A. Patterson, “Computer Architecture, A Quantitative Approach“, 4th edition, Morgan Kaufmann Publishers, 2007

[31] E. Hennig, “Symbolic Approximation and Modeling Techniques for Analysis and Design of Analog Circuits”, Doctoral Thesis, Shaker Verlag, Apr. 2000

[32] W. Heupke, C. Grimm, K. Waldschmidt, “A New Method for Modeling and Analysis of Accuracy and Tolerances in Mixed-Signal Systems”, Proc. of the Forum on Design Languages (FDL), Frankfurt/Germany, Sep. 2003

[33] K. Hofmann, “Differential Model Generation for Microsystem Components Using Analog Hardware Description Languages“, Doctoral Thesis, Disserations Druck Darmstadt, Oct. 1997

[34] M. Honkala, J. Roos, M. Valtonen, “New Multilevel Newton-Raphson Method for Parallel Circuit Simulation”, Proc. of the 15th European Conference on Circuit Theory and Design (ECCTD), pp. II-113-II-116, Espoo/Finland, Aug 2001

[35] B. Hu, C. Wakayama, L. Zhou, C.-J. R. Shi, “Developing Device Models“, IEEE Circuits and Devices Magazine, Vol. 21, No. 4, pp. 6-11, Jul. 2005

[36] R. Jancke, S. Böhme, C. Clauß, T. Halfmann, P. Schwarz, R. Sommer, “Modellierungsunterstützung für Mixed-Signal-Systeme durch symbolische Vereinfachung nichtlinearer Blöcke“, GMM/ITG ANALOG 2005, Mar. 2005

[37] R. Jancke, P. Schwarz, “Supporting Analog Synthesis by Abstracting Circuit Behavior Using a Modeling Methodology”, Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS), Kos/Greece, May 2006

[38] B. P. Hu, G. Shi, C.-J. R. Shi, “Symbolic Model Order Reduction“, Proc. of the International Workshop on Behavioral Modeling and Simulation (BMAS), pp. 34-40, Oct. 2003

[39] K. S. Kundert, O. Zinke, “The Designer’s Guide to SPICE & SPECTRE“, Springer Science+Business Media, 2004

[40] K. S. Kundert, “Introduction to RF Simulation and its Application”, IEEE Journal of Solid-State Circuits, Vol. 34, No. 9, pp. 1298-1319, Sep. 1999

[41] K. S. Kundert, “Principles of Top-Down Mixed-Signal Design“, `www.designers-guide.org`, Feb. 2003

[42] K. Kundert, H. Chang, “Top-Down Design and Verification of Mixed-Signal Circuits“,
`www.designers-guide.com`, Jun. 2005

[43] H. Chang, K. Kundert, “Verification of Complex Analog and RF IC Designs“, `www.designers-guide.com`, Feb. 2007

[44] L. Lemaitre, C. McAndrew, S. Hamm, “ADMS-Automatic Device Model Synthesizer”, Proc. of the IEEE Custom Integrated Circuits Conference (CICC), pp. 27-30, May 2002

[45] Linear Algebra PACKage (LAPACK): `http://www.netlib.org/lapack/`

[46] J. Mades, “Strukturelle Konsistenz und Regularisierung von VHDL-AMS-Modellen”, Doctoral Thesis, Shaker Verlag, 2003

[47] J. Mades, T. Schneider, M. Glesner, A. Windisch, W. Ecker, “A JAVA-Based Mixed-Signal Design Environment“, Proc. of the XIII Symposium on Integrated Circuit and System Design, p. 301, 2000, Manaus/Brazil

[48] H. A. Mantooth, M. F. Fiegenbaum, “Modeling With an Analog Hardware Description Language”, Kluwer Academic Publishers, 1995

[49] Mathematica (Wolfram Reseach, Inc.): `http://www.wolfram.com/`

[50] W. Mathis, “Theorie nichtlinearer Netzwerke”, Springer Verlag, Berlin, 1987

[51] S. E. Mattsson, M. Otter, E. Hilding, “Modelica Hybrid Modeling and Efficient Simulation”, Proc. of the 38th IEEE Conference on Decision and Control, Vol. 4, pp. 3502-3507, Phoenix/USA, Dec 1999

[52] H. Mielenz, R. Dölling, “Automatic Identification Method for Analog and Mixed Analog/Digital ICs in Automotive Electronics”, International Automobile Conference, Stuttgart/Germany, June 2004

[53] O. Mikulchenko, K. Mayaram, “Neural Network Design for Behavioral Model Generation with Shape Preserving Properties“, Proc. of the International Workshop on Behavioral Modeling and Simulation (BMAS), pp. 97-102, Orlando/USA, Oct. 2000

[54] Modelica: `www.modelica.org`

[55] J. Mohring, J. Hoffmann, T. Halfmann, A. Zemitis, G. Basso, P. Lagoni, “Automated Model Reduction of Complex Gas Pipeline Networks”, Pipeline Simulation Interest Group, Palm Springs/USA, Oct. 2004

[56] Multifrontal Massively Parallel Sparse Direct Solver:
`http://graal.ens-lyon.fr/MUMPS/`

[57] L. Näthke, R. Popp, L. Hedrich, E. Barke, “Using Term Ordering to Improve Symbolic Behavioral Model Generation of Nonlinear Dynamic Analog Circuits“, Proc. of the European Conference on Circuit Theory and Design (ECCTD), 1999

[58] L. Näthke, “Ansätze zur automatischen Generierung hierarchischer Verhaltensmodelle von nichtlinearen integrierten Analogschaltungen“, Doctoral Thesis, Logos Verlag, 2004

[59] L. Näthke, L. Hedrich, E. Barke, “Betrachtungen zur Simulationsgeschwindigkeit von Verhaltensmodellen nichtlinearer integrierter Analogschaltungen“, ITG/GMM ANALOG 2002, Mai 2002

[60] L. Näthke, V. Burkhay, L. Hedrich, E. Barke, “Hierarchical Automatic Behavioral Model Generation of Nonlinear Analog Circuits based on Nonlinear Symbolic Techniques”, Proc. of the IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE), Vol. 1, No. 1, p. 10442, 2004

[61] M. Olbrich, R. Popp, L. Näthke, L. Hedrich, E. Barke, “A Combined Structural and Symbolic Method for Automatic Behavioral Modeling of Nonlinear Analog Circuits“, Proc. of the 15th European Conference on Circuit Theory and Design (ECCTD), Vol. 1, pp. 442-447, Espoo/Finland, Aug. 2001

[62] J. Phillips, “A Statistical Perspective on Nonlinear Model Reduction“, Proc. of the International Workshop on Behavioral Modeling and Simulation (BMAS), pp. 41-46, Oct. 2003

[63] A. Pothen, C.-J. Fan, “Computing the Block Triangular Form of a Sparse Matrix“, ACM Transactions on Mathematical Software, Volume 16, Issue 4, pp. 303-324, Dec. 1990

[64] R. Popp, W. Hartong, L. Hedrich, E. Barke, “Error Estimation on Symbolic Behavioral Models of Nonlinear Analog Circuits“, Proc. of the Intern. Conference on Symbolic Methods and Applications to Circuit Design (SMACD), 1998

[65] R. Popp, E. Barke, “Symbolic Analysis of Nonlinear Analog Circuits by Simplification of Nested Expressions“, Proc. of the Intern. Conference on Symbolic Methods and Applications to Circuit Design (SMACD), 2000

[66] R. Rosenberger, “Zur Generierung von Verhaltensmodellen für gemischt analog/digitale Schaltungen auf Basis der Theorie dynamischer Systeme“, Doctoral Thesis, Technical University of Darmstadt/Germany, 2001

[67] J. Roychowdhury, “Automated Macromodel Generation for Electronic Systems”, Proc. of the International Workshop on Behavioral Modeling and Simulation (BMAS), pp. 11-16, Oct. 2003

[68] Y. Saad, “Iterative Methods for Sparse Linear Systems“, PWS Pub. Co Boston, 1996

[69] T. Schneider, J. Mades, M. Glesner, A. Windisch, W. Ecker, “An Open VHDL-AMS Simulation Framework”, Proc. of the IEEE/ACM International Workshop on Behavioral Modeling and Simulation (BMAS), pp. 89-94, Oct. 2000

[70] M. Sofroniou, “A Package For Code Optimization Using Mathematica“, `http://library.wolfram.com/infocenter/MathSource/3947/`

[71] R. Sommer, E. Hennig, T. Halfmann, T. Wichmann, “Symbolic Modeling and Analysis of Analog Integrated Circuits“, Proc. of the European Conference on Circuit Theory and Design (ECCTD), Stresa/Italy, 1999

[72] C. Tischendorf, “Solution of Index-2-DAEs and its Application in Circuit Simulation”, Doctoral Thesis, Humboldt-Univ. zu Berlin, Logos Verlag, Berlin, 1996

[73] C. Tischendorf, D. Estévez Schwarz, “Mathematical Problems in Circuit Simulation”, Mathematical and Computer Modelling of Dynamical Systems, 2001, Vol. 7, No. 2

[74] J. Vlach, K. Singhal, “Computer Methods for Circuit Analysis and Design“, Van Nostrand Reinhold, 1994

[75] A. Vladimirescu, “The SPICE Book“, Wiley, 1994

[76] B. H. Wakayama, C. L. Zhou, C.-J.R. Shi, “Developing Device Models”, IEEE Circuits and Devices Magazine, Vol. 21, Issue 4, pp. 6-11, Aug. 2005

[77] O. Wallat, “Partitionierung und Simulation elektrischer Netzwerke mit einem Parallelen mehrstufigen Newton-Verfahren“, Doctoral Thesis, University of Hambug/Germany, 1998

[78] B. Wan and C. R. Shi, “Hierarchical Multi-Dimension Table Lookup for Model Compiler based Circuit Simulation”, Proc. of the IEEE Design, Automation and Test in Europe Conference and Exhibition (DATE), 2004

[79] B. Wan, E. Acar, S. Nassif, R. Shi, “Design-Adaptive Device Modeling in Model Compiler for Efficient and Accurate Circuit Simulation”, Proc. of the IEEE International Behavioral Modeling and Simulation Workshop (BMAS), pp. 113-118, San Jose/USA, Sep. 2004

[80] B. Wan, B. P. Hu, L. Zhou, C.–J. R. Shi, “MCAST: An Abstract-Syntax-Tree based Model Compiler for Circuit Simulation”, Proc. of the IEEE Custom Integrated Circuit Conference (CICC), pp. 249-252, Sept. 2003

[81] T. Wichmann, M. Thole, “Computer Aided Generation of Analytic Models for Nonlinear Function Blocks”, Proc. of the 10th International Workshop on Power and Timing Modeling, Optimization and Simulation, Vol. 1918/2000, p. 327, Springer, 2000

[82] T. Wichmann, “Symbolische Reduktionsverfahren für nichtlineare DAE-Systeme“, Doctoral Thesis, Shaker Verlag, 2004

[83] T. Wichmann, R. Popp, W. Hartong, L. Hedrich, “On the Simplification of Nonlinear DAE Systems in Analog Circuit Design“, Proc. of Computer Algebra in Scientific Computing, Jun. 1999

[84] S. Wolfram, “The Mathematica Book“, 5th edition, Wolfram Media, 2003

[85] P. Yang, B. D. Epler, P. K. Chatterjee, “An Investigation of the Charge Conservation Problem for MOSFET Circuit Simulation”, IEEE Journal of Solid-State Circuits, Vol. 18, No. 1, pp. 128-138, 1983

[86] G. Yu, P. Li, “Lookup table based simulation and statistical modeling of Sigma-Delta ADCs”, Proc. of the 43rd ACM IEEE Design Automation Conference (DAC), pp. 1035-1040, San Francisco/USA, 2006

[87] W. Zheng, Y. Feng, X. Huang, H. A. Mantooth, “Ascend: Automatic Bottom-up Behavioral Modeling Tool for Analog Circuits”, IEEE International Symposium on Circuits and Systems (ISCAS), Vol. 5, pp. 5186-5189, May 2005

[88] Accellera, “Verilog-AMS Language Reference Manual”, Version 2.2, Nov 2004

[89] BSIM3 Research Group of UC Berkeley:
`www-device.eecs.berkeley.edu/~bsim3/`

[90] IEEE Standard 1076.1, “IEEE Standard VHDL Analog and Mixed-Signal Extensions”, Mar. 1999

[91] Cadence Design Systems, “Compiled-Model Interface Reference”, Version 4.0, June 2004

[92] Cadence Design Systems, “Using Hierarchy and Isomorphism to Accelerate Circuit Simulation“, White paper, 2006

Publications and Supervised Theses

[93] C. Knoth, D. Platte, T. Halfmann, J. Broz, P. Rotter, “Generierung effizienter Verhaltensmodelle mittels Modellkompilierung und Modellreduktion”, ANALOG 2008: 10. GMM/ITG-Fachtagung Entwicklung von Analogschaltungen mit CAE-Methoden, Siegen/Germany, Apr. 2008

[94] D. Platte, S. Jing, R. Sommer, E. Barke, “Using Sequential Equations to Improve Efficiency and Robustness of Analog Behavioral Models”, Proc. of the ECSI Forum on Specification and Design Languages (FDL), Darmstadt/Germany, pp. 83-89, Sept. 2006

[95] D. Platte, R. Sommer, E. Barke, “An Approach to Analyze and Improve the Simulation Efficiency of Complex Behavioral Models”, Proc. of the IEEE International Behavioral Modeling and Simulation Conference (BMAS), pp. 79-84, San Jose/USA, Sept. 2006

[96] D. Platte, S. Jing, R. Sommer, E. Barke, “Ansätze zur Verbesserung der Simulationsperformance automatisch generierter analoger Verhaltensmodelle“, 9. ITG/GMM/GI Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), pp. 191-200, Feb. 2006

[97] D. Platte, R. Sommer, J. Broz, A. Dreyer, T. Halfmann, E. Barke, “Automatische nichtlineare Verhaltensmodellgenerierung mit sequentieller Gleichungsstruktur”, ANALOG 2006: 9. ITG/GMM-Diskussionssitzung Entwicklung von Analogschaltungen mit CAE-Methoden, pp. 149-154, Dresden/Germany, Sept. 2006

[98] D. Platte, S. Jing, R. Sommer, E. Barke, “Improving Efficiency and Robustness of Analog Behavioral Models” in Advances in Design and Specification Languages for Embedded Systems, pp. 53-68, ISBN 978-1-4020-6147-9, Springer Verlag Netherlands, 2007

[99] D. Platte, C. Knoth, R. Sommer, E. Barke, “High Performance Model Compilation for Complex Behavioral Models”, Proc. of the IEEE International Behavioral Modeling and Simulation Conference (BMAS), pp. 34-39, San Jose/USA, Sept. 2007

[100] R. Sommer, D. Platte, J. Broz, A. Dreyer, T. Halfmann, E. Barke, “Automatic Nonlinear Behavioral Model Generation using Sequential Equation Structure”, International Workshop on Symbolic Methods and Applications to Circuit and Design (SMACD), Florence/Italy, Oct. 2006

[101] S. Jing, “Optimizing Behavioral Simulation by Efficient Handling of Sparse Sequential Equations”, Master thesis, University of Hannover/Germany, July 2006

[102] C. Knoth, “Model Compilation for Analytic Behavioral Models of Analog Systems“, Master thesis, Technische Universität Ilmenau/Germany, Aug. 2007

Curriculum Vitae

Daniel Platte
born November 15th 1977 in Aachen/Germany

Education

May 28th 2004	Diplom (equivalent to Master of Science)
1998 – 2004	Studies in Electrical Engineering (Computer Engineering) University of Hannover/Germany
June 18th 1997	Abitur (equivalent to High School Diploma)
1990 – 1997	Kaiser-Wilhelm- und Ratsgymnasium, Hannover/Germany
1988 – 1990	Orientierungsstufe Birkenstraße, Hannover/Germany
1984 – 1988	Grundschule am Wäldchen, Arnum/Germany

Professional Experience

since June 2007	Verification Engineer, Qimonda AG, Munich/Germany
2004 – 2007	Ph.D. Student, Infineon Technologies AG, Munich/Germany