Formal Abstraction and Verification of Analog Circuits

Dissertation

**zur Erlangung des Doktorgrades
der Naturwissenschaften**

vorgelegt beim Fachbereich Informatik und Mathematik

der Johann Wolfgang Goethe-Universität

in Frankfurt am Main

von

Ahmad Tarraf

aus

Rinteln, Deutschland

Frankfurt 2020

(D 30)

vom Fachbereich Informatik und Mathematik der Johann Wolfgang Goethe-Universität als Dissertation angenommen.

Dekan

Prof. Dr. Lars Hedrich

Johann Wolfgang Goethe-Universität

Gutachter

Prof. Dr. Lars Hedrich

Johann Wolfgang Goethe-Universität

Prof. Dr. Christoph Grimm

Technische Universität Kaiserslautern

Datum der Disputation

19. Januar 2021

Foreword

This dissertation summarizes my research during the last 3 years at the institute of computer science in the design methodology group of the Goethe University in Frankfurt.

First, I would like to thank my supervisor prof. Lars Hedrich for his dedicated support and guidance. The meetings and conversations we had, were vital in inspiring me to think outside the box and from multiple perspectives. Without your guidance, this work would not be possible. Moreover, I am grateful to all of those with whom I have had the pleasure to work with at our institute and our project partners.

I want to thank all my teachers and professors that thought and encouraged me throughout my academic carrier. I am sitting on your shoulders. Hopefully someday I can help others to see further as well.

The direction our lives take is guided through a lot of consequent decisions. We choose a major part of these decisions to achieve the outcome we desire and think of as the best. During early age however, our parents take these decisions, enriching us with their hopes and dreams. For that, I want to thank my mother Nohad and my father Hassan for helping me become the person I am today. I am especially grateful to you mother for opening my eyes towards the future. Without your guidance, I would not be the person I am today. You have thought me more I can even express in words.

Special thanks to my brother Ali and my two sisters as well. I am grateful that you are a part of my life. Well brother, who thought that we both would follow academic careers and get so far. But I guess this is life, as a kid you dream to become a pilot, during your early twenties you dream of becoming an inventor and decide to study engineering, and today you write a dissertation about analog circuits. Funny how the sails we set for our course are changed by the winds of life.

Finally, I want to thank my friend Saskia Panthen for constantly supporting and encouraging me to follow my dreams. Thank you for being a part of my life.

Ahmad Tarraf

www.orcid.org/0000-0002-9174-5598
www.em.cs.uni-frankfurt.de
www.researchgate.net/profile/Ahmad_Tarraf3

Abstract

In this dissertation the formal abstraction and verification of analog circuits is examined. An approach is introduced that automatically abstracts a transistor level circuit with full Spice accuracy into a hybrid automaton (HA) in various output languages. The generated behavioral model exhibits a significant simulation speed-up compared to the original netlist, while maintaining an acceptable accuracy, and can be therefore used in various verification and validation routines. On top of that, the generated models can be formally verified against their Spice netlists, making the obtained models correct by construction.

The generated abstract models can be extended to enclose modeling as well as technology dependent parameter variations with little over approximations. As these models enclose the various behaviors of the sampled netlists, the obtained models are of significant importance as they can replace several simulations with just a single reachability analysis or symbolic simulation. Moreover, these models can be as well be used in different verification routines as demonstrated in this dissertation.

As the obtained models are described by HAs with linear behaviors in the locations, the abstract models can be as well compositionally linked, allowing thereby the abstraction of complex analog circuits.

Depending on the specified modeling settings, including for example the number of locations of the HA and the description of the system behavior, the accuracy, speedup, and various additional properties of the HA can be influenced. This is examined in detail in this dissertation. The underlying abstraction process is first covered in detail. Several extensions are then handled including the modeling of the HAs with parameter variations. The obtained models are then verified using various verification methodologies. The accuracy and speed-up of the abstraction methodology is finally evaluated on several transistor level circuits ranging from simple operational amplifiers up to a complex circuits.

Zusammenfassung

Die formale Verifikation von analogen Schaltungen und insbesondere die von analogen Mixed signal (AMS) Schaltungen, ist ein Forschungsgebiet, dass in unserer Zeit zunehmend an Wichtigkeit gewinnt. In erster Linie ist dies darauf zurückzuführen, dass die Zahl von Applikationen von sicherheitskritischen Systemen stetig zunimmt. Da bei diesen Applikationen das Eintreten von Fehlern verheerende Folgen hätten, müssen eben diese Fehler ausgeschlossen werden. Eine Möglichkeit hierfür bietet die formale Verifikation, welche eine mathematischen Beweis für die korrekt Funktion eines Systems liefert. Im Vergleich zu digitalen Systemen, bei denen die Verifikation ein standardisierter Bestandteil des Entwurfsprozesses ist, ist die Verifikation von analogen Systemen noch weit entfernt von diesem Zustand. Dies ist in erster Linie auf zwei Probleme zurückzuführen. Das wohl bekannteste Problem für die formale Verifikation im analogen Bereich ist die *Zustandsraumexplosion*. Durch die hohe Anzahl der kontinuierlichen Zustände, welche mit der Größe der Systemen beinahe exponentiell mit wächst, scheitern die meisten Ansätze daran, einen formalen Beweis für die Funktionalität eines Systems zu liefern.

Während im digitalen Bereich die Signale durch den diskreten Wertebereich beschränkt sind, können analoge Signale in der Regel beliebige Werte vom kontinuierliche Wertebereich annehmen. Dieses Problem wird in der vorliegenden Dissertationen als *kontinuierliches* Explosions-Problem bezeichnet. Beide genannten Probleme müssten gelöst werden, um die formale Verifikation im analogen Bereich zu ermöglichen.

Zurzeit dominieren in der Industrie die Simulationen in ihren verschiedenen Formen (Transienten, Monte Carlo-, Corner-, Sweep-Simulationen, etc.) den Verifikationsprozess, während die formale Verifikation im analogen Bereich nur beschränkt zum Einsatz kommt. Zwar liefert die Simulation eines Systems hinreichend gute Ergebnisse die es ermöglichen, dass System zu analysieren und zu prüfen, allerdings bietet diese Methode keinen formalen Beweis für die Funktionalität der entworfen Schaltung und kann dementsprechend nicht als formale Verifikation betrachtet werden. Ein weiterer Begriff, der oft in diese Zusammenhang fällt, ist die Emulation. Zwar kann die Emulation durch geeignete Hardware schneller als eine typische Simulation durchgeführt werden, allerdings ist dies durch die hohe Anzahl der Zustände und technische Limitierungen oft nicht realisierbar. Ähnlich wie die Simulation liefert die Emulation keinen formalen Beweise für die Funktionalität der Schaltung, sondern führt eine schnelle Validierung des Systems aus, welche auf die ausgeführten Fälle begrenzt ist. Außerdem ist für analoge Schaltungen keine Emulation i.a. gut verfügbar.

Ein Ansatz, der in dieser Dissertationen verfolgt wird, die Abstraktion des Systems, ermöglicht es, die Komplexität der Systeme zu reduzieren und damit einen Schritt Richtung durchführbarer formaler Verifikation zu gehen. Um ein genaues Modell zu erzeugen ist es notwendig, die analoge Schaltung auf Transistorebene zu abstrahieren. Die automatische Modellierung von analogen Schaltungen auf Transistorebene ist ein lange bekanntes Problem, welches bisher noch nicht zufrieden-

stellend gelöst werden konnte. Die meisten existierenden Abstraktionsverfahren sind in erster Linie manuell und beanspruchen dementsprechend relativ viel Zeit.

Außerdem kommt noch hinzu, dass die meisten Verfahren relativ ungenau Modelle liefern, selbst dann, wenn die genutzten Verfahren automatisiert sind. Diese ungenauen Modelle sind für die Verifikation ungeeignet, da sie gegenüber der originalen Netzliste große Abweichungen aufweisen. Das in dieser Dissertation vorgestellte Abstraktionsverfahren, welches automatisch Modelle mit einer hohen Genauigkeit liefert, soll diese Problem lösen. Dabei ist der gesamte Abstraktionsablauf automatisiert und resultiert durch die oben genannte signifikante Simulationsbeschleunigung gegenüber der Spice Netzliste in einer einstellbar genauen Beschreibung der Modelle. Das Verfahren zeichnet sich durch einen sehr hohen Abstraktionsgewinn und damit eine sehr hohe Beschleunigung der Simulation der Modelle aus. Die erzeugten Modelle können mit ihrer originalen Netzliste auf Transistorebene verglichen werden, um die Genauigkeit und den Speedup nachzuweisen. Dazu kommt noch, dass die Modelle ebenfalls in verschiedene Verifikationsroutinen verwendet werden können, dies wird in der vorliegenden Dissertation näher erläutert.

Das Verfahren, dass in dieser Dissertation vorgestellt wird, abstrahiert eine Spice Transistorlevel Netzliste mit voller BSIM4 Genauigkeit, von der Transistorebene auf die Systemebene, als einen hybriden Automaten. Dabei können verschiedene Einstellungen verwendet werden, welche die Genauigkeit der Modelle beeinflussen. Die vorliegen Dissertation versucht systematisch den Konstruktionsweg zu erklären, wobei im Vordergrund die Eigenschaften und Einflüsse der verwendeten Methoden analysiert und diskutiert wird.

Die vorgestellte Methode basiert auf der Transformation des nichtlinearen Systems stückweise in lineare Systeme in Kronecker-Form, verbunden mit einer Dominant-Pole-Ordnungsreduktion. Grundlegend stellt die Methode sicher, dass alle erreichbaren (nichtlinearen) Systemzustände berücksichtigt werden und bietet zudem einen formalen Beweis der Modellierungsgenauigkeit. Damit ist das Verfahren erstmals in der Lage, vollständig verifizierte, schnelle Verhaltensmodelle automatisch zu erzeugen.

Die Modellierung des abstrakten Modells kann in wesentlich in zwei Blöcke unterteilt werden. Im ersten Block wird der Zustandsraum einer Spice Netzliste mittels eines Programmes (*Vera*, dieses wurde am Institut für Entwurfsmethodik an der Goethe Universität in Frankfurt am Main entworfen) abgetastet. Grundsätzlich ist *Vera* ein Programm das entworfen wurde, um einen formalen Äquivalenzvergleich im analogen Bereich zwischen zwei Schaltungen durchzuführen. Dabei können die Schaltungen entweder in Spice oder in Verilog-A beschrieben sein. Das Programm wurde teilweise erweitert, um die oben genannten Abtastung nur einer Schaltung durchzuführen. Die Abtastung der Netzliste wird mit voller Spice Genauigkeit durchgeführt, wobei dies in einem automatisch reduzierten Zustandsraum erfolgt. Dabei wird das System um die abgetasteten Punkte linearisiert und die Ordnung des Systems auf ihr relevantes dynamisches Verhalten reduziert. Dies kann entweder durch die Vorgabe einer Ordnung geschehen, oder durch die Bestimmungen des relevanten Frequenzbereichs. Neben der Abtastung der Signale sowohl im Originalraum als auch im reduzierten Zustandsraum, werden wesentliche Werte ermittelt, die für die Modellierung essentiell sind. Diese Werte enthalten beispielsweise die Eigenwerte und Eigenvektoren des linearisierten Systems, die Verbindungen zwischen den Punkten gegeben durch einen gerichteten Verbindungsgraphen und die Transformationsmatrizen, welche die Verknüpfung zwischen den Zustandsräumen beschreiben. Nachdem das System abgetastet wurde, erzeugt der zweite Block der Modellierung den abstrakten hybriden Automaten. Dieser Block verwendet das Programm, dass das Ergebnis der vorgestellten Forschung darstellt: Eigenvalue Based Hybrid Linear System Abstraktion, oder kurz *Elsa*.

Die hinter *Elsa* liegende Methode soll aus den abgetasteten Werte ein Modell erzeugen, dass als hybrider Automat beschrieben wird. Dabei besitzt der erzeugte Automat eine Menge von diskreten Orten (Engl. locations). In jedem Ort $loc \in Loc = \{g1r1, \dots\}$ wird dabei das Systemverhalten durch eine lineare Zustandsraumdarstellung mit einer Menge von endlichen, in der Regel kontinuierlichen, Zustandsvariablen beschrieben. Der hybride Automat kann sich in einem Ort loc befinden, solange die Invariante des Ortes inv_{loc} gültig ist. Sobald die Invariante eines Ortes ungültig wird, muss der hybride Automat den Ort verlassen. Wie der Automat den Ort verlässt, beschreiben die Übergänge zwischen den Orten und die zugehörigen Sprungbedingungen. Der Automat kann einen Ort mittels eines Überganges (Engl. guard) verlassen, muss dies allerdings nicht sofort ausführen sobald der Übergang valide ist. Erst wenn die Invariante eines Ortes ungültig wird, muss der Automat den Übergang durchführen und in den dementsprechenden aktuellen Ort wechseln.

Die Methode zur Erzeugung des hybriden Automaten geht wie folgt vor: Basierend auf den abgetasteten Eigenwerte des linearisierten Systems werden zunächst Gruppen durch eine Clusteranalyse generiert. Im Anschluss werden Gruppen in Regionen aufgeteilt, die in dem selben Bereich im Zustandsraum sind. Gemeinsam bilden Gruppen und Regionen die Orte des Automaten. Bestimmte Einstellungen in *Elsa*, wie etwa die Gewichtungen der Eigenwerte während der Clusteranalyse, können die Anzahl der ermittelten Orte beeinflussen. Die Methodik, sowie die Einstellungen die diese Analyse beeinflussen werden in der vorliegenden Dissertation detaillierter analysiert.

Die abgetasteten Punkte werden den jeweiligen Orten zugewiesen aufgrund der Clusteranalyse. Die Punkte, die somit zu einem Ort gehören, werden mit Hilfe von konvexen Hüllen umschlossen. Diese Hüllen werden im Anschluss durch geeignete geometrische Objekte (Polytope, Zonotope oder Intervall Hüllen) repräsentiert und beschreiben somit die Invarianten der Orte. Zwischen den Invarianten werden im Nachhinein die Übergänge und die zugehörigen Sprungbedingungen ermittelt.

In jedem grundlegend Ort wird das Verhalten des Systems durch eine lineare Zustandsraumdarstellung beschrieben. Dabei bilden die Eigenwerte den wesentlichen Bestandteil der Systemmatrix. Die Eingangsmatrix wird durch geeignete Matrizen beschrieben, welche sich durch das Abstraktionsverfahren berechnen lassen. Dabei ist zu beachten, dass die Systembeschreibung in einem reduzierten Zustandsraum stattfindet ($\mathcal{S}_\lambda$). Um die Werte des Systems im Originalraum ($\mathcal{S}_o$) der Schaltung zu finden, ist ein Rücktransformation erforderlich, welche die Ergebnisse vom $\mathcal{S}_\lambda$ Raum zurück in den $\mathcal{S}_o$ Raum transformiert. Die erforderlichen Transformationsmatrizen werden dabei hauptsächlich durch die Eigenvektoren des linearisierten Systems beschrieben.

Die erzeugten abstrakten Modelle können in drei verschiedenen Ausgabesprachen generiert werden: Matlab (*Cora*), Verilog-A, und SystemC-AMS. Dabei unterscheiden sich sowohl die Beschreibungen, als auch die Methodik der erzeugten Modelle je nach gewählter Sprache. Während die Matlab (*Cora*) Modelle nichtdeterministisch sind, sind die Verilog-A als auch die SystemC-AMS Modelle deterministisch. Daher muss für die letzten beiden genannten Ausgabesprachen die Definition des hybriden Automaten angepasst werden. So wird, je nach gewählter Methodik, der aktuelle Ort des Systems entweder anhand der Invarianten unter Vernachlässigung der Übergänge, oder anhand der Übergänge unter Vernachlässigung der Invarianten bestimmt. Die erzeugten Matlab Modelle können anschließend in einer Erreichbarkeitsanalyse mittels *Cora* [Alt15] analysiert werden. Für die SystemC-AMS und Verilog-A Modelle können Simulationen mittels Standard-Simulatoren durchgeführt werden. Das vorgestellte Verfahren erzeugt somit aus einer Schaltung auf Transistorebene ein Verhaltensmodell, dass auf Systemebene durch einen hybriden Automaten beschrieben wird.

Das beschrieben Abstraktionsverfahren verwendet für die Bestimmung der Systemmatrizen, Eingangsmatrizen und Transformationsmatrizen verschieden Methoden, die allerdings alle das linear Modell durch Mittelwertbildung über viele linearisierte Abtastpunkte berechnen. Durch diese Mittelwertbildung können Fehler in den erzeugten Modellen entstehen, welche als Abstraktionsfehler bezeichnet werden. In der vorliegenden Dissertation wird das beschriebe Abstraktionsverfahren erweitert, um die Abstraktionsfehler zu umgehen, indem die Schwankungen der ermittelten Werte des linearen Modells durch eine geeignete Modellierung in das Modell mit aufgenommen werden. Dies kann sowohl für die Matlab (*Cora*) als auch für die SystemC-AMS Modelle verwendet werden. Für die *Cora* Modelle, können die Matrizen die zur Systembeschreibung dienen durch Matrizen-Zonotope (matZonotope [Alt15]) oder Intervall-Matrizen ersetzt werden. Für die SystemC-AMS Modelle werden die Elemente der Matrizen durch affine Formen aus der Affine Arithmetic Dicision Diagrams (AADD) Bibliothek [RGJR17] erweitert. Die erzeugten SystemC-AMS Modelle werden anschließend symbolisch simuliert, während die *Cora* Modelle weiterhin in einer Erreichbarkeitsanalyse mittels *Cora* verwendet werden können. Somit kann das Verfahren durch Umschließen der Schwankungen der Werte das Verhalten des abstrakten Modells erweitern, um den Abstraktionsfehler zu umgehen und das System mittels Bereichsarithmetik zu verbessern. Allerdings werden dadurch während der Simulationen zu jedem Zeitschritt Wertebereich bestimmt und nicht mehr einzelne Werte die in der Regel mit Überapproximationen verbunden sind.

Eine zusätzlich Erweiterung für das Abstraktionsverfahren ist die Modellierung der Parameterschwankungen, die wegen den Prozessparametern auftreten. Wie beschrieben, abstrahiert *Elsa* eine Transistorlevel Schaltung zu einem hybride Automaten. Dabei können die Prozessparameterschwankungen für die Elemente der Spice Netzliste angegeben werden. Dementsprechend werden mehrere Netzlisten von *Vera* erzeugt, die ähnlich wie bei einer Monte Carlo Simulation die Parameter der Netzlist variieren. Alle erzeugten Netzlisten werden parallel mit *Vera* abgetastet und ebenfalls parallel mittels *Elsa* zu hybriden Automaten abstrahiert. Die somit erzeugten Automaten werden anschließend zu einem Automaten zusammengeführt, welcher das Verhalten aller Automaten beinhaltet. Dementsprechend wird ein Automat erzeugt, der die Parameterabweichungen der Netzliste beinhaltet.

Ein wichtiger Aspekt der generierten Modelle ist deren Kompositionalität. Beispielsweise kann eine große Netzliste gegebenenfalls in mehrere kleine unterteilt werden, die unabhängig voneinander zu hybriden Automaten abstrahiert werden. Falls die erzeugten Modelle in Verilog-A oder SystemC-AMS sind, ist durch die Modularität der erzeugten hybride Automaten die Kompositionalität direkt gegeben. Durch Verbinden der erzeugten Module kann somit ein komplexes System durch einen kompositionalen hybride Automaten abstrahiert werden. Das entwickelte Programm *Elsa* ist ebenfalls in der Lage einen kompositionale Automaten in Matlab für *Cora* zu generieren. Dabei werden die Unterblöcke der Schaltung separat abstrahiert und anschließend als kompositionalen Automaten beschrieben, der einem Produktautomaten ähnelt. Anschließend kann eine

Erreichbarkeitsanalyse in *Cora* durchgeführt werden. Somit kann ein hybrider Automat nicht nur für schnelle Simulationen des analogen Blocks verwendet werden, sondern auch kompositional für größere Systeme eingesetzt werden.

Weitere Erweiterungen erlauben es, einen hybriden Automaten über die Optionen von *Elsa* zu optimieren und somit einen Automaten zu erzeugen, der die geringste Abweichung zur Spice Netzlist aufweist, oder einen hybriden Automaten so zu erweitern, dass das Verhalten der realisierten Schaltung beinhaltet wird [KTR+20].

Wie Anfangs erwähnt steht die formale Verifikation im Vordergrund in dieser Dissertation. Dabei können, je nach Ausgabesprache des erzeugten hybride Automaten, unterschiedliche Verifikationsverfahren verwendet werden. Wird beispielsweise die Netzliste zu einem hybriden Automaten in Verilog-A abstrahiert, so kann ein formaler Äquivalenzvergleich (Equivalence-Checking) zwischen dem Verilog-A Modell und der Spice Netzlist durchgeführt werden. Dies kann mittels *Vera* realisiert werden. Dabei vergleicht *Vera* den reduzierten Zustandsraum beider Systeme. Dadurch kann der Fehler zur Netzliste ermittelt werden und dementsprechend ein Fehlermaß angegeben werden, welcher den Unterschied beider Systeme beschreibt. Somit ist das Abstraktionsverfahren durch die Verifikation der erstellen Modelle abgesichert.

Für die Matlab (*Cora*) Modelle wird ein andere Ansatz verfolgt. Diese Modelle können in einer Erreichbarkeitsanalyse verwendet werden, wodurch Regionen im Zustandsraum analysiert werden können (Engl. state space exploration). Zusätzlich wurde ein Model-Checking-Verfahren entwickelt, dass im Anschluss solch einer Analyse spezifische Eigenschaft auf deren Erfüllbarkeit überprüft. Somit kann festgestellt werden, ob das System die Spezifikationen erfüllt. Durch die oben genannten Erweiterungen kann das System durch einen hybriden Automaten mit Parametervariation erzeugt werden, der sowohl die Abstraktionsfehler als auch die Parameterschwankungen durch den verwendeten Prozess beinhaltet und somit für die Verifikation des Systems geeignet ist.

Wurde der hybride Automat in SystemC-AMS beschrieben können Online-Monitore verwendet werden, die es erlauben während der Simulation das System auf spezielle Eigenschaften zu untersuchen, und im Falle einer Verletzung der Bedingung einen Fehler liefern. Ähnlich zu den Matlab Modellen können die SystemC-AMS Modelle ebenfalls mit Parametervariation mittels der AADD modelliert und anschließend symbolisch simuliert werden. Die Online-Monitore werden dabei ebenfalls mit der genannten Bibliothek erweitert und können somit das Gesamtverhalten des erzeugten Modells beobachten. Je nach Art des Modells können dementsprechend verschiedene Verifikationsverfahren verwendet werden, um die Funktionalität der generierten Modelle zu gewährleisten.

Das vorgestellte Verfahren wird anhand verschiedener Beispiele demonstriert, die von simplen Filtern zweiter Ordnung, bis hin zu komplexen Schaltungen aus der Industrie reichen. Zusätzlich wird der kompositionale Ansatz anhand eines Beispieles aus der Automobileindustrie verdeutlicht, um die Skalierbarkeit des Ansatzes zu zeigen.

Zusammengefasst wird in der vorliegenden Dissertation ein Ansatz um die formale Verifikation im analogen Bereich zu ermöglichen beschrieben. Dieser basiert auf der Abstraktion des Systems. Durch Modellierung mittels Parametervariationen löst der Ansatz das Problem der Ungenauigkeiten, die meistens mit Abstraktionsverfahren verbunden sind. Verschiedene Verifikationsverfahren die, basierend auf der Ausgabesprache der Modelle, verwendet werden können, wurden

vorgestellt. Durch die Kompositionalität des Verfahrens kann die Skalierbarkeit des Ansatzes realisiert werden. Wodurch komplexe Schaltungen im analogen Bereich verifiziert werden können.

Contents

Notations

General Notation

A/a	a non-bold letter in math font represents a scalar
$\boldsymbol{A}$	an uppercase bold letter represents a matrix
$\boldsymbol{a}$	a lowercase bold letter represents a vector. If two lowercase letters are in bold, the first letter is the starting point of the vector, while the second one the ending point

Abbreviations

AADD	affinearithmetic decision diagrams
AMS	analog mixed signal
BIBO	bounded input bounded output
BSIM	Berkeley Short-channel IGFET Model
CH	convex hull
CHA	compositional hybrid automaton
CTL	computational tree logic
DAE	differential algebraic equation
EC	equivalence checking
freq	frequency
HA	hybrid automaton
HAs	hybrid automatons/automata
JNF	Jordan normal form
KCF	Weierstrass-Kronecker canonical form
LTI	linear time-invariant
MC	Monte Carlo
MIMO	multiple input multiple output
MNA	modified nodal approach

ODE	ordinary differential equation
s	seconds
SISO	single input single output
TDF	timed data flow
V	volt

Variables

δ	error
ϵ	threshold for a neighborhood search radius
η	index of nilpotency
γ_i	geometric multiplicity of an eigenvalue λ_i
λ	eigenvalue
$\mathcal{I}$	interval hull
$\mathcal{P}$	polytope
$\mathcal{S}$	state space spanned by the components of the state vector x
$\mathcal{S}_\infty$	state space spanned by the components of the state vector x_∞
$\mathcal{S}_\lambda$	state space spanned by the components of the state vector x_λ
$\mathcal{S}_o$	state space spanned by the components of the state vector x_o
$\mathcal{Z}$	zonotope
$\boldsymbol{A}$	$n \times n$ system matrix
$\boldsymbol{B}$	$n \times k$ input matrix
$\boldsymbol{C}$	$p \times n$ output matrix
$\boldsymbol{D}$	$p \times k$ feedthrough matrix
$\boldsymbol{E}$	$n \times n$ mass matrix
$\boldsymbol{I}$	identity matrix
μ_i	algebraic multiplicity of an eigenvalue λ_i
ν	maximum number of locations
$\boldsymbol{q}$	vector of charge variables
$\boldsymbol{u}$	input vector
$\boldsymbol{x}$	state space vector
$\boldsymbol{y}$	output vector
k	number of inputs

l	number of sampled points
m	number of state space variables in the reduced space $\mathcal{S}_\lambda$
n	number of state space variables in the $\mathcal{S}_o$ space
p	number of outputs
r	number of dynamic state space variables in the state space $\mathcal{S}_s$ before order reduction

1 Introduction

1.1 Motivation

In our modern world, recent technological trends have witnessed significant growth, enhancing their influencing on our lives. These trends include the deployment of artificial intelligence in various environments like autonomous driving, industrial automation, and human-robot collaboration. With the rise of the internet of things (IoT) and the obstacles accompanied by it, like big data and integration problems, artificial intelligence proves to be a game changer. Even in application like energy management and security systems, artificial intelligence can be presented. Aside from artificial intelligence, in general electronics devices and controllers are becoming more and more parts of our daily lives. From circuits deployed in constantly evolving manufacturing processes to circuits embedded in humans for the control of robots [AIA14], there seems to be no limit to the application environment. Recent circuit are generated as self-healable electronic tattoos [WLL+19] and are used for noninvasive and high-fidelity sensing. Thus, through various applications and emerging technologies, electronic circuits prove themselves to be the key to an era.

These innovative technologies, regardless of their applications, are usually accompanied by complex circuits or system on chip (SoC) designs. Considering the safety critical scenarios these technologies often underlie, there is a demand for the absolute reliability of not only the functionality of the designs, but also of the fabricated circuits. New standards, such as ISO 262626 force integrated circuit designers to invest much more effort into the verification and validation process. In some industries, the verification task increasingly dominates the design flow with up to 70 % [BGG+09; LAH+15], which is a significant amount of time. Thus it's no surprise that verification has become one of the most important topics in the circuit design.

Formal verification, on the other hand, is only partially used in such a design flow. In contrast to the digital design, where system verification has become part of the basic design routines, the analog and especially the analog mixed signal (AMS) verification clearly lags behind. Moreover, formal verification approaches lack advanced methodologies to handle formal verification on a large scale. Thus, formal techniques do not scale well with large complex circuits. This can be traced back to several factors, of which one of the most dominant factor is the continuous nature of the analog domain. Compared to a digital system, which is usually modeled as a discrete event dynamic system, an analog system has a continuous dynamic behavior, in generally specified by differential equations. In contrast to the finite set of possible input stimuli and the various binary sequences they induce in the digital domain, the real valued input signals of analog systems could attain theoretically infinitely many values generating thereby infinite trajectories in the continuous state space of the system, which challenges the verification task. Compared to the digital formal verification methodologies, much more effort must be deployed to achieve similar advances in the analog/mixed-signal domain.

1.2 State of the Art

In general, mixed-signal verification can be categorized as shown in Fig. 1.1. AMS verification can be classified according to [GXGM19; Rad16] into two approaches: formal methods and simulation-based methods. Often, emulation is not considered, even though it can yield faster results than traditional simulations. However, it is usually accompanied by high costs, a lot of modeling effort, and several technological limitations especially in the analog domain.

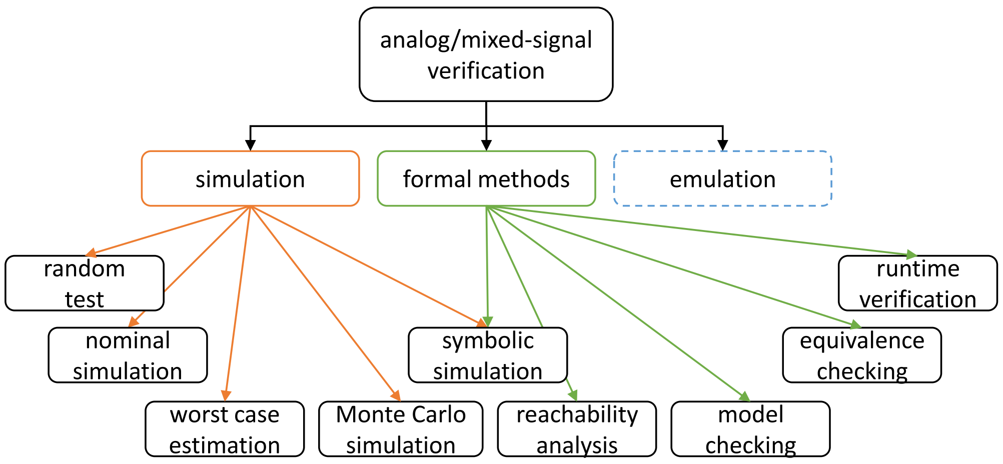


Fig. 1.1. Overview of the verification methods for analog/mixed-signal circuits.

Traditional simulation-based methods cover some test scenarios for a specific set of input stimuli and initial conditions. These computational expensive simulations including simple simulations enhanced by corners, sweeps, and Monte Carlo simulations, deliver a clear understanding in the functionality of the developed circuit. However, they do not fully verify the system behavior, and therefore cannot be considered formal methods [Gie05; RG16]. More precisely, we cannot eliminate the possibility by performing simple simulations that the circuit might reach bad states that will cause system failure. From a different perspective, this lies in the fact that simulation-based methods to not completely search the state space, and that the evaluation of the results is usually manually performed in an informal fashion. Therefore, these methods only validates a design, but do not verify it.

Formal methods attempt to prove in a formal manner that a circuit satisfies the specified specification and performs correct under all circumstances as desired by the circuit designer. Specifically, the objective of formal verification is to mathematically prove the properties of a system, usually during the design phase. This is obviously accompanied by a lot more computational effort than a single simulation. The results obtained, are often valid over a specified range of the inputs stimuli and cover a portion of the state space of the circuit. Due to the continuous nature of analog circuits, there are actually infinitely many possible input stimuli. However, due to the technological as well as the environmental constraints, this aspect is usually bounded, favoring the verification over bounded regions. Another aspect that challenges formal verification are process parameter deviations. Often, nominal models are considered in the verification tasks. However, due to the process parameter, the system can still fail as deviations emerge between the verified system behavior and the behavior of the real circuit. Hence, the process parameters must be considered as well in the verification task, making it an ever harder challenge. On top on that, even more challenges arise from the sensitivity of the designed circuit to environmental factors like signal noise, temperature, and higher order physical effects like different parasitics and current leakage [ZTB08]. Thus, there are a lot of factors that need to be considered in the verification process.

Several publications exist that differently categorize the formal techniques. In [GXGM19] the terms pre- and post-silicon analog verification are used. While pre-silicon verification focuses on the correctness of a design, post-silicon verification focuses on the verification of the fabricated circuit. Pre-silicon verification can be classified into two approaches: formal methods and simulation-based techniques. While [GXGM19] divides the formal methods into three categories: property checking and monitoring, affine arithmetic, and model and equivalence checkers, [ZTB08] divides the formal methods into four categories: equivalence checking, model checking and reachability analysis, runtime verification, and proof-based and symbolic methods. Here, a classification similar to [Rad16] is adapted, which classifies the formal methods into five categories: symbolic simulations, reachability analysis, model checking, runtime verification, and equivalence checking.

Depending on the nature of the specifications, there are several methods that can be used for the formal verification of AMS circuits. For example, in the presence of a golden model, equivalence checking can be used to examine whether these systems are equivalent with respect to their functionality. At its simplest, equivalence checking compares the input-output behavior of both systems. This usually requires the specification of tolerances or bounds on the parameters and signals [ZTB08]. Hence, a failure occurs once the specified tolerance values are violated. Several approaches exist that perform equivalence checking in the AMS domain [BHA95; SA01; HB5; HB23; Sal02; SL10; SH10a; SH12a]. While in [BHA95] equivalence checking is performed on two analog circuits using their transfer functions, [SA01] models the verification problem as a non-linear optimization problem by ensuring that the implementation response is bounded within an envelope around the specification under the influence of parameter variation. [Sal02] purposes and equivalence checking at system-level, rather than at circuit-level, by partitioning the specification and implementation codes into digital, analog, and data converter components, followed by verifying the digital part with a SAT/BDD algorithm, the analog part by a set of rewriting rules, and the converters between these parts by a matching procedure. However, this approach is only applicable on simple designs as it is difficult to find appropriate rewriting rules to arbitrary classes of analog circuits. A practical hierarchical, however, semi-formal equivalence checking methodology is provided in [SL10], that formulates equivalence checking as a constrained optimization problem. In [HB23], the first paper of a series is introduced, that delivers an approach for the formal verification of linear analog circuits with parameter tolerances, proving that a circuit fulfills a specification in a given frequency interval for all parameter variations. [HB5], the series is continued, proposing an equivalence checking approach for nonlinear circuits by comparing the implicit nonlinear state space descriptions of the two systems. The approach is extended in [SH10a] by a structural recognition and mapping of eigenvalues to circuit elements via circuit variables, and by a reachability analysis that restricts the investigated state space to the relevant parts. In [SH12a], the approach is even further enhanced by an efficient input stimuli generation algorithm that guarantees coverage of the entire reachable state space. The main drawback of this algorithm, similarly to main other formal verification methodologies, is the state space explosion problem which occurs when to many states are considered in the examination of the circuits. However, as this approach operates on a reduced state space, there is room for a trade off accuracy for states to consider.

If the specifications are at hand, offline or online monitors can be used that fall into the category of runtime verification [MN04; GT07; WAN+09a; JKN10; MN13]. Several specification languages exist for runtime verification, which can be categorized into offline methods like Ana CTL (computation tree logic for analog circuit verification) [DC05] and approaches based on PSL (Property Specification Language) [GT07], and online methods like CT-CTL (continuous time CTL) [ZTB06] that extends TCTL, and many others [NM07; MPDG09; WAN+09b]. Other technique such as STL (signal temporal logic) [MN04; MN13], which is an extension to the MTL (metric timed linear temporal logic) [TR05], support both online and offline monitors.

If these specifications are expressed in logical statements, and the whole state space is explored to check whether the system satisfies the desired specification, the verification methods fall into the domain of model checking [HHB02; HKH04; GBR04; DDM04; GPHB05; GPHB05]. [HKH04] proposes the discretization of the infinite continuous state space of nonlinear analog systems. The system properties are described using computation tree logic (CTL). [DC05; GPHB05] extent CTL to cover analog behavior. For example BLTL (bounded linear temporal logic) [WKZC11] uses model checking and sequential statistical techniques to verify properties of analog circuits in both the temporal and the frequency domain. In this process, randomly sampled system traces are sequentially generated using Spice and passed to a trace checker to examine the validity of a specification until the desired statistical strength is achieved. On the other hand, ASL (Analog Specification Language) [SH08] allows the definition of circuit properties such as gain, rise time, and slew rate, and examines these properties on the circuits which are modeled as discrete graph structures. Compared to runtime verification, model checking is computational much more expensive. Moreover, in order to examine a specification, model checking approaches completely explore the state space of the circuit. In contrast, runtime verification cannot guarantee conformance to specification due to the finite number of tested signal traces [LAH+15].

Another approach that examines a set of given specifications can be achieved using symbolic simulations. These simulations use symbols instead of numeric values [Hen00; AZT07; WLM+08]. Compared to standard simulation, symbolic simulation achieves a higher comprehensive coverage in fewer simulations runs, as for example through substitution in the results, a single symbolic simulation can replace several numerical ones. In [AHP96] for example, an automated symbolic model checking procedure is introduced for embedded systems. The specifications are described in a temporal logic and verified by a symbolic fixpoint computation. Symbolic simulations can prove that the circuit behavior is contained in the set of given specifications. However, as the number of symbolic variables increases and more complex equations are used, these simulations become no longer feasible. Affine arithmetic [RSRG12] or interval arithmetic [ZATB07; YDL12] can be used in these calculations, thereby reducing in general the number of symbolic variables in a trade for accuracy (over approximations). In [RG16] an extension to affine arithmetic is proposed called XAAF (extended affine arithmetic form) that additionally considers the relational operators in the control flow.

The last branch of formal verification is the reachability analysis. This analysis works with reachable sets using different geometrical representations to perform state exploration directly on system dynamics. Since it is often not possible to find the exact representations of all reachable sets, the reachable sets are often over approximated. Reachable sets are usually expressed by polyhedra

[ABDM00; GBR04; CK03; DDM04; FKR06] or by zonotopes [ARK+13].

Recently in [TKR+20], a reachability analysis was used to perform an equivalence checking in the analog domain of an abstracted model with a conformant model that was generated trained with the measurements from the real circuit, thus, performing a post fabrication verification of the model. So, connecting the verification methods can sometimes yield even better and faster approaches than trying to apply them separately. Another innovative method, where numeric and symbolic simulations were combined to yield the term *nubolic* is presented in [ZG19].

Even though several methods have been developed for the analog verification, simulation-based methods still dominate the industrial design flow. Certainly, verifying such complex analog circuits, and especially AMS circuits, is not an easy task. Due to the modern increasing complexity of circuits arising partially from the demand of accurate models and partially from the size of these circuits, modern verification methodologies are often not able to keep up with this rapid increase. This can be often traced back to the state space explosion problem. In fact, a recent survey [ZTB08] conducted on formal verification methods, raveled that the methods still suffer from the state space explosion problem. Furthermore, the equivalence checking task in the AMS domain becomes a difficult problem in the presence of tolerance margins, while model checking approaches that utilize abstracted models suffer from the over approximated behavior of these models. On top on the state explosion, which is also well known in the digital domain, it is possible to define a continuous explosion [ZTB08; FH18] that must be handled in the analog domain due to the infinite many possible continuous values a signal may attain. Thus, as the formal verification of analog/-mixed signal circuits is a relatively young research field, there is still a room for improvements.

Several approaches have been deployed that try to solve continuous and state explosion problems differently. Some approaches tackle these problems directly by a model order reduction and the use of a discretized state space [HKH04]. Other approaches use range arithmetic to tackle the state and continuous explosion [RG16; ASB07]. Recent methods try to use indirect measures like coverage on top of standard simulations to close the confidence gap, and thereby be more formal [FGG+17; BFG+16]. However, these approaches often still consume a lot of time. Moreover, even though modern approaches yield solid and good results, they are still challenged by strong nonlinear behaviors as well as the size of the circuits, i.e. most approaches are not scalable.

According to [GXGM19], formal verification of AMS circuits typically involves working on a higher level of abstraction, as this results in significant speed-up of validation routines. However, this has the drawback that inaccuracies might emerge due to this abstraction. More precisely, a large speed-up factor of abstracted behavioral models is desired to support complex simulation of a circuit at system or at least at module level while maintaining accurate results. As the systems integrable on a chip become more complex and heterogeneous, the use of accurate behavioral models for analog signal processing and interfacing would enhance design and simulation routines, on top of offering new possibilities and improving the current verification routines. Thus, one way to make formal verification applicable on complex analog circuits can be achieved by behavioral abstraction, which permits faster verification routines with fewer state variables. Nonetheless, a behavior model is often abstracted to a degree that it does not accommodate the full system behavior. This challenges the verification task as accurate abstracted models are mandatory.

The problem of generating an accurate abstract model from a transistor level circuit has been around for some time. Different approaches exist that have tried to solve this problem, e.g. automatic behavioral modeling [ZFHM05; Bor98; CWL+15; SWL+17]. These techniques are not targeting hybrid automata and are mainly improving the simulation speed. The method in [LAH+15] models the underlying DAE-system of electrical networks using piecewise linear regions, for each nonlinear element, on-the-fly. It suffers from using an abstract transistor-model and is limited to a specific number of transistors to be verified. However, it generates a complex hybrid automaton on-the-fly preventing the state explosion problem at initialization and during evaluation of a given input stimulus. Unfortunately, the HA is very complex as it is a cross product of all linearized regions of all nonlinearities

For high level continuous systems, methods modeling the analog circuit as a hybrid system are widely used [DDM04; FLD+11; FHS+07; ARK+13]. These methods are able to handle up to 20 state variables, if the underlying locations use linear ordinary differential equations (ODEs) to describe the system behavior. Mostly, they use reachability analysis to prove safety properties. To close the chain of proof at transistor level, hybrid automatons (HAs) are usually not suitable as the ODEs become nonlinear differential algebraic equations (DAEs). Another aspect that must be considered in the generation of an abstract model are the technology dependent parameter variations. Most abstraction approach do not incorporate these variations into the generated model, and if they do, they result in large over approximations and modeling times. In order to have results with technology accuracy e.g. a BSIM3, BSIM4, or Hicum accurate verification, sample-based formal verification methods [SH12b] could be used, however, the range-based proof is lost. These methods, in contrast, can handle much larger circuits; up to 80 transistors. Hence, there is a need for an abstraction approach that incorporates technology dependent parameter variations into the abstract models. Moreover, there is as well a need for an accurate abstraction methodology that delivers accurate abstract models with a reduced order suitable for verification routines.

1.3 Contribution

This dissertation aims to contribute to the formal verification of AMS circuits by generating accurate behavioral models that can be used for verification. As accurate behavioral models are often handwritten, this dissertation proposes an automatic abstraction method based on sampling a Spice netlist at transistor level with full Spice BSIM accuracy. The approach generates a HA that exhibits a linear behavior described by a state space representation in each of its locations, thereby modeling the nonlinear behavior of the netlist via multiple locations. Hence, due to the linearity of the obtained model, the approach is easily scalable.

Large speed-up factors can be achieved by the generated models while maintaining a high accuracy, making them suitable for verification purposes. On the other hand, the modeling process proposed in this dissertation is able to enclose technology dependent parameter variations of the circuit elements as well as the deviations that result from the abstraction process.

Moreover, the generated models can be formally verified against their original Spice netlist, yielding the deviations between the original Spice circuit and the abstracted model. Various verification routines can be executed with the obtained models. For example, a reachability analysis or a symbolic simulation can be performed on the models that enclose the parameter variations, thereby capturing all possible behaviors the circuit exhibits.

To sum up, the contributions presented in this dissertation are:

1. an automated abstraction method is presented that starts with a netlist in Spice or Verilog-A syntax and yields accurate behavioral models in Matlab (*Cora* [Alt15]), Verilog-A, or SystemC-AMS that can be used for fast and accurate simulations
2. generated by the pointwise analysis of the linear properties of the circuit, the HA created has a linear behavior in its locations. Compared to the original system, the generated HA has a significant lower dynamic order due to the order reduction performed during the abstraction process. Moreover, the technology dependent behavior of the original netlist is embedded in the obtained models, and the accuracy of the models can be controlled by the user
3. various extensions of the models exist:
   a) the creation of abstract models with little over approximations that model the parameter variations that result from the variation of the process parameters of the netlist as well as from the abstraction procedure
   b) the creation of compositional models that tackles the state space explosion problem. As the Verilog-A and SystemC-AMS models are pin-wise compatible and compositionality is directly given for these models, the compositional approach targets *Cora* models that are used for reachability analysis.
4. created models (Verilog-A) can be formally verified using equivalence checking against the original netlist, enabling correctness proof
5. use these models in various formal verification routines including reachability analysis coupled with model checking, symbolic simulations and runtime verifications.

1.4 General Concept and Outline

In the following, the underlying general concept for a fully automated abstraction methodology will be presented. The approach starts with a netlist described in Verilog-A or Spice with BSIM accuracy and results in a model described at system level as a HA (Section 2.4). An overview of the approach is illustrated in Fig. 1.2.

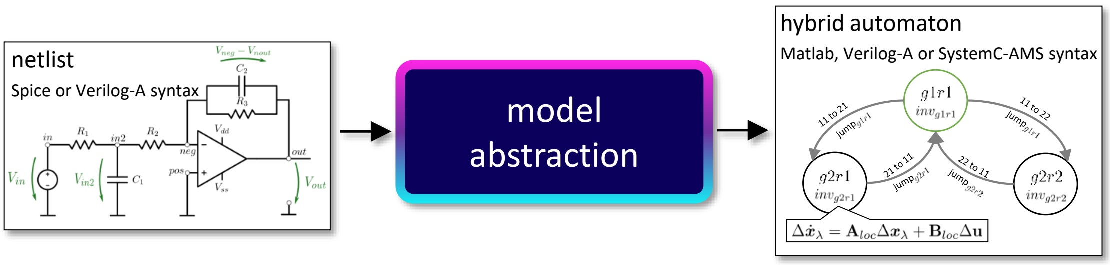


Fig. 1.2. Overview of the model abstraction approach.

The introduced approach uses two tools to realize the model abstraction: *Vera* and *Elsa*. Fig. 1.3 shows a closer look at the model abstraction approach. The abstraction methodology starts with sampling the netlist via in-house tool called *Vera*. Using the sampled data, *Elsa*, which stands for **e**igenvalue-based hybrid **l**inear **s**ystem **a**bstraction, abstracts the netlist into a HA.

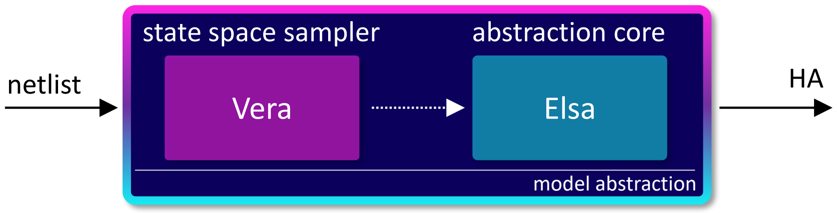


Fig. 1.3. Overview of the different tools used for the model abstraction.

Significant data and properties of the sampled systems, such as the eigenvalues, are used in order to construct the HA with a linear behavioral description in each of its locations. Fig. 1.4 provides a detailed illustration of the model abstraction process. As observed in Fig. 1.4, *Vera* samples the netlist and stores the result in an *acv* file. This file is then processed by *spaceM* to transfer the data into the memory of Matlab as a structure referred to as *space*. The Matlab structure *space*, on the other hand, is processed by *Elsa* to generate a HA.

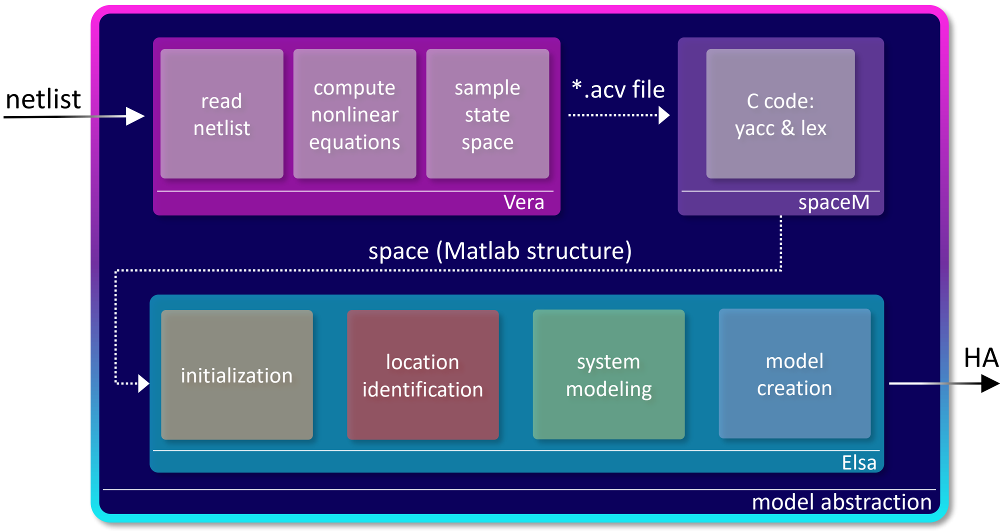


Fig. 1.4. Detailed overview of the model abstraction approach showing the components as well as the interconnections between them.

In the following, the abstraction process will be examined in detail. Starting with Chapter 2, the fundamental basics and principles used in this dissertation are briefly reviewed. As this dissertation is an interdisciplinary work, combining control theory with different branches from electrical engineering, such as computer science and artificial intelligence, only a part of the fundamentals is reviewed. After this revision, the sampling of the netlist performed by *Vera* will be examined in Chapter 3. According to Fig. 1.4, the abstraction process performed by *Elsa* can be divided into 4 main blocks: initialization, location identification, system modeling, and model creation. Each of these blocks consists of different layers. In Chapter 4, a closer look is taken at the abstraction core with all its underlying blocks and layers, followed by some powerful extensions for the generated abstract models in Chapter 5. The possible formal verification processes on the generated models and their reference netlists or specifications are demonstrated in Chapter 6. In Chapter 7, several examples are handled to illustrate and examine the model abstraction approach. Finally, a conclusion is stated in Chapter 8 along with some future directions.

2 General Basics and Principles

"We are like dwarfs sitting on the shoulders of giants. We see more, and things that are more distant, than they did, not because our sight is superior or because we are taller than they, but because they raise us up, and by their great stature add to ours."

– John of Salisbury, *Metalogicon Of John Salisbury*

A solid background in different principles is necessary for the proceeding topics. In this chapter, fundamental basics and principles are recapped.

2.1 Geometry

For what follows in the later chapters, geometry plays a key role in establishing the methodologies and concepts. In this section, few geometric objects are reviewed.

2.1.1 Polytopes

Polytopes are geometric objects in $\mathbb{R}^n$. Throughout this dissertation only convex polytopes will be considered. A (convex) polytope can have two types of representation:

1. halfspace representation
2. vertex representation

A halfspace:

$$\mathcal{H} = \{\boldsymbol{x} \mid \boldsymbol{c}^T\boldsymbol{x} \leq d\}, \tag{2.1}$$

is one of two parts obtained by bisecting the n-dimensional Euclidean space with a hyperplane $\mathcal{P}_h = \{\boldsymbol{x} \mid \boldsymbol{c}^T\boldsymbol{x} = d\}$, such that $\boldsymbol{c} \in \mathbb{R}^n$ represents the normal vector to the hyperplane and $d \in \mathbb{R}$ is the scalar product of any point on the hyperplane with the vector $\mathbf{c}$. Thus, a polytope $\mathcal{P}$ is the nonempty intersection of m halfspaces. A formal definition is stated in Theorem 2.1.1.

Theorem 2.1.1. *A convex polytope $\mathcal{P}$ is bounded by m intersections of halfspaces:*

$$\mathcal{P} = \left\{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{Cx} \leq \boldsymbol{d},\, \boldsymbol{C} \in \mathbb{R}^{m\times n},\, \boldsymbol{d} \in \mathbb{R}^m\right\} \tag{2.2}$$

Representing a polytope with the vertex representation can be performed by generating a convex hull (CH) over the finite set of points as stated in Theorem 2.1.2.

Theorem 2.1.2. *For k vertices $\boldsymbol{pt}_i \in \mathbb{R}^n$ a convex polytope $\mathcal{P}$ is the set:*

$$\mathcal{P} = CH(\boldsymbol{pt}_1, \ldots, \boldsymbol{pt}_k) \tag{2.3}$$

Polytopes used in the model abstraction in this dissertation are generated using the mpt toolbox [HKJM13]. The algorithm described in [BDH96] is used to generate convex hulls.

2.1.2 Zonotopes

Similarly to polytopes, zonotopes are geometric objects in $\mathbb{R}^n$ as well. Zonotopes have special symmetric properties which allow a compact representation. A formal definition of a zonotope is stated in Theorem 2.1.3.

Theorem 2.1.3. *Consider a zonotope $\mathcal{Z}$ with a center $\boldsymbol{c} \in \mathbb{R}^n$ and k generators $\{\boldsymbol{g}_i \in \mathbb{R}^n \mid i = 1, \ldots, k\}$, $\mathcal{Z}$ is defined as:*

$$\mathcal{Z} = \left\{ \boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{x} = \boldsymbol{c} + \sum_{i=1}^{k} \beta_i \boldsymbol{g}_i,\ \beta_i \in [-1, 1] \right\} \tag{2.4}$$

Thus, a zonotope can be interpreted as a Minkowski sum of line segments [Alt15].

A zonotope can be associated with affine arithmetic, as the operations performed on this geometric shapes are like to those performed with this computational type. As in affine arithmetic, a zonotope keeps track of the correlations between the spanning variables using generators, thereby solving as well the dependency problem associated with interval arithmetic.

If a given set of points is hulled by a zonotope, the zonotope has in general significant over approximation compared to hulling the same points with a polytope as shown in Fig. 2.1. This can be traced back to the symmetry of a zonotope.

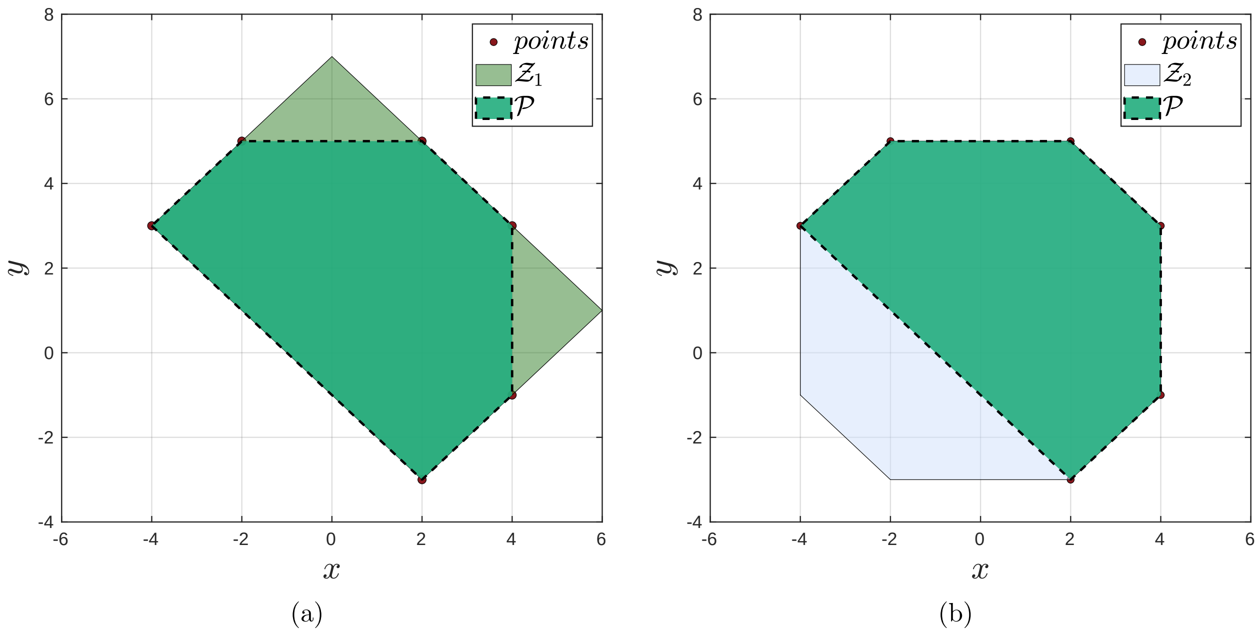


Fig. 2.1. Example illustrating the over approximated hulling performed by zonotopes with (a) two and (b) three generators compared to polytopes.

In Fig. 2.1, some random points are hulled by a zonotope $\mathcal{Z}_i$ as well as by a polytope $\mathcal{P}$. As observed, the zonotope $\mathcal{Z}_1$ which has only two generators over approximates the convex hull containing the points. If the number of generators used is increased to three, the result is still over approximated ($\mathcal{Z}_2$) compared to the hulling performed by the polytope as shown in Fig. 2.1a. For this example, $\mathcal{Z}_2$ is even larger than $\mathcal{Z}_1$.

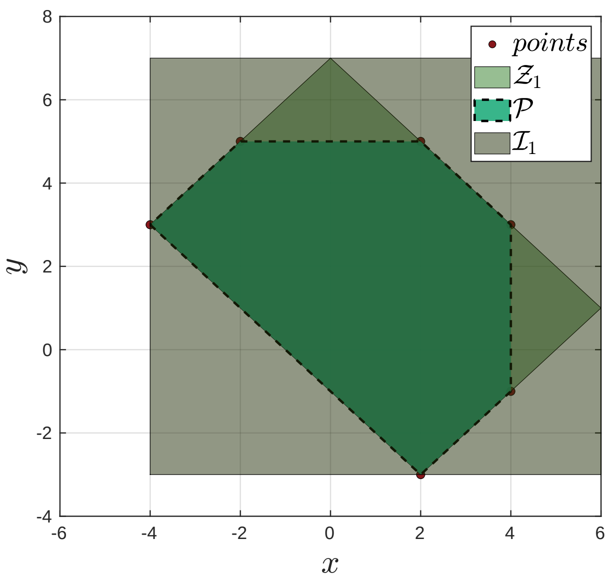
2.1.3 Interval Hulls

An interval hull $\mathcal{I} \in \mathbb{R}^n$, which can be represented as a multidimensional interval, can be interpreted from a geometrical aspect as a n-dimensional hyper rectangle. A formal definition is:

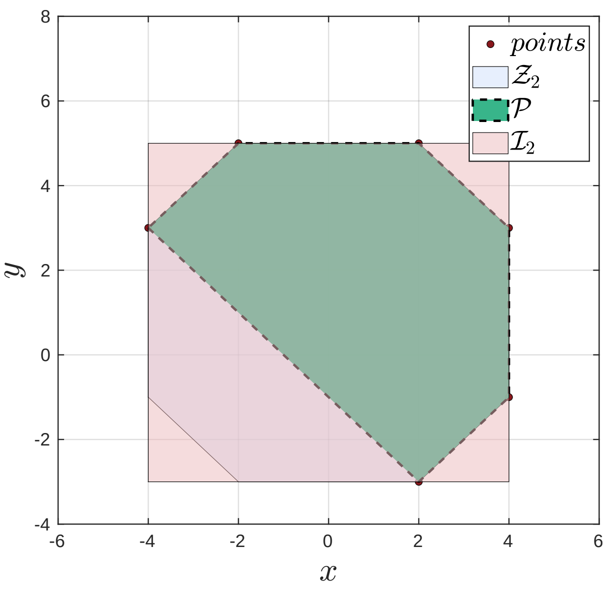
Theorem 2.1.4. *Consider an interval hull $\mathcal{I}$ which is bounded by a minimum $\boldsymbol{x}_{min} \in \mathbb{R}^n$ and a maximum $\boldsymbol{x}_{max} \in \mathbb{R}^n$, $\mathcal{I}$ is defined as:*

$$\mathcal{I} = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{x}_{min} \leq \boldsymbol{x} \leq \boldsymbol{x}_{max}\} \tag{2.5}$$

In short, $\mathcal{I}$ can also be written as $\mathcal{I} = [\boldsymbol{x}_{min}, \boldsymbol{x}_{max}]$. In contrast to zonotopes, computations with intervals can be associated to interval arithmetic. Compared to affine arithmetic, interval arithmetic suffers from the well known dependency problem which over approximates the exact solution significantly. This is also known as the wrapping effect. In Fig. 2.2 the wrapping effect of an interval hull is illustrated based on the previous seen shapes.



(a)



(b)

Fig. 2.2. Example illustrating the over approximated hulling of zonotopes by interval hulls.

In Fig. 2.2a and Fig. 2.2b, the interval hulls $\mathcal{I}_1$ and $\mathcal{I}_2$ are used to hull the zonotope $\mathcal{Z}_1$ and $\mathcal{Z}_2$, respectively. As observed in both figures, the result is over approximated. Moreover, hulling the sampled points in Fig. 2.2b with the interval hulls $\mathcal{I}_2$ results as well in over approximations compared to using the polytope $\mathcal{P}$ or the zonotope $\mathcal{Z}_2$. For the given example, polytopes seem to result in exact solutions. However, for the case the shape obtained by connecting several sample points is non-convex (concave), a convex polytope would over approximate the solution as well. This must be kept in mind, as the polytopes used in this dissertation are always convex.

2.2 Eigenvalues and Eigenvectors

The study of eigenvalues and eigenvectors has a great importance in the analysis of different kinds of problems. For dynamic systems, eigenvalues reveal insight information regarding the system behavior and its variation over time. For an LTI system, this is described in Section 2.3.1. In the context of linear algebra and computer vision, eigenvalues are often associated with matrices and matrix transformations. Nevertheless, the study of eigenvalues and eigenvectors finds its application in vibrations analysis, heat flow, economics, computer graphics, etc. In the following, the standard eigenvalue problem as well as the generalized eigenvalue problem will be briefly examined.

2.2.1 The Standard Eigenvalue Problem

In this dissertation, eigenvalues and eigenvectors find as well a key role to play, especially in abstraction process presented in Chapter 4. In this sense, the definitions of the eigenvectors and eigenvalues are both reviewed. For this purpose, consider the matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ and a nonzero vector $\boldsymbol{v} \in \mathbb{R}^n$. In general, the linear transformation of a vector $\boldsymbol{v}$ upon the multiplication with the square matrix $\boldsymbol{A}$ results in a vector that differs from $\boldsymbol{v}$ in both magnitude and direction. However, for the case that $\boldsymbol{v}$ is an eigenvector of $\boldsymbol{A}$, the multiplication of this vector with $\boldsymbol{A}$ does not change the direction of the vector (except the orientation), but modifies the magnitude of $\boldsymbol{v}$ as described by the scalar value λ called eigenvalue. A more formal definition from [LF09] is given in Theorem 2.2.1:

Theorem 2.2.1. *Let $\boldsymbol{A}$ be an $n \times n$ matrix. The scalar λ is called an eigenvalue of $\boldsymbol{A}$ if there is a nonzero vector $\boldsymbol{v}$ such that:*

$$\boldsymbol{Av} = \lambda \boldsymbol{v} \tag{2.6}$$

the vector $\boldsymbol{v}$ is an eigenvector of $\boldsymbol{A}$ corresponding to λ.

The term eigenvector used through this dissertation refers to the right eigenvectors unless stated otherwise. Hence, an eigenvector of a matrix $\boldsymbol{A}$ is only scaled by its corresponding eigenvalue λ upon undergoing a linear transformation through the matrix multiplication with $\boldsymbol{A}$. Note that Eq. (2.6) is called the (standard) eigenvalue problem.

To find the eigenvalues, consider again Eq. (2.6) which can be rewritten as:

$$\boldsymbol{Av} = \lambda \boldsymbol{Iv} \implies (\lambda \boldsymbol{I} - \boldsymbol{A})\boldsymbol{v} = \boldsymbol{0} \tag{2.7}$$

The task of determining the eigenvalues becomes finding the values of λ which satisfies Eq. (2.7) for a nonzero vector $\boldsymbol{x}$. This demands $(\lambda \boldsymbol{I} - \boldsymbol{A})$ to be singular. By that, we can state the following theorem:

Theorem 2.2.2. *Consider the $n \times n$ matrix $\boldsymbol{A}$, the characteristic equation of $\boldsymbol{A}$ is:*

$$det(\lambda \boldsymbol{I} - \boldsymbol{A}) = 0 \tag{2.8}$$

The scalar λ is an eigenvalue of $\boldsymbol{A}$ if and only if it satisfies Eq. (2.8). Expanding Eq. (2.8) yields the characteristic polynomial of $\boldsymbol{A}$:

$$\lambda^n + c_{n-1}\lambda^{n-1} + \cdots + c_1\lambda^1 + c_0 = 0 \tag{2.9}$$

The roots of the characteristic polynomial are the eigenvalues of $\boldsymbol{A}$. The eigenvectors of $\boldsymbol{A}$ corresponding to the eigenvalue λ are the nonzero solutions of $\boldsymbol{v}$ that satisfy:

$$(\lambda \boldsymbol{I} - \boldsymbol{A})\boldsymbol{v} = \boldsymbol{0} \tag{2.10}$$

The *algebraic multiplicity* μ_i of an eigenvalue λ_i refers to the μ_i-repeated root λ_i of the characteristic polynomial. This means that the algebraic multiplicity states how often an eigenvalue is repeated in the characteristic polynomial.
The *geometric multiplicity* γ_i, on the other hand, states the number of linearly independent eigenvectors associated with an eigenvalue λ_i. That is, γ_i states the dimension of the nullspace $(\lambda_i \boldsymbol{I} - \boldsymbol{A})$:

$$\gamma_i = dim(ker(\lambda_i \boldsymbol{I} - \boldsymbol{A})) \implies \gamma_i = n - rank(\lambda_i \boldsymbol{I} - \boldsymbol{A}) \tag{2.11}$$

Each eigenvalue has at least a geometric multiplicity of one. That is, each eigenvalue has at least one associated eigenvector.

Eigenvalue Decomposition of a Matrix

The aim of the eigenvalue decomposition is to transform a matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ into its diagonal (if possible) form. Decomposing a matrix means to factorize it i.e. to find a product of matrices that is similar to the initial one. Hence, the eigenvalue decomposition of a matrix decomposes the matrix into the product of its eigenvectors and eigenvalues.

Not all matrices are diagonalizable. For a matrix to be diagonalizable, it must satisfy Theorem 2.2.3 as stated in [ARIS14].

Theorem 2.2.3. *An $n \times n$ matrix $\boldsymbol{A}$ is said to be diagonalizable if and only if it has n linearly independent eigenvectors. If $\lambda_1, \lambda_2, \dots, \lambda_n$ are distinct eigenvalues of $\boldsymbol{A}$, then the corresponding eigenvectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \dots, \boldsymbol{v}_n$ are linearly independent and $\boldsymbol{A}$ is diagonalizable. In general, $\boldsymbol{A}$ is said to be diagonalizable if and only if the geometric multiplicity of every eigenvalue is equal to its algebraic multiplicity.*

For the matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ with n linearly independent eigenvectors, consider the matrix of the right eigenvectors $\boldsymbol{F} \in \mathbb{R}^{n\times n}$ with columns consisting of these eigenvectors, that is:

$$\boldsymbol{F} = \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 & \dots & \boldsymbol{v}_n \end{bmatrix} \tag{2.12}$$

The diagonal matrix $\boldsymbol{\Lambda}$, with the eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_n$ on the diagonal ordered like the corresponding eigenvectors in $\boldsymbol{F}$, is obtained by replacing $\boldsymbol{v}$ and λ with $\boldsymbol{F}$ and $\boldsymbol{\Lambda}$ in Eq. (2.7), respectively, and solving for $\boldsymbol{\Lambda}$. This yields:

$$\begin{aligned} \boldsymbol{\Lambda} &= \boldsymbol{F}^{-1}\boldsymbol{A}\boldsymbol{F} \\ &= \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_n \end{bmatrix} \end{aligned} \tag{2.13}$$

Jordan Normal Form of a Matrix

Not every matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ is diagonalizable. A more general form is the Jordan normal form (JNF). For every matrix $\boldsymbol{A}$, there is a transformation matrix $\boldsymbol{T}$ such that $\boldsymbol{J} = \boldsymbol{T}^{-1}\boldsymbol{A}\boldsymbol{T}$ is in the

Jordan normal form:

$$\boldsymbol{J} = \begin{bmatrix} \boldsymbol{J}_1 & & & \\ & \boldsymbol{J}_2 & & \\ & & \ddots & \\ & & & \boldsymbol{J}_k \end{bmatrix} \tag{2.14}$$

Where $\boldsymbol{J}_1$, $\boldsymbol{J}_2$,..., $\boldsymbol{J}_k$ are called Jordan blocks. A Jordan block $\boldsymbol{J}_i$ corresponding to an eigenvalue λ_i has the form:

$$\boldsymbol{J}_i = \begin{bmatrix} \lambda_i & 1 & & & \\ & \lambda_i & 1 & & \\ & & \ddots & \ddots & \\ & & & \lambda_i & 1 \\ & & & & \lambda_i \end{bmatrix} \tag{2.15}$$

The geometric multiplicity γ_i of the eigenvalue λ_i states the number of Jordan blocks $\boldsymbol{J}_i$ corresponding to λ_i. On the other hand, the algebraic multiplicity μ_i states the sum of the sizes of the Jordan blocks $\boldsymbol{J}_i$ corresponding to λ_i.

For the case that λ_i is complex, the Jordan blocks can be reformed to purely real matrices:

$$\tilde{\boldsymbol{J}}_i = \begin{bmatrix} \boldsymbol{L}_i & \boldsymbol{I}_i & & & \\ & \boldsymbol{L}_i & \boldsymbol{I}_i & & \\ & & \ddots & \ddots & \\ & & & \boldsymbol{L}_i & \boldsymbol{I}_i \\ & & & & \boldsymbol{L}_i \end{bmatrix} \tag{2.16}$$

Where $\boldsymbol{I}_i$ is an identity matrix, and $\boldsymbol{L}_i$ for the eigenvalue $\lambda_i = \sigma_i + j\omega_i$ is:

$$\boldsymbol{L}_i = \begin{bmatrix} \sigma_i & \omega_i \\ -\omega_i & \sigma_i \end{bmatrix} \tag{2.17}$$

The origin of $\boldsymbol{L}_i$ can be traced back to the similarity transformation as explained in Appendix A.2.

2.2.2 The Generalized Eigenvalue Problem

In Section 2.2.1, the standard eigenvalue problem was discussed and analyzed. In fact, the standard eigenvalue problem is a special case of the generalized eigenvalue problem, which is often encountered when working with DAE systems. For the dissertation at hand, this section plays a significant role in understanding the calculations performed during the sampling process presented later in Chapter 3. In the following, the generalized eigenvalue problem will be examined in a similar fashion to Section 2.2.1.

Consider two $n \times n$ matrices $\boldsymbol{A}$ and $\boldsymbol{E}$, sometimes denoted as a matrix pair or a matrix pencil $(\boldsymbol{E},\boldsymbol{A})$. A scalar λ is a generalized eigenvalue of the pair $(\boldsymbol{E},\boldsymbol{A})$ if:

$$\boldsymbol{A}\boldsymbol{v} = \lambda \boldsymbol{E}\boldsymbol{v} \tag{2.18}$$

Note that the standard eigenvalue problem is a special case of the generalized eigenvalue problem for the case that $\boldsymbol{E} = \boldsymbol{I}$.

The nonzero column vector that satisfies Eq. (2.18) is called a right generalized eigenvector [KHD17]. For convenience, $\boldsymbol{v}$ is just referred to as an eigenvector. Reforming Eq. (2.18) leads to the well known generalized eigenvalue problem:

$$(\lambda \boldsymbol{E} - \boldsymbol{A})\boldsymbol{v} = \boldsymbol{0} \tag{2.19}$$

Theorem 2.2.4 states a more formal definition of this problem along with some of its significant properties.

Theorem 2.2.4. *Consider the two $n \times n$ matrices $\boldsymbol{A}$ and $\boldsymbol{E}$ that compose the matrix pencil $(\boldsymbol{E},\boldsymbol{A})$,* and the associated characteristic polynomial $p(\lambda) = det(\lambda \boldsymbol{E} - \boldsymbol{A})$

1. the matrix pencil is regular if $det(\boldsymbol{A} - \lambda \boldsymbol{E}) \neq 0$. Otherwise, it is called singular
2. a pencil $(\boldsymbol{E}, \boldsymbol{A})$ with nonsingular $\boldsymbol{E}$ is always regular [Ban14]
3. if $\boldsymbol{E}$ is singular, the matrix pencil $(\boldsymbol{E}, \boldsymbol{A})$ has an eigenvalue at infinity with multiplicity:

$$\mu = (n - rank(\boldsymbol{E})) \tag{2.20}$$

   Note that if d denotes the degree of $p(\lambda)$, then the pencil $(\boldsymbol{E}, \boldsymbol{A})$ has $(n - d)$ eigenvalues at infinity [Bai00]

4. if $\boldsymbol{E}$ is nonsingular, the generalized eigenvalue problem can be simplified to the standard eigenvalue problem with:

$$\boldsymbol{A}\boldsymbol{v} = \lambda \boldsymbol{E}\boldsymbol{v} \implies \boldsymbol{E}^{-1}\boldsymbol{A}\boldsymbol{v} = \lambda \boldsymbol{v} \tag{2.21}$$

5. if $\lambda = \infty$ is an eigenvalue, the nonzero vectors $\boldsymbol{v}$ satisfying

$$\boldsymbol{E}\boldsymbol{v} = 0 \tag{2.22}$$

   are the corresponding right eigenvectors [Bai00]

6. if $\boldsymbol{A}$ is singular, then $\lambda = 0$ is an eigenvalue of the system

To deal with both finite and infinite eigenvalues, some approaches decompose an eigenvalue λ into the pair (α, β) such that $\lambda = \frac{\alpha}{\beta}$. Thus, the generalized eigenvalue problem can be rewritten as

$$\beta \boldsymbol{A}\boldsymbol{v} = \alpha \boldsymbol{B}\boldsymbol{v} \tag{2.23}$$

Eq. (2.23) states that for the case $\beta \neq 0$, $\lambda = \frac{\alpha}{\beta}$ is a finite eigenvalue. While for the case that $\beta = 0$, $\lambda = \infty$ is an eigenvalue of the system.

Weierstrass-Kronecker Form

There are a lot of decomposition theorems that can bring a matrix pencil to different forms, an example of such a form is: the diagonal form, the Weierstrass form, the generalized Schur form, and the Weierstrass-Schur form [Bai00]. Although the study of these forms is out of the range of this dissertation, the Weierstrass-Kronecker canonical form will be briefly reviewed.

Theorem 2.2.5. *Consider the matrix pencil $(\boldsymbol{E}, \boldsymbol{A})$ with $\boldsymbol{A}, \boldsymbol{E} \in \mathbb{R}^{n\times n}$. If the matrix pencil $(\boldsymbol{E}, \boldsymbol{A})$ is regular, then there exist two nonsingular matrices $\boldsymbol{F}$ and $\boldsymbol{H}$ such that:*

$$\begin{aligned} \tilde{\boldsymbol{E}} &= \boldsymbol{FEH} & \tilde{\boldsymbol{A}} &= \boldsymbol{FAH} \\ &= \begin{bmatrix} \boldsymbol{I}_\Lambda & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{N} \end{bmatrix} & &= \begin{bmatrix} \boldsymbol{J} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I}_\infty \end{bmatrix} \end{aligned} \tag{2.24}$$

The matrix pencils $(\boldsymbol{E}, \boldsymbol{A})$ and $(\tilde{\boldsymbol{E}}, \tilde{\boldsymbol{A}})$ are said to be equivalent [Ria08]. Where $\boldsymbol{J} \in \mathbb{R}^{m\times m}$ is in general in the JNF (Section 2.2.1) corresponding to the finite eigenvalues of Eq. (2.19), while $\boldsymbol{N} \in \mathbb{R}^{(n-m)\times(n-m)}$ is nilpotent matrix corresponding to the infinite eigenvalues.

As mentioned in Theorem A.1.1 in Appendix A.1, η represents the index of nilpotency of $\boldsymbol{N}$. The nilpotent matrix $\boldsymbol{N}$ represents the singularity of the DAE [IR17]. Moreover, the index of nilpotency is identical to the index of the DAE. As the Weierstrass-Kronecker canonical form is often referred to in literature as Kronecker canonical form [Ria08], it is abbreviated as KCF.

2.3 Linear and Nonlinear Systems

Physical systems and procedures that exist in nature as well as industrial processes are often described via models for analysis and examinations. The simplest approach to describe the behavior of a system is to model it as a linear system. Linear models can describe a large part of the systems from the real world. In fact, linear system and control theory has a broad application spectrum. However, in general these methodologies cannot be applied on nonlinear systems [Ada09]. While some systems cannot be described accurately with linear models, nonlinear model descriptions usually result in more precise and better behavioral results, emphasizing the importance of nonlinear system theory. An exception to this are nonlinear systems that can be, to some degree, approximated by linear systems. An accurate linearization of a nonlinear system allows for the applications of the efficient, easy to deploy, less complex algorithms from the linear system theory.

In general, several representations exist to model the behavior of a system. For linear time-invariant (LTI) systems, the most common used ones are linear ordinary differential equations (ODEs), transfer functions in the time as well as in the Laplace domain, and the state space representation in different forms: controllable normal form, observable normal form, and Jordan normal form (JNF). For nonlinear systems, usually more general forms, such as differential algebraic equations (DAEs), are used. Of course, linearizing a nonlinear system would additionally allow for representations such as the linear descriptor system representation, which can to some extent describe the system behavior accurately. In this section, several model descriptions for linear as well as for nonlinear systems are handled.

2.3.1 The State Space Representation of an LTI System

A linear system can be represented using the state space representation. With n state space variables described by the state space vector $\boldsymbol{x}(t) \in \mathbb{R}^n$, and the initial condition of this vector given by $\boldsymbol{x}(0) = \boldsymbol{x}_0$, the state space representation of a multiple input multiple output (MIMO) continuous time-invariant system is:

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{Ax}(t) + \boldsymbol{Bu}(t) \tag{2.25a}$$

$$\boldsymbol{y}(t) = \boldsymbol{Cx}(t) + \boldsymbol{Du}(t) \tag{2.25b}$$

Where the p output variables and k input variables are described by the output and input vectors, $\boldsymbol{y}(t) \in \mathbb{R}^p$ and $\boldsymbol{u}(t) \in \mathbb{R}^k$, respectively. Moreover, the matrices from Eq. (2.25) are:

- the system matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$
- the input matrix $\boldsymbol{B} \in \mathbb{R}^{n \times k}$
- the output matrix $\boldsymbol{C} \in \mathbb{R}^{p \times n}$
- the feedthrough matrix $\boldsymbol{D} \in \mathbb{R}^{p \times k}$

Note that these matrices are time-invariant. Time variant system descriptions will not be covered in this dissertation. Eq. (2.25) can be extended by some terms to model the noise as stated in [Lun10]. This representation will be skipped here as well, as the noise is not modeled.

Let $\mathcal{S}$ denote the state space of a system. At any instance, the system state can be interpreted as a point in the n dimensional state space $\mathcal{S}$. The state response $\boldsymbol{x}(t)$ over the time t, on the other hand, refers to the trajectory through the state space $\mathcal{S}$. Note that t does not appear explicitly in the state space of the system.

Time-domain Solution of an LTI State Space Equation

The state response i.e. the solution of the differential system given by Eq. (2.25a) with the initial condition $\boldsymbol{x}_0$, can be stated in terms of the homogeneous solution $\boldsymbol{x}_{hom}(t)$ and the particular solution $\boldsymbol{x}_{par}(t)$ [Ada13] as:

$$\boldsymbol{x}(t) = \underbrace{e^{\boldsymbol{A}t}\boldsymbol{x}_0}_{\boldsymbol{x}_{hom}} + \underbrace{\int_0^t e^{\boldsymbol{A}(t-\tau)}\boldsymbol{B}\boldsymbol{u}(\tau)d\tau}_{\boldsymbol{x}_{par}} \tag{2.26}$$

As observed, the state response depends on two components: the homogeneous response $\boldsymbol{x}_{hom}(t)$ which in terms depends on the initial condition $\boldsymbol{x}_0$, and the particular solution $\boldsymbol{x}_{par}(t)$ which depends on the input vector $\boldsymbol{u}(t)$. Note that the particular solution is computed using a convolution integral. Since the homogeneous solution is simply computed by the multiplication of the matrix $e^{\boldsymbol{A}t}$ with the initial condition $\boldsymbol{x}_0$, a state transition matrix $\boldsymbol{\Phi}(t)$ can be defined:

$$\boldsymbol{\Phi}(t) = e^{\boldsymbol{A}t}\,, \tag{2.27}$$

which upon the multiplication with $\boldsymbol{x}_0$, can deliver at any time t the homogeneous response of the system. For the particular solution $\boldsymbol{x}_{par}$, one can define $\boldsymbol{\Phi}(t-\tau)$:

$$\boldsymbol{\Phi}(t-\tau) = e^{\boldsymbol{A}(t-\tau)}\,, \tag{2.28}$$

and compute the convolution integral of Eq. (2.26). Inserting this solution in the output equation Eq. (2.25b), yields the system output response:

$$\boldsymbol{y}(t) = \boldsymbol{C}e^{\boldsymbol{A}t}\boldsymbol{x}_0 + \boldsymbol{C}\int_0^t e^{\boldsymbol{A}(t-\tau)}\boldsymbol{B}\boldsymbol{u}(\tau)d\tau + \boldsymbol{D}\boldsymbol{u}(t) \tag{2.29}$$

Thus, in order to compute the state and output responses of the system at hand, the main task becomes finding the transition matrix from Eq. (2.27). Using the matrix exponential, the transition matrix can be expanded to an infinite series [Mey08]:

$$\begin{aligned}\boldsymbol{\Phi}(t) &= e^{\boldsymbol{A}t} \\ &= \sum_{i=0}^{\infty} \frac{(\boldsymbol{A}t)^i}{i!} \\ &= \boldsymbol{I} + \boldsymbol{A}t + \boldsymbol{A}^2\frac{t^2}{2!} + \dots\end{aligned} \tag{2.30}$$

For the case that $\boldsymbol{A}$ is a diagonal matrix, the transition matrix can be easily computed by applying the exponential function on the entities of the diagonal of the matrix $\boldsymbol{A}$ [Ada13]:

$$\boldsymbol{A} = \begin{bmatrix} \lambda_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \lambda_n \end{bmatrix} \implies \boldsymbol{\Phi}(t) = \begin{bmatrix} e^{\lambda_1 t} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & e^{\lambda_n t} \end{bmatrix} \tag{2.31}$$

If the matrix $\boldsymbol{A}$ is not a diagonal matrix, in some cases it can be transformed into a diagonal one (Section 2.2.1). If the matrix is not diagonalizable, one can first calculate the JNF of $\boldsymbol{A}$, and then compute the solution of the system in a similar fashion.

It is important to notice that the eigenvalues of $\boldsymbol{A}$ play a significant role in defining the solution of the LTI system. Thus, if the eigenvalues are in general known, the solution of the LTI system can be easily computed. Considering this fact from a different perspective, the system matrix $\boldsymbol{A}$ can be reconstructed from the eigenvalues that characterize the solution of the LTI system. This fact states the underlying concept of the abstraction approach stated in Chapter 4.

The Transfer Function of an LTI system

An LTI system is often transformed into the Laplace domain. This is because this domain represents a powerful tool to analyze and solve an LTI system, due to some special properties obtained during the transformation from one domain to the other. For example considering a transformation from the time domain to the Laplace domain, a convolution integral results in a multiplication, while a differential equation results in an algebraic equation. Note that in both cases the solution is easier to compute in the Laplace domain. Moreover, properties such as the frequency and phase response can be easily represented in the Laplace domain.

With $s = \sigma + j\omega$ denoting the complex frequency parameter with the real numbers σ and ω, the state space in the time domain from Eq. (2.25) can be transformed into the Laplace domain:

$$\boldsymbol{X}(s) = (s\boldsymbol{I} - \boldsymbol{A})^{-1}\boldsymbol{x}_0 + (s\boldsymbol{I} - \boldsymbol{A})^{-1}\boldsymbol{B}\boldsymbol{U}(s) \tag{2.32a}$$

$$\boldsymbol{Y}(s) = \underbrace{\boldsymbol{C}(s\boldsymbol{I} - \boldsymbol{A})^{-1}\boldsymbol{x}_0}_{\text{initial condition response}} + \underbrace{[\boldsymbol{C}(s\boldsymbol{I} - \boldsymbol{A})^{-1}\boldsymbol{B} + \boldsymbol{D}]}_{\text{transfer function } \boldsymbol{G}(s)}\boldsymbol{U}(s) \tag{2.32b}$$

As stated in [Tsu03], the transfer function $\boldsymbol{G}(s)$ gives a direct relationship between the outputs of the system described by the output vector $\boldsymbol{Y}(s)$ and the inputs to the system described by the input vector $\boldsymbol{U}(s)$. The initial condition response is also known as the zero-input response while the transfer function $\boldsymbol{G}(s)$ is also known as the zero-state response. For the case that the initial condition $\boldsymbol{x}_0$ is zero, the outputs of the system are described by the second term of Eq. (2.32b) defined by the zero-state response and the inputs.

Comparing Eqs. (2.26, 2.29) with Eq. (2.32) shows again the importance of the transition matrix $\boldsymbol{\Phi}(t)$ and the relationship to the eigenvalue problem from Eq. (2.7):

$$\boldsymbol{\Phi}(t) = e^{\boldsymbol{A}t} \quad \circ\!\!-\!\!\bullet \quad \boldsymbol{\Phi}(s) = (s\boldsymbol{I} - \boldsymbol{A})^{-1} \tag{2.33}$$

Stability of an LTI System

There exist a solid number of theorems that can determine the stability of an LTI system. Some well known theorems are for example the Nyquist stability criterion and Routh-Hurwitz stability criterion. Examining the stability of a system is of significant importance. For example, one might find a Lyapunov function and proof that the system is or is not Lyapunov stable. BIBO (bounded input bounded output) stability, which states that any bounded input generates a bounded output, is also an important stability type. A special importance, however, is given to the asymptotic stability. For an LTI system, in contrast a to nonlinear system, the stability condition for BIBO and asymptotic stability are the same, demanding that all eigenvalues of the system matrix $\boldsymbol{A}$ have only negative real parts. Hence, all eigenvalues are located in the left half in the complex s-plane.

Transformation of the State Space

The state space representation of a system is not unique. In fact, by reordering the state variables or defining new state variables in terms of the old ones, infinite many possible state space models can be constructed. Hence, some standardized state space representation forms have been introduced. These forms are known as canonical forms: controllable normal form, observable normal form, and the diagonal, or in general the Jordan normal form (JNF). For the aim this dissertation, only the JNF and the diagonal form are of interest. As stated in Theorem 2.2.3, the system matrix $\boldsymbol{A}$ from Eq. (2.25) can be transformed into its diagonal form, if the system has n distinct eigenvectors. If this is not the case, in general $\boldsymbol{A}$ can be transformed to the Jordan normal form.

Generally speaking, transforming a model from the state space $\mathcal{S}_o$ with the state space variable $\boldsymbol{x}$ to a new state space $\mathcal{S}_s$ with the state space variable $\boldsymbol{x}_s$ can be preformed by applying a linear transformation:

$$\boldsymbol{x} = \boldsymbol{T}\boldsymbol{x}_s, \tag{2.34}$$

with the transformation matrix $\boldsymbol{T}$. Upon undergoing a linear transformation with a regular transformation matrix $\boldsymbol{T}$, that is $det(\boldsymbol{T}) \neq 0$, the properties of the linear system including eigenvalues, observability, and controllability are preserved.

2.3.2 Linear Descriptor System Representation

The state space representation from Eq. (2.25), which describes a linear system by first order ordinary differential equations, can be extended to describe a system with linear DAEs. This is done by using a linear descriptor system representation:

$$\boldsymbol{E}\dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{B}\boldsymbol{u}(t) \tag{2.35a}$$

$$\boldsymbol{y}(t) = \boldsymbol{C}\boldsymbol{x}(t) + \boldsymbol{D}\boldsymbol{u}(t) \tag{2.35b}$$

With the mass matrix $\boldsymbol{E} \in \mathbb{R}^{n\times n}$. The dimensions of the matrices $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{D}$ are as stated in Section 2.3.1. Compared to Eq. (2.25a), the multiplication of the matrix $\boldsymbol{E}$ with the derivative of the state vector ($\dot{\boldsymbol{x}}(t)$) transforms the equation from an ODE to a DAE. Moreover, the rank of $\boldsymbol{E}$:

$$rank(\boldsymbol{E}) = r \quad \leq \quad n,$$

is of significant importance, as it states that Eq. (2.35a) consists of (r) differential equations and ($n-r$) algebraic equations. In the case for $\boldsymbol{E} = \boldsymbol{I}$, where $\boldsymbol{I} \in \mathbb{R}^{n\times n}$ is an identity matrix, Eq. (2.35) becomes identical to the state space representation from Eq. (2.25).

As stated at the end of Section 2.2.2, Eq. (2.35a) can be transformed into the KCF. This comes in handy when performing a model order reduction on the system, as stated in [BAS16] for example, or when trying to separate and solve the DAE system.

2.3.3 Local Linearization of a Nonlinear Descriptor System

Systems, especially technical ones, can be described very accurately using mathematical models. In general this can be performed by using DAEs. In this dissertation, we focus on implicit DAEs of the following structure:

$$\boldsymbol{f}(\dot{\boldsymbol{x}}(t), \boldsymbol{x}(t), \boldsymbol{u}(t)) = \boldsymbol{0} \tag{2.36a}$$

$$\boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{u}(t)) = \boldsymbol{y}(t) \tag{2.36b}$$

Eq. (2.36) is also known in general as a nonlinear descriptor system with the descriptor vector $\boldsymbol{x} \in \mathbb{R}^n$ containing the system variables (descriptor variables). With the input vector $\boldsymbol{u} \in \mathbb{R}^k$ and the descriptor vector $\boldsymbol{x}$, the output vector $\boldsymbol{y}$ can be defined using Eq. (2.36b).

Electric circuits, control systems, and many physical process are often described using this representation. However, working with nonlinear systems descriptions often challenges the design process. An easy approach to describe a nonlinear system linearly in a specific portion of the state space is local linearization. For example, Eq. (2.36) can be linearized using the Taylor series around the operating point $\boldsymbol{x}_{DC}$ for an operating input $\boldsymbol{u}_{DC}$. By using the first order Taylor approximation for multivariable functions on Eq. (2.36), neglecting thereby the higher order terms, the equation is linearized to:

$$\begin{aligned} \boldsymbol{f}(\dot{\boldsymbol{x}}, \boldsymbol{x}, \boldsymbol{u}) \approx \underbrace{\boldsymbol{f}(\dot{\boldsymbol{x}}_{DC}, \boldsymbol{x}_{DC}, \boldsymbol{u}_{DC})}_{\boldsymbol{0}} + \underbrace{\frac{\partial \boldsymbol{f}(\dot{\boldsymbol{x}}_{DC}, \boldsymbol{x}_{DC}, \boldsymbol{u}_{DC})}{\partial \dot{\boldsymbol{x}}}}_{\boldsymbol{E}} \Delta\dot{\boldsymbol{x}} \\ + \underbrace{\frac{\partial \boldsymbol{f}(\dot{\boldsymbol{x}}_{DC}, \boldsymbol{x}_{DC}, \boldsymbol{u}_{DC})}{\partial \boldsymbol{x}}}_{-\boldsymbol{A}} \Delta\boldsymbol{x} + \underbrace{\frac{\partial \boldsymbol{f}(\dot{\boldsymbol{x}}_{DC}, \boldsymbol{x}_{DC}, \boldsymbol{u}_{DC})}{\partial \boldsymbol{u}}}_{-\boldsymbol{B}} \Delta\boldsymbol{u}, \end{aligned} \tag{2.37}$$

and

$$\begin{aligned} \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{u}) \approx \underbrace{\boldsymbol{g}(\boldsymbol{x}_{DC}, \boldsymbol{u}_{DC})}_{\boldsymbol{y}_{DC}} + \underbrace{\frac{\partial \boldsymbol{g}(\boldsymbol{x}_{DC}, \boldsymbol{u}_{DC})}{\partial \boldsymbol{x}}}_{\boldsymbol{C}} \Delta\boldsymbol{x} \\ + \underbrace{\frac{\partial \boldsymbol{g}(\boldsymbol{x}_{DC}, \boldsymbol{u}_{DC})}{\partial \boldsymbol{u}}}_{\boldsymbol{D}} \Delta\boldsymbol{u} \end{aligned} \tag{2.38}$$

Since $\dot{\boldsymbol{x}}_{DC} = 0$, $\boldsymbol{f}(0, \boldsymbol{x}_{DC}, \boldsymbol{u}_{DC}) = 0$ as given by Eqs. (2.36a, 2.37). Finally, the system can be brought to the descriptor system representation presented in Eq. (2.35):

$$\boldsymbol{E}\Delta\dot{\boldsymbol{x}}(t) = \boldsymbol{A}\Delta\boldsymbol{x}(t) + \boldsymbol{B}\Delta\boldsymbol{u}(t) \tag{2.39a}$$

$$\Delta\boldsymbol{y}(t) = \boldsymbol{C}\Delta\boldsymbol{x}(t) + \boldsymbol{D}\Delta\boldsymbol{u}(t) \tag{2.39b}$$

Such that:

$$\Delta\boldsymbol{x}(t) = \boldsymbol{x}(t) - \boldsymbol{x}_{DC} \qquad \Delta\boldsymbol{u}(t) = \boldsymbol{u}(t) - \boldsymbol{u}_{DC} \qquad \Delta\boldsymbol{y}(t) = \boldsymbol{y}(t) - \boldsymbol{y}_{DC}$$

The dimensions of the matrices $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{D}$, and $\boldsymbol{E}$ are as stated in Section 2.3.2. Eq. (2.39) is in general only valid in a small region around the linearization point. Hence, in order to describe a nonlinear system accurately, often several linearization points are needed. This approach will find its application in Section 3.2.2.

Eigenvalues of the System

Eigenvalues are defined in the context of linear systems. For nonlinear systems, linearizing a nonlinear system (for example using the Taylor series) and solving the underlying eigenvalue problem allows for the computation of the eigenvalues around the operating points of the linearized system. As the nonlinear system evolves, the eigenvalues and eigenvectors change. Thus, in the context of nonlinear systems, one might define a set of eigenvalues and eigenvectors computed around the linearization points.

Depending on the linearization, i.e. order of Taylor terms, the underlying eigenvalue problem changes from the generalized eigenvalue problem stated in Section 2.2.2 to a polynomial eigenvalue problem. This problem is often encountered during vibration analysis, MEMS simulation, and the solution of least squares problems with quadratic constrains. In this dissertation, only the generalized eigenvalue problem will be considered. For more details on this topic see [TM01; HGL04].

2.4 Hybrid Automata

Hybrid systems can be modeled using hybrid automata. A hybrid automaton (HA) is a generalized finite state machine with continuous state variables. It has a finite set of locations of which one is the specified starting location (initial location).

In each location, the system behavior is described by differential equations with continuous state variables. As long as the invariant condition of a location is valid, the system can stay in this location. Once the invariant condition becomes invalid, the automaton must leave the location.

When a location can be left is described by the guards, and how it is left is specified by the corresponding jump function. This function is applied once the corresponding guard is taken. The guards allow for location switch, but do not define when exactly this switch happens. This makes a HA non-deterministic. A HA can also be non-deterministic if different locations share the same invariant, multiple location transitions can occur at the same time, or the same transition results in different locations. Of course by choosing the invariants, jump functions, and guards in an appropriate way, a HA can be created that is deterministic. The formal definition of the HA is similarly defined as in [SK03] with some restrictions on the jumps and the guards:

Definition 1. *A hybrid automaton is a tuple* $HA = (Loc, loc_0, \boldsymbol{x}_\lambda, \boldsymbol{x}_{\lambda,0}, inv, tran, grd, \boldsymbol{J}, \boldsymbol{u}, \boldsymbol{f})$ *containing:*

- *the finite set of locations* $Loc = \{loc_1, \ldots, loc_\nu\}$ *with an initial location* loc_0
- *the continuous state variables* $\boldsymbol{x}_\lambda \in \mathbb{R}^m$ *corresponding to the state space* $\mathcal{S}_\lambda \subseteq \mathbb{R}^m$ *and their initial values* $\boldsymbol{x}_{\lambda,0}$
- *the invariant mapping inv:* $Loc \to 2^{\mathcal{S}_\lambda}$[1] *which assigns an invariant* $inv(loc) \subseteq \mathcal{S}_\lambda$ *to each location* $loc \in Loc$
- *the set of discrete transitions* $tran \subseteq loc \times loc$. *A transition* (loc_i, loc_j) *denotes a transition from* $loc_i \in Loc$ *to* $loc_j \in Loc$
- *the guard function grd:* $tran \to 2^{\mathcal{S}_\lambda}$ *that assigns a guard set* $grd(loc_i, loc_j)$ *for each transition from* loc_i *to* loc_j
- *the jump function* $\boldsymbol{J}$*:* $T \times \mathcal{S}_\lambda \to \mathcal{S}_\lambda$, *which returns the next continuous state when a transition is taken*
- *the continuous input variables* $\boldsymbol{u} \in \mathbb{R}^k$ *corresponding to the input space* $\mathcal{S}_u \subseteq \mathbb{R}^k$
- *the flow function* $\boldsymbol{f}$*:* $Loc \times \mathcal{S}_\lambda \times \mathcal{S}_u \to \mathbb{R}^m$, *which defines a continuous vector field for the time derivative of* $\boldsymbol{x}_\lambda$*:* $\dot{\boldsymbol{x}}_\lambda = \boldsymbol{f}(loc, \boldsymbol{x}_\lambda, \boldsymbol{u})$

Throughout this dissertation, invariants will be described by the geometric shapes stated in Section 2.1. For simplicity, an invariant of a location loc is denoted as inv_{loc}.

The guards are modeled as halfspaces, polytopes, zonotopes or interval hulls (Section 2.1). A guard is denoted as:

$$grd_h : \quad \underbrace{loc_i}_{\text{current location}} \quad \to \quad \underbrace{loc_j}_{\text{target location}} \tag{2.40}$$

With loc_i and loc_j representing the current location and the target location, respectively. The current location indicates the location the guard belongs to. The target location of the guard is the next location the system switches to if this guard is taken. As a location can have several guards, the guards of a location are distinguished by the guard index h.

The jump functions are restricted to linear mappings. With the matrix $\boldsymbol{Q}_r \in \mathbb{R}^{m \times m}$ and the reset vector $\boldsymbol{v}_r \in \mathbb{R}^m$, along with the new state vector $\boldsymbol{x}_{\lambda,new} \in \mathbb{R}^m$ after and $\boldsymbol{x}_{\lambda,old} \in \mathbb{R}^m$ before a transition is taken becomes, a jump function has in general the following form:

$$\boldsymbol{x}_{\lambda,new} = \boldsymbol{Q}_r \boldsymbol{x}_{\lambda,old} + \boldsymbol{v}_r \tag{2.41}$$

2.5 Cluster Analysis

One of the essential building blocks of the algorithm presented in Chapter 4 is the cluster analysis. In the last decade, cluster analysis has gained significant importance especially upon its application in emerging fields such as big data and the internet of things, as it has proven itself as an effective unsupervised machine learning algorithms. In this dissertation, cluster analysis plays a significant role in the model abstraction process handled later in Section 4.4.

[1] $2^{\mathcal{S}_\lambda}$ is a power set of $\mathcal{S}_\lambda$

Cluster analysis aims to divide a given data set into groups (clusters), such that points in the same group are similar to each other and dissimilar to points from other groups. Hence, groups formed have low within-cluster variance and similar patterns. Moreover, it is often desired to obtain a representing value for each cluster, the centroid.

When clustering a data set, the algorithm must answer some basic questions including the optimal number of clusters and the quality of the found cluster. For some algorithms a set of specification can be provided to aid this process. After a clustering analysis is launched on a data set, the algorithm tries to group the data based on similarities, patterns, and differences with no previous labels. Clustering analysis is also known as one of the main methods of unsupervised machine learning, as the algorithm tries to find the best suited result from a set of given data and specifications without human interaction. As the entire spectrum of cluster analysis exceeds this dissertation, only a few topics will be picked and reviewed in this section. These topics include two clustering algorithms: k-means and DBSCAN, and a cluster evaluation criterion: the silhouette coefficient. An extensive study on data mining and especially clustering analysis can be found in [TSKK19].

In order to explore the clustering algorithms, a common data set represented by the matrix $\boldsymbol{Eig} \in \mathbb{R}^{l\times 4}$ is considered. This data set consists of $l = 18500$ sampled data points:

$$\boldsymbol{Eig} = \begin{bmatrix} Re(\lambda_{1,1}) & Im(\lambda_{1,1}) & Re(\lambda_{1,2}) & Im(\lambda_{1,2}) \\ Re(\lambda_{2,1}) & Im(\lambda_{2,1}) & Re(\lambda_{2,2}) & Im(\lambda_{2,2}) \\ \vdots & \vdots & \vdots & \vdots \\ Re(\lambda_{l,1}) & Im(\lambda_{l,1}) & Re(\lambda_{l,2}) & Im(\lambda_{l,2}) \end{bmatrix} \tag{2.42}$$

For simplicity, the second and fourth columns of Eq. (2.42) are zero vectors. Plotting the first column values against the values from the third column is illustrated in Fig. 2.3. For this matrix, two clustering goals are targeted. the first goal is to effectively cluster the data points into clusters, while the second goal is to find a centroid for each identified cluster.

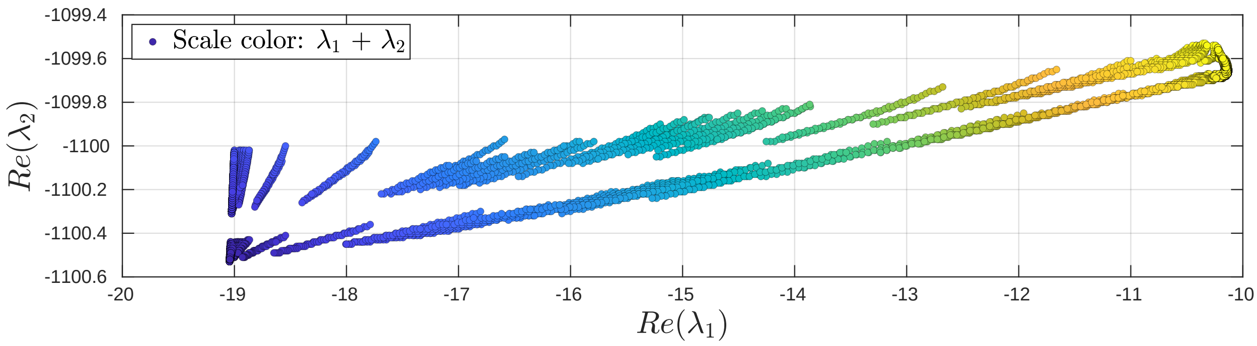


Fig. 2.3. Unclustered data set contained in $\boldsymbol{Eig}$.

2.5.1 Clustering Algorithm: K-means

The k-means algorithm, also known as Lloyd's algorithm [Llo82], is a partitioning based method. The algorithm tries to partition the data set into k_m clusters. Note that the desired number of clusters k_m is specified by the user. For the clustering of the 4-dimensional data set $\boldsymbol{Eig}$, the k-means algorithm is described in Algorithm 1.

Algorithm 1 K-means clustering algorithm

```
1: procedure k-means($\boldsymbol{Eig}$,$k_m$)
2:     choose $k_m$ initial centroids (cluster centers) randomly
3:     while centroids change and maximum number of iterations is not reached do
4:         for each point in $\boldsymbol{Eig}$ do
5:             calculate the distances from the point to all centroids
6:             assign the point to the closest centroid
7:         end for
8:         compute the average of all points in a cluster to obtain $k_m$ new centroids
9:     end while
10: end procedure
```

The standard implementation from Matlab extends Algorithm 1. The k-means Matlab implemented offers a variety of options. For example, additionally to the standard way to proceed at line 5 (batch update), Matlab offers an online update which reassigns points to centroids if several criteria are met. By default, this option is off. The distance metric used in line 5 of Algorithm 1 can also be specified. Even though the abstraction approach handled later in Chapter 4 can use various distance metrics, for simplicity only the default squared Euclidean distance metric will be used throughout this dissertation.

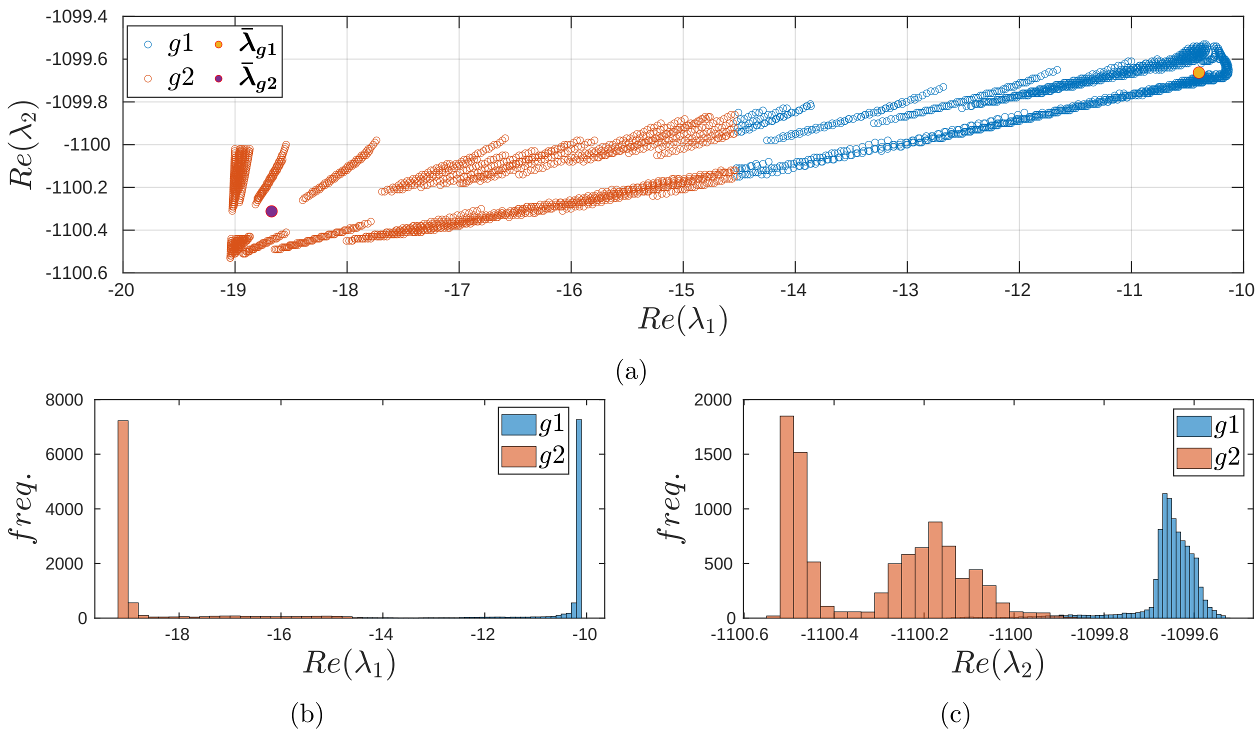
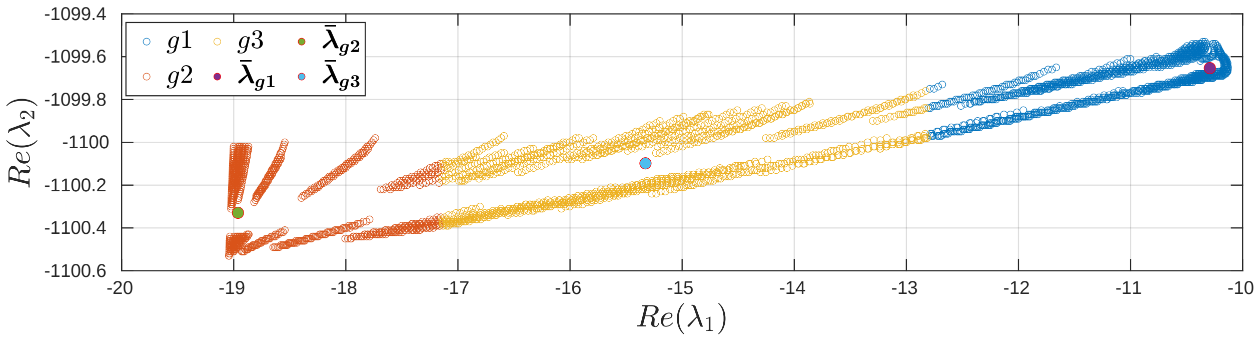


Fig. 2.4. Result of the k-means clustering algorithm for $k_m = 2$.

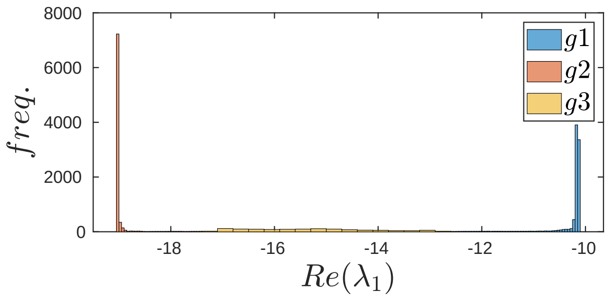
Fig. 2.4 shows the result of the k-means clustering process with $k_m = 2$ and the maximum number of replicates set to $r = 4$. During the 4 replicates, which were executed in parallel, the algorithm converged rapidly with a maximum number of iterations ranging between 3 to 5. Note that the number of maximum iterations was not explicitly specified and hence the default value (100) was used.

As illustrated in Fig. 2.4a, the non-zero column vectors of $\boldsymbol{Eig}$ (column 1 and 3), have been successfully clustered into two groups $g1$ and $g2$. The centroids of the clusters $g1$ and $g2$ have been labeled $\boldsymbol{\lambda}_{op,g1}$ and $\boldsymbol{\lambda}_{op,g2}$, respectively. Fig. 2.4b shows the labeled distribution of the first column of $\boldsymbol{Eig}$, while Fig. 2.4c presents the result for the third column.

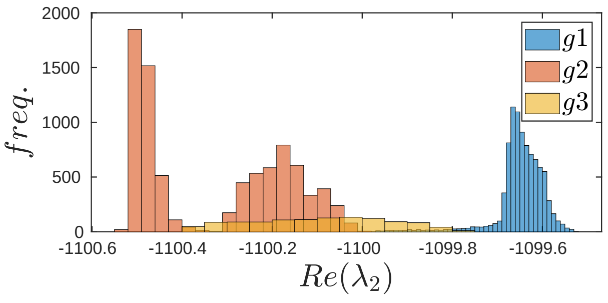
The result of the k-means with $k_m = 3$ is illustrated in Fig. 2.5 with the same number of replicates. This time the algorithm needs a maximum of 14 iterations to converge. Compared to previous case ($k_m = 2$) where the data points overlapped minimal along the $Re(\lambda_2)$ dimension, Fig. 2.5 shows that these two groups do not overlap at all if $k_m = 3$ is used. However, $g3$ overlaps significantly with the neighbor groups as shown clearly in Fig. 2.5c.



(a)



(b)



(c)

Fig. 2.5. Result of the k-means clustering algorithm for $k_m = 3$.

The result of clustering $\boldsymbol{Eig}$ with $k_m = 4$ and $k_m = 5$ is presented in Fig. 2.6a and Fig. 2.6b, respectively. For the case that $k_m = 4$, the algorithm needs a maximum of 19 iterations, while for $k_m = 5$ the algorithm needs a maximum of 23 iterations.

Considering the various results obtain for different values of the number of clusters k_m, a question is raised regarding the right number to choose. Moreover, there is a need for a metric that allows for the comparison of the results. This is where the silhouette coefficient discussed later in Section 2.5.3 comes in.

As seen, the k-means is a simple algorithm which is easily applicable on a wide variety of data. Solid results are obtained due to multiple runs as well as due to the specified desired number of clusters k_m. However, k-means is a partitioning algorithm, which makes it unsuitable for some data types, especially when the data contains outliers or the clusters are of various destinies. The problem with outliers can be solved by dropping out points, whereas the problem with the density of the data is linked to the partitioning nature of this algorithm. The more data are located

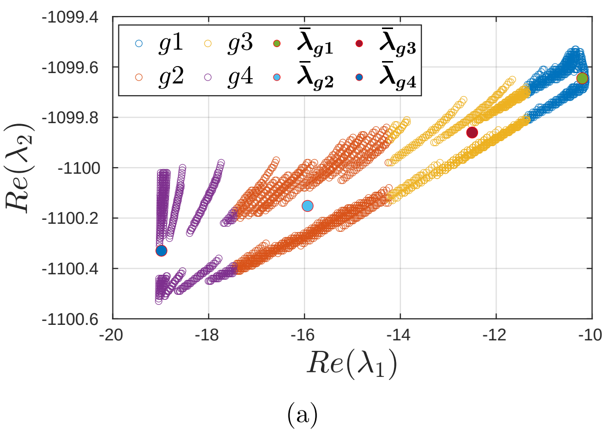


Fig. 2.6. Result of the k-means clustering algorithm for (a) $k_m = 4$ and (b) $k_m = 5$.

in a portion of the data space, the more likely it becomes that these points form a cluster. To demonstrate this, consider Fig. 2.7.

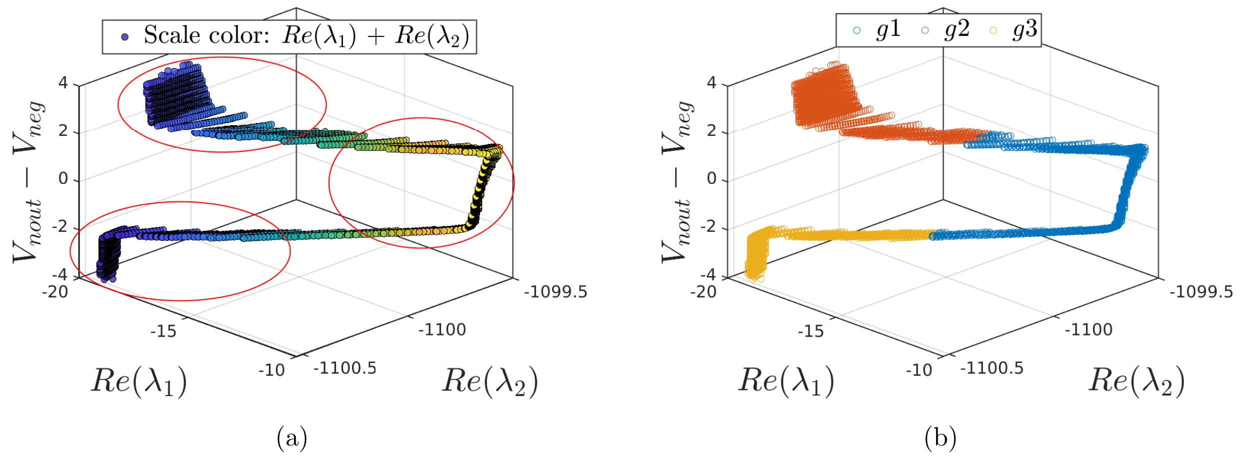


Fig. 2.7. Results of the k-means algorithm with $k_m = 3$.

An additional dimension named $(V_{nout} - V_{neg})$ has been added to the four dimensions of $\boldsymbol{Eig}$. For the new data set, we desire 3 clusters, as presented in Fig. 2.7a, and have therefore specified $k_m = 3$. As observed in Fig. 2.7b, k-means did successfully achieve this goal. Note that k-means could have formed the clusters similar to Fig. 2.5a, when looking at 2.7b from the top view $(Re(\lambda_1), Re(\lambda_2))$, but the distribution of the points favored this constellation.

Considering Fig. 2.8, when on the other hand 5 clusters are desired as shown in Fig. 2.8a, k-means failed to achieve this result as presented in Fig. 2.8b, even though the data points are well separated. Modifying the distance metric, or like in our case scaling the data set, can sometimes achieve the desired result as shown in Fig. 2.8c.

There are various extensions to the k-means clustering algorithm, such as the k-means++[AV07] which uses a heuristic to find the centroids of the k_m clusters improving the run time and the quality of the results. Note that, by default Matlab k-means algorithm uses k-means++ for cluster center (centroids) initialization. Another extended version is the k-medoids algorithm, which chooses the

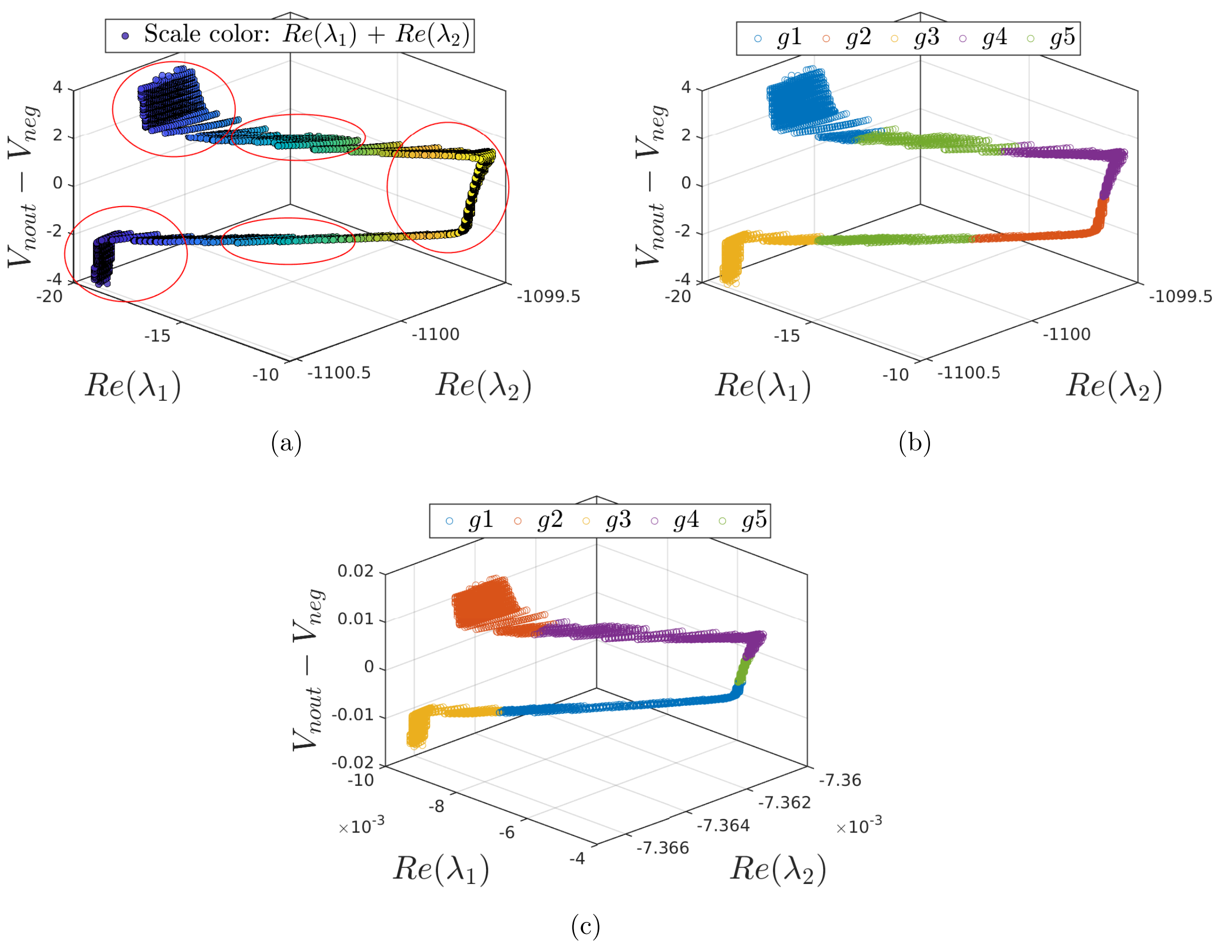


Fig. 2.8. Results of the k-means algorithm with $k_m = 5$. The desired result is shown in (a). The obtained results on the untreated data set is shown in (b). Upon scaling the data set, (c) is obtained.

center of each cluster as one of the data points. In [TSKK19], various extended k-means algorithms are listed and examined, like bisecting k-means.

2.5.2 Clustering Algorithm: DBSCAN

Another clustering algorithm is DBSCAN [EKX], which unlike k-means is a density based clustering algorithm. This clustering algorithm can distinguish the points belonging to a cluster and those which represent noise in a data set, as the name implies: Density-based spatial clustering of applications with noise. In fact, points are divided into three types: core points, border points, and noise points. These are defined as:

- Core points: points that have at least N points within a distance ϵ from themselves.
- Border points: points that has at least one core point at a distance ϵ from themselves, but do not have the required N points within this distance.
- Noise points: points that are neither core nor border points.

A cluster can contain both, core and border points, while noise points don't belong to any cluster. Fig. 2.9 shows an example of this point classification.

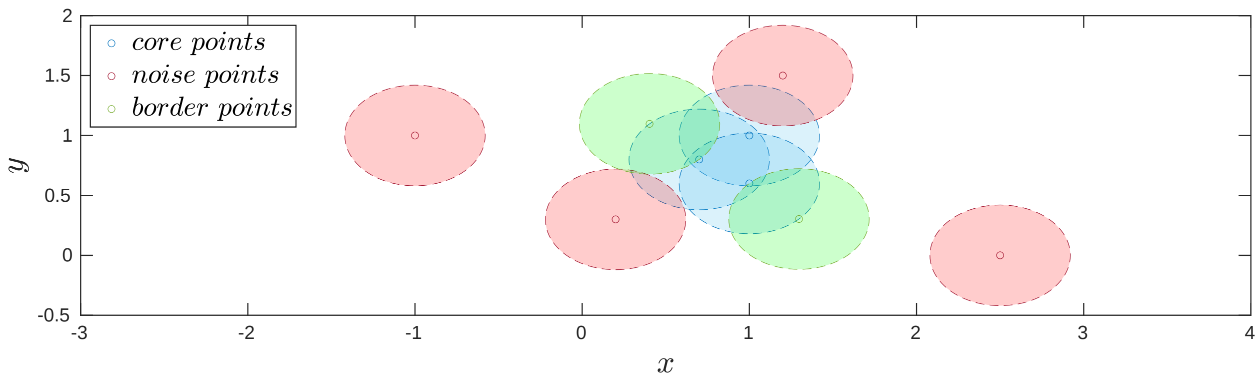


Fig. 2.9. Classification of points in DBSCAN. All circles have a radius of ϵ. The centers of the circles represent the coordinates of the example points.

The DBSCAN algorithm is given in Algorithm 2 for the data set ***Eig***, the threshold radius for a neighborhood search query ϵ, and the minimum number of neighbors N required for a core point.

Algorithm 2 DBSCAN clustering algorithm

```
 1: procedure DBSCAN(Eig,$\epsilon$,$N$)
 2:     from Eig choose an initial core point and initialize the first cluster label $k_m$ to 1
 3:     while there are unlabeled points  do
 4:         while new points can be labeled to the current cluster $k_m$ do
 5:             label the points that are at a distance $\epsilon$ from the current point
 6:             if  at least $N$ points are in a distance less than $\epsilon$ then
 7:                 label the current point as a core point
 8:             else
 9:                 label the current point as a noise point or a border point
10:             end if
11:             choose the current point as one of the found core points
12:         end while
13:         select next unlabeled point as the current point and increase the cluster label $k_m$ by one
14:     end while
15: end procedure
```

Noise as well as border points can be reassigned in this algorithm. Similar to the k-means algorithm, the Matlab implementation of the DBSCAN algorithm comes with a wide variety of possible settings that optimize the performance and results obtained.

Compared to k-means, DBSCAN has the advantage that the number of clusters is found by the algorithm. On the other hand, this requires reasonable values for the input parameters ϵ and N. Moreover, in some cases better results can be achieved by prepossessing the data set before launching the clustering algorithm. The impact of changing only one parameter (ϵ) slightly is shown in Fig. 2.10.

Along with the stated algorithms, the clustering algorithms OPTICS and mean-shift are also used in this dissertation. As they are later stated only as substitution to the default algorithm, they will not be reviewed here.

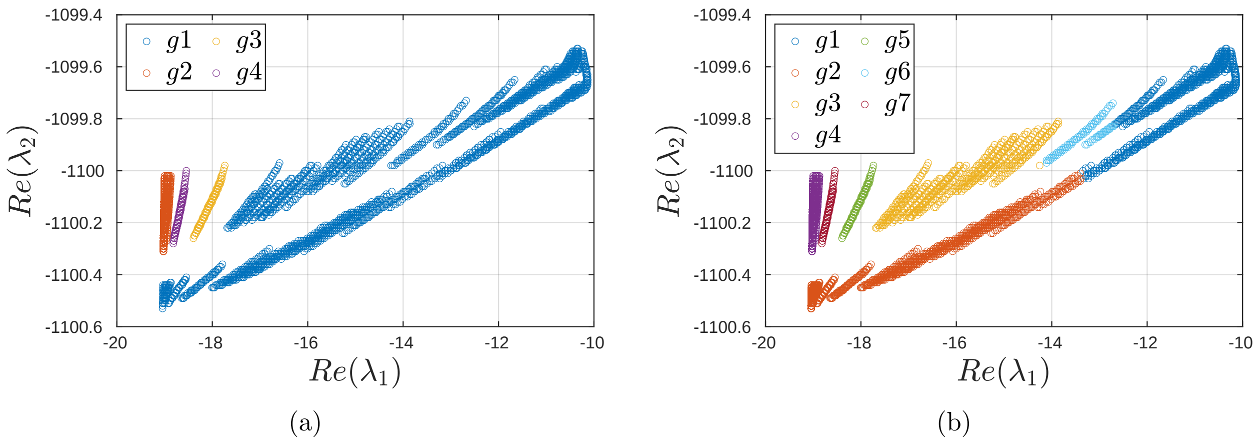


Fig. 2.10. Results of the DBSCAN algorithm with the data set $\boldsymbol{Eig}$ and $N = 1$. In (a) the clustering is performed with $\epsilon = 0.1$, while in (b) $\epsilon = 0.09$ is used.

2.5.3 Clustering Evaluation: The Silhouette Coefficient

To measure the quality of clustering, the silhouette coefficient can be used. The silhouette value for each point shows how similar this point is to the other points belonging to a cluster. The silhouette coefficient for a point $\boldsymbol{\lambda}_{pt}$ is defined as:

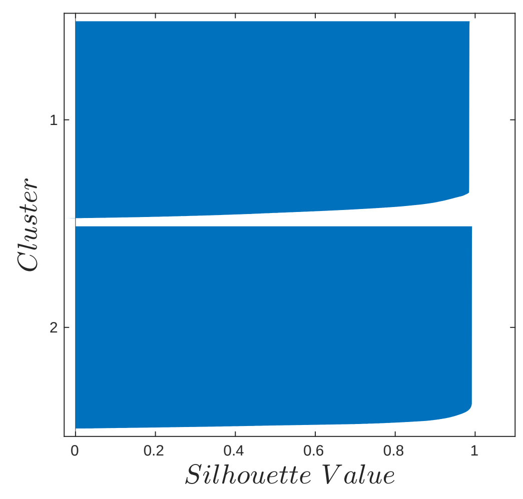
$$silhouette(\boldsymbol{\lambda}_{pt}) = \frac{b_{pt} - a_{pt}}{max(a_{pt}, b_{pt})} \tag{2.43}$$

Where a_{pt} represents the average distance from the point $\boldsymbol{\lambda}_{pt}$ to the remaining points of the same cluster, while b_{pt} represents the minimum average distance from the point $\boldsymbol{\lambda}_{pt}$ to the points belonging to a different cluster.

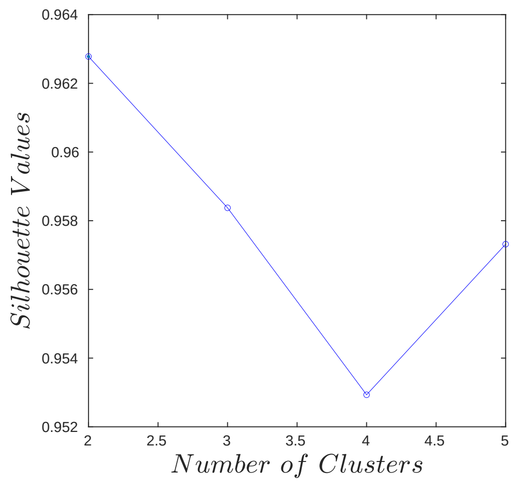
The silhouette coefficient can attain values between $[-1, 1]$. A close value to 1 indicates that the point is well clustered and thereby assigned to the appropriate cluster. If this value is close to zero, the point lies in between two clusters. A negative value indicates that the point is closer to other clusters than to its own. This indicates that the clustering performed can be improved.

Considering the data set $\boldsymbol{Eig}$ and the clustering performed in Section 2.5.1 with the cluster number $k_m = 2$, the silhouette coefficient for each of the 18500 points is shown in Fig. 2.11a. As observed, the points are well labeled in the two clusters.

As stated in Section 2.5.1, when using the k-means algorithm the optimal number of clusters k_m must be provided. For that, the silhouette coefficient can be used. In fact, Matlab provides a clustering evaluation function called *evalclusters*, which according to a set of specified criteria evaluates the cluster analysis. The result of this evaluation for the studied case from Section 2.5.1 is illustrated in Fig. 2.11b, with the silhouette coefficient set as the clustering evaluation criterion. As observed, the optimal number of clusters for this case is $k_m = 2$.



(a)



(b)

Fig. 2.11. In (a) the silhouette coefficients of the data points after performing k-means with $k_m = 2$ on $\boldsymbol{Eig}$ is shown, while in (b) the result of the *evalclusters* function with the silhouette as the evaluation criterion and a maximum of 5 clusters to test is illustrated.

3 Sampling the State Space

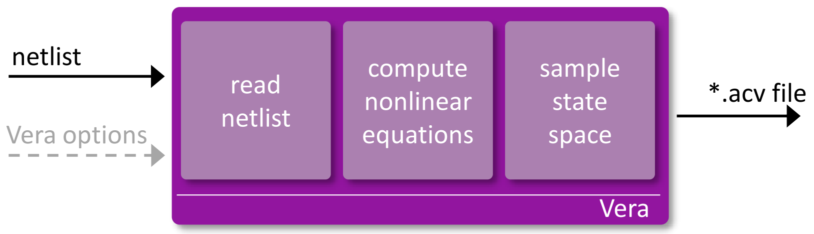


Fig. 3.1. Overview of *Vera* and its basic building blocks.

The first step towards the model abstraction, as illustrated in Fig. 1.4, is to sample the netlist via *Vera* [HKH04; HB5; SH10a]. Originally, *Vera* is an analog equivalence checking tool that formally verifies the equivalence of two circuits by comparing their *reduced* state space. This will be examined deeper in Chapter 6. For the purpose of this dissertation, *Vera* was partially extended to fit the needs for the model generation process described in Chapter 4.

In the following, *Vera* is used to sample only one netlist. Fig. 3.1 shows an overview of the sampling process as well as the building blocks of *Vera*. During the sampling process, an order reduction is performed on the circuit as well as reachability analysis on the sampled data points classifying them as reachable or not reachable. As shown in Fig. 3.1, the sampling yields an *acv* file. This file contains significant information such as the values of sampled data points, their reachability status, the connection between them given by a directed graph, and the eigenvalues and eigenvectors of the pointwise linearized system.

The sampling performed by *Vera* is done on the original netlist at transistor level with full BSIM accuracy. In order to generate an accurate model, not only the nodal currents and voltages need to be sampled, but also significant behavioral data, such as the eigenvalues and eigenvectors. Note that there is a nonlinear evolution of these sampled values in regard to the different sampled points, which is the reason why these values need to be determined at every sampled point.

Therefore, to keep the overall system behavior, it is not enough to sample along some transient trajectories. Rather, the circuit must be sampled in an adequate manner in the reachable state space. The sampling process of *Vera* is described in Algorithm 3. *Vera* samples a netlist with BSIM accuracy stepping thereby through the state space like in [SH10a], nonlinearly reducing the order as in [PAOS03], and examining the reachability of the sampled points [SH10a]. The result is a set of data points connected by a directed graph [SH12b]. In the following, the sampling process performed by *Vera* will be examined in detail. Moreover, the generated *acv* file obtained from this sampling, will be handled at the end of this chapter.

Algorithm 3 State space sampling using *Vera*

```
1: procedure State space sampling
2:     read the netlist
3:     set up the nonlinear differential algebraic system:
4:     $\boldsymbol{f}(\boldsymbol{x}(t), \dot{\boldsymbol{x}}(t), \boldsymbol{u}(t)) = \boldsymbol{0}$    $\boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{u}(t)) = \boldsymbol{y}(t)$
       } computing the nonlinear Eqs. (Section 3.1)
5:     for every input value in a predefined range do
6:         DC analysis $\rightarrow$ initial state vector $\boldsymbol{x}_{DC}$
7:         compute the eigenvector matrix $\boldsymbol{F}$
8:         for every sample point in predefined ranges do
9:             compute new step size $\Delta\boldsymbol{x}_s$
10:            compute state vectors for next sample point:
               $\boldsymbol{x}_{est} = \boldsymbol{x}_{old} + \boldsymbol{F}\Delta\boldsymbol{x}_{s,\Lambda}$
11:            compute consistent sample points:
               $\boldsymbol{x}_{cons}$ using $\boldsymbol{x}_{est}$
12:            compute the new eigenvector matrix $\boldsymbol{F}$
13:            $\boldsymbol{x}_{old} = \boldsymbol{x}_{cons}$
14:        end for
15:    end for
16: end procedure
       } (lines 5–16) state space sampling (Section 3.2)
```

3.1 Computing the Nonlinear Equations

Starting from the Spice netlist description of the circuit, the nonlinear differential algebraic equations are set up as stated in [SH10b]:

$$\boldsymbol{f}(\boldsymbol{x}(t), \dot{\boldsymbol{x}}(t), \boldsymbol{u}(t)) = \boldsymbol{0} \tag{3.1a}$$

$$\boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{u}(t)) = \boldsymbol{y}(t) \tag{3.1b}$$

Where $\boldsymbol{x}(t) \in \mathbb{R}^n$ represents the vector of n system variables $\{x_i(t) \mid i \in 1, \ldots, n\}$, $\boldsymbol{u}(t) \in \mathbb{R}^k$ represents the vector of k input variables $\{u_i(t) \mid i \in 1, \ldots, k\}$, and $\boldsymbol{y}(t) \in \mathbb{R}^p$ represents the vector of p output variables $\{y_i(t) \mid i \in 1, \ldots, p\}$. Note that in the following the indication that the variables $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{u}$ are functions of the time t will be dropped to shrink the size of the equations.

The system of equations $\boldsymbol{f}$ from Eq. (3.1a) is set up by using the modified nodal approach (MNA) [HRB75]. Application of the MNA results in an implicit equation for each circuit node with the system variables usually being the nodal voltages, some device currents, and additional variables resulting from device equations or behavioral description of parts from the analog circuit [SH10b]. Using the MNA, the charge-oriented equations can be set up:

$$\boldsymbol{N}\dot{\boldsymbol{q}} = \boldsymbol{f}_s(\boldsymbol{x}, \boldsymbol{u}) \tag{3.2}$$

$$\boldsymbol{q} = \boldsymbol{f}_a(\boldsymbol{x}) \tag{3.3}$$

Where $\boldsymbol{q}$ harbors the charge variables. For simplicity, inductive elements are not considered. If inductive elements were to be considered, an approach similar to the charge oriented MNA stated in [ABB11] could be used. Note that the vector of the unknowns would be extended by the flux variables $\boldsymbol{\phi}$ of the inductive elements.

A closer look at Eq. (3.2) reveals that the equation represents in general a system of ordinary differential equations (ODEs), for the case that $\boldsymbol{N}$ has full rank, while Eq. (3.3) represents a system of algebraic equation. Inserting Eq. (3.3) into Eq. (3.2) would again result in Eq. (3.1a), a system of differential algebraic equations (DAEs).

3.2 State Space Sampling

As described in Algorithm 3, the state space sampling involves several steps. These steps will be examined in detail in the following.

3.2.1 DC Analysis

After the system equations have been set up, a DC analysis is performed to find the operating point for a given input $\boldsymbol{u} = \boldsymbol{u}_{DC}$, as mentioned in Algorithm 3 at line 6. For that, the time derivative of the charge variables $\dot{\boldsymbol{q}}$ in Eq. (3.2) is set to $\boldsymbol{0}$, as at the steady state ($t \to \infty$) the system reaches an equilibrium. By that, Eq. (3.2) becomes:

$$\boldsymbol{f}_s(\boldsymbol{x}, \boldsymbol{u}) = \boldsymbol{0} \tag{3.4}$$

For a given input $\boldsymbol{u}_{DC}$, the only unknown in Eq. (3.4) is the state vector $\boldsymbol{x}$. Therefore, the task becomes finding $\boldsymbol{x}_{DC}$ such that:

$$\boldsymbol{f}_s(\boldsymbol{x}_{DC}, \boldsymbol{u}_{DC}) = \boldsymbol{0} \tag{3.5}$$

For simplicity, denote $\boldsymbol{f}_s(\boldsymbol{x}, \boldsymbol{u})$ as $\boldsymbol{f}_s(\boldsymbol{x})$ for the case that $\boldsymbol{u}$ is given. Eq. (3.5) can be solved using the explicit Newton-Raphson method. For each iteration, the Jacobian matrix is calculated:

$$\boldsymbol{J}_{\boldsymbol{f}_s}(\boldsymbol{x}_i) = \begin{bmatrix} \frac{\partial \boldsymbol{f}_{s_1}(\boldsymbol{x}_i)}{\partial x_1} & \frac{\partial \boldsymbol{f}_{s_1}(\boldsymbol{x}_i)}{\partial x_2} & \cdots & \frac{\partial \boldsymbol{f}_{s_1}(\boldsymbol{x}_i)}{\partial x_n} \\ \frac{\partial \boldsymbol{f}_{s_2}(\boldsymbol{x}_i)}{\partial x_1} & \frac{\partial \boldsymbol{f}_{s_2}(\boldsymbol{x}_i)}{\partial x_2} & \cdots & \frac{\partial \boldsymbol{f}_{s_2}(\boldsymbol{x}_i)}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \boldsymbol{f}_{s_n}(\boldsymbol{x}_i)}{\partial x_1} & \frac{\partial \boldsymbol{f}_{s_n}(\boldsymbol{x}_i)}{\partial x_2} & \cdots & \frac{\partial \boldsymbol{f}_{s_n}(\boldsymbol{x}_i)}{\partial x_n} \end{bmatrix} \tag{3.6}$$

With the Jacobian matrix at hand, its inverse can be computed and the solution of Eq. (3.4) can be determined by applying the Newton-Raphson method:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \boldsymbol{J}_{\boldsymbol{f}_s}^{-1}(\boldsymbol{x}_i)\boldsymbol{f}_s(\boldsymbol{x}_i) \tag{3.7}$$

After several iterations applying Eq. (3.7), the termination condition becomes valid and the DC solution $\boldsymbol{x}_{DC}$ for the input $\boldsymbol{u}_{DC}$ is found.

3.2.2 Computing the Linear Eigenvector Matrix

With the DC solution $\boldsymbol{x}_{DC}$ at hand, the eigenvector matrix $\boldsymbol{F}$ of the corresponding linearized system is computed, which is of significant importance for the sampling process. This computation is divided into several steps as shown in Algorithm 4. These steps are examined in the following.

Algorithm 4 Computing the eigenvector matrix $\boldsymbol{F}$

```
1: procedure Compute $\boldsymbol{F}$
2:     linearize Eq. (3.1a) and find the conduction matrix $\boldsymbol{A}$ and the capacitance matrix $\boldsymbol{E}$
3:     find the reduced state space $\mathcal{S}_\lambda$
4:     compute the eigenvector matrix $\boldsymbol{F}$
5: end procedure
```

From the Nonlinear DAE to the Linear System Description: Computing $\boldsymbol{A}$ and $\boldsymbol{E}$

As stated in Algorithm 4 at line 2, to find the conduction matrix $\boldsymbol{A}$ and the capacitance matrix $\boldsymbol{E}$, Eq. (3.1a) is linearized around the operating point $\boldsymbol{x}_{DC}$ for the input vector $\boldsymbol{u}_{DC}$. Note that the conduction matrix and the capacitance matrix are often denoted as $\boldsymbol{G}$ and $\boldsymbol{C}$, however this notation is not used here as it is more convenient to work with the stated notation, especially in Chapter 4.

As mentioned, Eq. (3.1a) can be divided into the two Eqs. (3.2, 3.3). So, the task becomes linearizing the equations:

$$\boldsymbol{N}\dot{\boldsymbol{q}} = \boldsymbol{f}_s(\boldsymbol{x}, \boldsymbol{u}) \qquad \Longrightarrow \qquad \boldsymbol{E}\Delta\dot{\boldsymbol{x}} + \boldsymbol{A}\Delta\boldsymbol{x} = \boldsymbol{B}\Delta\boldsymbol{u}$$

To determine the capacitance matrix $\boldsymbol{E} \in \mathbb{R}^{n\times n}$, the chain rule is first applied on Eq. (3.3) to determine $\dot{\boldsymbol{q}}$:

$$\begin{aligned} \dot{\boldsymbol{q}} &= \frac{d\boldsymbol{q}}{dt} \\ &= \frac{d\boldsymbol{f}_a(\boldsymbol{x})}{dt} \\ &= \frac{d\boldsymbol{f}_a(\boldsymbol{x})}{d\boldsymbol{x}}\frac{d\boldsymbol{x}}{dt} \\ &= \boldsymbol{J}_{\boldsymbol{f}_a}(\boldsymbol{x})\dot{\boldsymbol{x}} \end{aligned} \tag{3.8}$$

Where $\boldsymbol{J}_{\boldsymbol{f}_a}(\boldsymbol{x})$ is a Jacobian matrix and $\dot{\boldsymbol{x}}$ is the derivative of the state vector $\boldsymbol{x}$ with respect to the time. Substituting Eq. (3.8) in $\dot{\boldsymbol{q}}$ from Eq. (3.2), for $\boldsymbol{x} = \boldsymbol{x}_{DC}$ and $\boldsymbol{u} = \boldsymbol{u}_{DC}$, yields:

$$\boldsymbol{N}\dot{\boldsymbol{q}} = \underbrace{\boldsymbol{N}\boldsymbol{J}_{\boldsymbol{f}_a}(\boldsymbol{x}_{DC})}_{\boldsymbol{E}}\dot{\boldsymbol{x}} \tag{3.9}$$

The remaining nonlinear function $\boldsymbol{f}_s(\boldsymbol{x}, \boldsymbol{u})$ of Eq. (3.2) is next linearized around the operating point $(\boldsymbol{x}_{DC}, \boldsymbol{u}_{DC})$ using the first order Taylor polynomial for multivariable functions. Using the corresponding Taylor terms along with Eq. (3.5), $\boldsymbol{f}_s(\boldsymbol{x}, \boldsymbol{u})$ is linearized to:

$$\boldsymbol{f}_s(\boldsymbol{x}, \boldsymbol{u}) \approx \underbrace{\boldsymbol{f}_s(\boldsymbol{x}_{DC}, \boldsymbol{u}_{DC})}_{\boldsymbol{0}} + \underbrace{\frac{\partial \boldsymbol{f}_s(\boldsymbol{x}_{DC}, \boldsymbol{u}_{DC})}{\partial \boldsymbol{x}}}_{\boldsymbol{A}}(\boldsymbol{x} - \boldsymbol{x}_{DC}) + \underbrace{\frac{\partial \boldsymbol{f}_s(\boldsymbol{x}_{DC}, \boldsymbol{u}_{DC})}{\partial \boldsymbol{u}}}_{\boldsymbol{B}}(\boldsymbol{u} - \boldsymbol{u}_{DC}) \tag{3.10}$$

This is an approximation as the reminder of the Taylor series in Eq. (3.10) is missing. In fact, the error of this approximation can be estimated by determining this reminder.

The conduction and input matrices in Eq. (3.10), $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{B} \in \mathbb{R}^{n\times k}$, respectively, are both Jacobian matrices. With $\Delta\boldsymbol{u} = \boldsymbol{u} - \boldsymbol{u}_{DC}$ and $\Delta\boldsymbol{x} = \boldsymbol{x} - \boldsymbol{x}_{DC}$ leading to $\Delta\dot{\boldsymbol{x}} = \dot{\boldsymbol{x}}$ (as $\dot{\boldsymbol{x}}_{DC} = \boldsymbol{0}$), the linearized system becomes:

$$\boldsymbol{E}\Delta\dot{\boldsymbol{x}} = \boldsymbol{A}\Delta\boldsymbol{x} + \boldsymbol{B}\Delta\boldsymbol{u} \tag{3.11}$$

Note that Eq. (3.11) could have also been derived from Eq. (3.1a) by applying the Taylor series directly on $\boldsymbol{f}(\dot{\boldsymbol{x}}, \boldsymbol{x}, \boldsymbol{u})$ as stated in Section 2.3.3 by applying Eq. (2.37).

Regarding the output of the system from Eq. (3.1b), the approach presented in 2.3.3 can be used to linearize the output equation. In fact, this is not needed here as the output voltages and currents are usually functions of the nodal currents and voltages presented in $\boldsymbol{x}$. Thus, the output vector $\boldsymbol{y}$ is often directly given as a function of the state variables $\boldsymbol{x}$. For consistency, by additionally consider the input vector $\boldsymbol{u}$, the output equation becomes:

$$\Delta\boldsymbol{y} = \boldsymbol{C}\Delta\boldsymbol{x} + \boldsymbol{D}\Delta\boldsymbol{u} \tag{3.12}$$

With:

$$\boldsymbol{y}_{DC} = \boldsymbol{C}\boldsymbol{x}_{DC} + \boldsymbol{D}\boldsymbol{u}_{DC} \qquad \boldsymbol{C} = \frac{\partial \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{u})}{\partial \boldsymbol{x}} \qquad \boldsymbol{D} = \frac{\partial \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{u})}{\partial \boldsymbol{u}} \tag{3.13}$$

Where $\boldsymbol{C} \in \mathbb{R}^{p\times n}$ and $\boldsymbol{D} \in \mathbb{R}^{p\times k}$. For simplicity, the feedthrough matrix $\boldsymbol{D}$ will be neglected in what follows, as the state vector $\boldsymbol{x}$ usually contains all nodal voltages and currents.

Summing up the previous results, the linear descriptor representation of the systems is:

$$\boldsymbol{E}\Delta\dot{\boldsymbol{x}} = \boldsymbol{A}\Delta\boldsymbol{x} + \boldsymbol{B}\Delta\boldsymbol{u} \tag{3.14a}$$

$$\Delta\boldsymbol{y} = \boldsymbol{C}\Delta\boldsymbol{x} + \boldsymbol{D}\Delta\boldsymbol{u} \tag{3.14b}$$

According to Section 2.3.2, Eq. (3.14) represent the linear descriptor representation of the system with $\boldsymbol{x} \in \mathbb{R}^n$ being the state space vector, sometimes denoted as descriptor vector, in the original state space $\mathcal{S}_o$ of the system. As stated previously, this vector hosts all nodal current and voltages of the circuit.

The Reduced State Space $\mathcal{S}_\lambda$

At this point, we aim to reduce the size of Eq. (3.14) pursuing two purposes:

- get rid of the algebraic equations presented in Eq. (3.14a)
- perform a dominant pole order reduction on Eq. (3.14a) to further reduce the order of the equation to an appropriate one that considers the relevant dynamics of the system

The first goal can be achieved by separating Eq. (3.14a) into two equations according to the correspondence to the finite and infinite eigenvalues. This can be achieved by transforming Eq. (3.14a) into the KCF as stated in Section 2.2.2.

Consider the two transformation matrices $\boldsymbol{F} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{H} \in \mathbb{R}^{n\times n}$. $\boldsymbol{F}$ is the matrix of the right eigenvectors, as the columns of $\boldsymbol{F}$ in general consist of the right eigenvectors of the linearized system. We demand that $\boldsymbol{F}$ is regular, by that all its column vectors are independent. For the case the finite eigenvalues of the system are distinct, this is straight forward, as the finite eigenvectors are by definition independent, and the infinite eigenvectors can be freely chosen as stated in theorem 2.2.4. $\boldsymbol{H}$ on the other hand is a properly calculated matrix. There exist different methods to calculate this matrix [Mär91][Doo79]. For example, if all eigenvectors are independent and arranged column-wise in the $\boldsymbol{F}$ matrix, to satisfy Eq. (2.24), $\boldsymbol{H}$ can be calculated as:

$$\boldsymbol{H} = \tilde{\boldsymbol{A}}\boldsymbol{F}^{-1}\boldsymbol{A}^{-1} \tag{3.15}$$

Note that $\boldsymbol{F}$ and especially $\boldsymbol{H}$ are sometimes ill calculated, which challenges the model abstraction process described in Chapter 4. Even though several methods have been implemented in *Vera* that compute these matrices, errors can still arise due to numerical problems. As seen later, Section 4.5.2 tries to counter this problem.

Using the transformation matrices $\boldsymbol{F}$, the state space vector $\boldsymbol{x} \in \mathbb{R}^n$ in the original state space $\mathcal{S}_o$ is transformed to the state space vector $\boldsymbol{x}_s \in \mathbb{R}^n$ in the new state space $\mathcal{S}_s$ by a linear transformation (see Section 2.3.1):

$$\Delta \boldsymbol{x} = \boldsymbol{F} \Delta \boldsymbol{x}_s \tag{3.16}$$

Using Eq. (3.16) and the transformation matrix $\boldsymbol{H}$, Eq. (3.14) is transformed into the KCF as stated in Section 2.2.2:

$$\underbrace{\boldsymbol{HEF}}_{\tilde{\boldsymbol{E}}} \Delta \dot{\boldsymbol{x}}_s = \underbrace{\boldsymbol{HAF}}_{\tilde{\boldsymbol{A}}} \Delta \boldsymbol{x}_s + \underbrace{\boldsymbol{HB}}_{\tilde{\boldsymbol{B}}} \Delta \boldsymbol{u} \tag{3.17a}$$

$$\Delta \boldsymbol{y} = \underbrace{\boldsymbol{CF}}_{\tilde{\boldsymbol{C}}} \Delta \boldsymbol{x}_s \tag{3.17b}$$

Note that transformed matrices are marked by a tilde (˜). Expanding $\tilde{\boldsymbol{E}}$ and $\tilde{\boldsymbol{A}}$ yields the form described in Eq. (2.24). By that, Eq. (3.17) becomes:

$$\begin{bmatrix} \boldsymbol{I}_\Lambda & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{N} \end{bmatrix} \Delta \dot{\boldsymbol{x}}_s = \begin{bmatrix} \boldsymbol{J} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I}_\infty \end{bmatrix} \Delta \boldsymbol{x}_s + \begin{bmatrix} \tilde{\boldsymbol{B}}_\Lambda \\ \tilde{\boldsymbol{B}}_\infty \end{bmatrix} \Delta \boldsymbol{u} \tag{3.18a}$$

$$\Delta \boldsymbol{y} = \begin{bmatrix} \tilde{\boldsymbol{C}}_\Lambda & \tilde{\boldsymbol{C}}_\infty \end{bmatrix} \Delta \boldsymbol{x}_s \tag{3.18b}$$

With $\boldsymbol{I}_\Lambda, \boldsymbol{J} \in \mathbb{R}^{r \times r}$, $\boldsymbol{I}_\infty, \boldsymbol{N} \in \mathbb{R}^{(n-r)\times(n-r)}$, $\tilde{\boldsymbol{B}}_\Lambda \in \mathbb{R}^{r \times k}$, $\tilde{\boldsymbol{B}}_\infty \in \mathbb{R}^{(n-r)\times k}$, $\tilde{\boldsymbol{C}}_\Lambda \in \mathbb{R}^{p \times r}$, and $\tilde{\boldsymbol{C}}_\infty \in \mathbb{R}^{p \times (n-r)}$. As stated in [HKH04], this transformation is only valid at the particular sampled point of the linearized system. Note that the initial system consisting of n variables can now be divided into r dynamic variables and $(n-r)$ algebraic variables.

In order to achieve the second goal stated at the beginning of this section i.e. extracting the relevant dynamics of the system, Eq. (3.18) is further processed. Performing a dominant pole reduction similar to [PAOS03] on Eq. (3.18), reduces the high order resulting from parasitic poles down to the functionality needed. This reduction can be specified by the number of poles of interest to be incorporated or by a corner frequency. All poles and zeros which are below the corner frequency are incorporated in the model. This can be thought of as moving the remaining i poles and zeroes, that are far to the left of the complex s-plane and not in the range of interest, to infinity. By that, the size of $\boldsymbol{N}$ and $\boldsymbol{I}_\infty$ is increased by i rows and columns, while the size of $\boldsymbol{I}_\Lambda$ and $\boldsymbol{J}$ is decreased. The system thereby is left with:

$$m = (r - i), \tag{3.19}$$

finite eigenvalues. Of course, other reduction method could also be used, for example those stated in [Ban14]. As the model order reduction of a system is out of the scope of this dissertation, only the stated case will be adapted. The model order reduction results in:

$$\begin{bmatrix} \boldsymbol{I}_{\Lambda,red} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{N}_{red} \end{bmatrix} \Delta \dot{\boldsymbol{x}}_s = \begin{bmatrix} \boldsymbol{J}_{red} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I}_{\infty,red} \end{bmatrix} \Delta \boldsymbol{x}_s + \begin{bmatrix} \tilde{\boldsymbol{B}}_{\Lambda,red} \\ \tilde{\boldsymbol{B}}_{\infty,red} \end{bmatrix} \Delta \boldsymbol{u} \tag{3.20a}$$

$$\Delta \boldsymbol{y} = \begin{bmatrix} \tilde{\boldsymbol{C}}_{\Lambda,red} & \tilde{\boldsymbol{C}}_{\infty,red} \end{bmatrix} \Delta \boldsymbol{x}_s \tag{3.20b}$$

Where the subindex *red* stands for the matrices after model order reduction. With $\boldsymbol{I}_{\Lambda,red}, \boldsymbol{J}_{red} \in \mathbb{R}^{m\times m}$, $\boldsymbol{I}_{\infty,red}, \boldsymbol{N}_{red} \in \mathbb{R}^{(n-m)\times(n-m)}$, $\tilde{\boldsymbol{B}}_{\Lambda,red} \in \mathbb{R}^{m\times k}$, $\tilde{\boldsymbol{B}}_{\infty,red} \in \mathbb{R}^{(n-m)\times k}$, $\tilde{\boldsymbol{C}}_{\Lambda,red} \in \mathbb{R}^{p\times m}$, and $\tilde{\boldsymbol{C}}_{\infty,red} \in \mathbb{R}^{p\times(n-m)}$. For simplicity, we perform the following assumptions

Assumption 1. *All finite eigenvalues of the system are distinct, the matrix $\boldsymbol{J}_{red}$ in JCF becomes thus a diagonal matrix $\boldsymbol{\Lambda}$.*

Assumption 2. *The index of nilpotency $\eta \leq 1$*

Assumption 1 is quite common in numerical applications, but is here only assumed for simplicity. Assumption 2 is made concerning the nilpotent matrix $\boldsymbol{N}_{red}$. Theorem 3.2.1 states the results from [Est00] summarized in [ABB11]. Note that the index of the DAE is the index of nilpotency as stated in Section 2.2.2.

Theorem 3.2.1. *If the differential index of the DAE is one, then the network contains neither cut-sets with only inductive and/or independent current sources, nor loops with capacitive elements and independent voltage sources.*

In what follows, these assumptions are made as stated in [HKH04]. Concerning Assumption 1, $\boldsymbol{\Lambda}$ can be replaced at any time with little adjustment by $\boldsymbol{J}_{red}$. For Assumption 2, the case $\eta > 1$ is handled in the Appendix A.1.

For the stated assumptions, the system becomes:

$$\begin{bmatrix} \boldsymbol{I}_{\Lambda,red} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \Delta\dot{\boldsymbol{x}}_s = \begin{bmatrix} \boldsymbol{\Lambda} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I}_{\infty,red} \end{bmatrix} \Delta\boldsymbol{x}_s + \begin{bmatrix} \tilde{\boldsymbol{B}}_{\Lambda,red} \\ \tilde{\boldsymbol{B}}_{\infty,red} \end{bmatrix} \Delta\boldsymbol{u} \tag{3.21a}$$

$$\Delta\boldsymbol{y} = \begin{bmatrix} \tilde{\boldsymbol{C}}_{\Lambda,red} & \tilde{\boldsymbol{C}}_{\infty,red} \end{bmatrix} \Delta\boldsymbol{x}_s \tag{3.21b}$$

Eq. (3.21a) can be divided into two parts: a dynamic and a static part. This is done by dividing the state space vector $\boldsymbol{x}_s \in \mathbb{R}^n$ into two parts: $\boldsymbol{x}_\lambda \in \mathbb{R}^m$ and $\boldsymbol{x}_\infty \in \mathbb{R}^{n-m}$.

$$\boldsymbol{x}_s = \begin{bmatrix} \boldsymbol{x}_\lambda \\ \boldsymbol{x}_\infty \end{bmatrix} \tag{3.22}$$

The vector $\boldsymbol{x}_\lambda$ is referred to as the state vector in the reduced canonical state space $\mathcal{S}_\lambda$. The m-finite eigenvalues of the system are associated with this state vector. The $(n-m)$-infinite eigenvalues are associated with $\boldsymbol{x}_\infty$, the state vector in the state space $\mathcal{S}_\infty$.The parasitic behavior of these poles is not totally rejected, as the nonlinear large signal dynamics are part of the consistent solutions as explained at the end of Section 3.2.4.

By replacing the subscript (Λ, red) by λ and splitting $\boldsymbol{x}_s$ into $\boldsymbol{x}_\lambda$ and $\boldsymbol{x}_\infty$ according to Eq. (3.22), Eq. (3.21) becomes:

$$\begin{bmatrix} \boldsymbol{I}_\lambda & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \Delta\dot{\boldsymbol{x}}_\lambda \\ \Delta\dot{\boldsymbol{x}}_\infty \end{bmatrix} = \begin{bmatrix} \boldsymbol{\Lambda} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I}_{\infty,red} \end{bmatrix} \begin{bmatrix} \Delta\boldsymbol{x}_\lambda \\ \Delta\boldsymbol{x}_\infty \end{bmatrix} + \begin{bmatrix} \tilde{\boldsymbol{B}}_\lambda \\ \tilde{\boldsymbol{B}}_{\infty,red} \end{bmatrix} \Delta\boldsymbol{u} \tag{3.23a}$$

$$\Delta\boldsymbol{y} = \begin{bmatrix} \tilde{\boldsymbol{C}}_\lambda & \tilde{\boldsymbol{C}}_{\infty,red} \end{bmatrix} \begin{bmatrix} \Delta\boldsymbol{x}_\lambda \\ \Delta\boldsymbol{x}_\infty \end{bmatrix} \tag{3.23b}$$

Hence, the dynamic or differential part of Eq. (3.23a) is:

$$\boldsymbol{I}_\lambda \Delta\dot{\boldsymbol{x}}_\lambda = \boldsymbol{\Lambda}\Delta\boldsymbol{x}_\lambda + \tilde{\boldsymbol{B}}_\lambda \Delta\boldsymbol{u}, \tag{3.24}$$

while the static or algebraic part is:

$$\boldsymbol{I}_{\infty,red}\Delta\boldsymbol{x}_{\infty} = -\tilde{\boldsymbol{B}}_{\infty,red}\Delta\boldsymbol{u} \tag{3.25}$$

In a similar fashion to the state vector $\boldsymbol{x}_s$, the eigenvector matrix $\boldsymbol{F}$ as well as the proper calculated matrix $\boldsymbol{H}$ can be divided into two parts:

$$\boldsymbol{F} = \begin{bmatrix}\boldsymbol{F}_{\lambda}\boldsymbol{F}_{\infty}\end{bmatrix} \qquad \boldsymbol{H} = \begin{bmatrix}\boldsymbol{H}_{\lambda} \\ \boldsymbol{H}_{\infty}\end{bmatrix} \tag{3.26}$$

Note that this distinction is performed after the model order reduction. Thus, $\boldsymbol{F}_{\Lambda} \in \mathbb{R}^{n\times m}$, $\boldsymbol{F}_{\infty} \in \mathbb{R}^{n\times(n-m)}$, $\boldsymbol{H}_{\Lambda} \in \mathbb{R}^{m\times n}$, and $\boldsymbol{H}_{\infty} \in \mathbb{R}^{(n-m)\times n}$. This concludes the calculation of the matrices $\boldsymbol{F}$ and $\boldsymbol{H}$ as stated in Algorithm 4.

3.2.3 State Space Step

Continuing the sampling process described in Algorithm 3, the next step is to calculate the state space step (line 9) followed by the calculation of the calculation of the estimated sample point $\boldsymbol{x}_{est}$ in the $\mathcal{S}_o$ domain (line 10). Previously, we have linearized Eq. (3.1a) around the operating point $(\boldsymbol{x}_{DC}, \boldsymbol{u}_{DC})$ and calculated the eigenvalues of the linearized system. With the transformation matrices $\boldsymbol{F}$ and $\boldsymbol{H}$, the system was transformed from the original state space $\mathcal{S}_o$ to the reduced state space $\mathcal{S}_{\lambda}$. The S_{∞} space is of minor importance as will be clarified in the following. The relationship of the state vectors of these state spaces is given by Eq. (3.16):

$$\Delta\boldsymbol{x} = \boldsymbol{F}_{\lambda}\Delta\boldsymbol{x}_{\lambda} + \boldsymbol{F}_{\infty}\Delta\boldsymbol{x}_{\infty} \tag{3.27}$$

By considering Eq. (3.25), it becomes clear that as long as $\Delta\boldsymbol{u} = \boldsymbol{0}$, $\Delta\boldsymbol{x}_{\infty} = \boldsymbol{0}$. Hence, for a given state step value $\Delta\boldsymbol{x}_{\lambda}$, a point in the $\mathcal{S}_o$ space can be estimated. For this, only the reduced part (with the subscript λ) from the right side of Eq. (3.27) needs to be considered:

$$\boldsymbol{x}_{est} = \boldsymbol{F}_{\lambda}\Delta\boldsymbol{x}_{\lambda} + \boldsymbol{x}_{old} \tag{3.28}$$

By solving Eq. (3.28), the estimated sample point $\boldsymbol{x}_{est}$ in the $\mathcal{S}_o$ domain is calculated. In the first step, $\boldsymbol{x}_{old}$ represents the DC operating point $\boldsymbol{x}_{DC}$.

This estimation however, suffers from a linearization error (see Section 3.2.2) which would propagate during the sampling process inducing even larger errors. To obtain accurate results, the calculated sample point must be corrected by computing the consistent solution.

3.2.4 The Consistent Solution

According to Algorithm 3, the next step in the sampling process is to calculate the consistent solution. The consistent circuit solution $\mathbf{x}_{cons}$ for each sampled point is calculated by solving the nonlinear circuit equations from Eq. (3.1) based on the estimate sample point from Eq. (3.28). This is done using a modified Spice circuit simulator [Dav03]. Each capacitor (netlist capacitor or parasitic capacitor) is virtually converted into a voltage source. The values of these voltage sources are the estimated voltage differences across the capacitor from $\mathbf{x}_{est}$. The solution of solving this modified equation system is the consistent solution $\mathbf{x}_{cons}$ with slightly differing capacitance voltages due to nonlinearities and capacitance loops but fully fulfilling the nonlinear DAE system Eq. (3.1).

Keeping in mind that Eq. (3.23) is linearized around the operating point ($\boldsymbol{x}_{DC}$, $\boldsymbol{u}_{DC}$), it becomes clear that this equation is only valid around a small range around this operating point. Hence, a small deviation, such as a state space step, from the operating point might lead to a change in the eigenvalues of the system and thereby to a change in the calculated matrices $\boldsymbol{F}$ and $\boldsymbol{H}$. Therefore, as stated at line 12 of Algorithm 3 and described in Algorithm 4, after the consistent solution is found, the system is linearized again around this point, the eigenvalues and transformation matrices are recalculated, and the system is again brought to the canonical form shown in Eq. (3.23). Hence, the system is linearized at every sampled point.

Since there exist unreachable regions in the state spaces, *Vera* incorporate a reachability analysis for the calculated sample points [SH10a]. For this, a directed graph is generated containing successor and predecessor relationships. Starting from the operating points, a reachability analysis is performed inside *Vera* and the reachable status of each sampled point is calculated.

3.3 Summary

The previous stated procedure from Section 3.2 is repeated for a given range of the state space and input voltages. That is, the system is first linearized around a DC point, an order reduction is performed, the transformation matrices are calculated, the estimated solution is computed based on a state step, and finally the consistent solution is calculated. For the next sampled point, the transformation matrices are recalculated, a new estimated point is calculated, and the consistent solution is determined using this point. This is repeated till the specified borders of the state space are reached, then a new DC point is selected, and the process is repeated.

All obtained information are stored in an *acv* file containing:

(a) all the $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{x}_\lambda \in \mathbb{R}^m$, and $\boldsymbol{x}_{virt} \in \mathbb{R}^m$ sampled values from the $\mathcal{S}_o$, $\mathcal{S}_\lambda$, and $\mathcal{S}_{virt}$ state spaces, respectively

(b) the transformation matrices $\boldsymbol{F} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{H} \in \mathbb{R}^{n\times n}$ that link both domains $\mathcal{S}_o$ and $\mathcal{S}_\lambda$

(c) the eigenvalues of the linearized system after model order reduction $\boldsymbol{\Lambda} \in \mathbb{R}^{m\times m}$, which are stored in the matrix $\boldsymbol{Eig}$ as described later in Section 4.4.1

(d) the directed edges between the sampled points

(e) the input matrices $\boldsymbol{B} \in \mathbb{R}^{n\times k}$ for the k input variables $\{u_i(t) : i \in 1, \ldots, k\}$, and the output matrix $\boldsymbol{C} \in \mathbb{R}^{p\times n}$ for the p output variables $\{y_i(t) \mid i \in 1, \ldots, p\}$

As stated, *Vera* performs a dominant pole order reduction. This is done by specifying the desired order of the reduced state space or by specifying the range of the frequencies of interest. By that, the tool identifies the nodes that harbor the state variables. These variables make up a subspace of the $\mathcal{S}_o$ space and are referred to as the *virt* variables. The state space vector corresponding to these variables is denoted as $\boldsymbol{x}_{virt} \in \mathbb{R}^m$ in the state space $\mathcal{S}_{virt} \subset \mathcal{S}_o$. An example of the $\mathcal{S}_{virt}$ space is given in Section 4.1.

Consequently, four significant state spaces can be identified:

1. $\mathcal{S}_o$: represents the original state space of the circuit. The state vector is $\boldsymbol{x} \in \mathbb{R}^n$, which contains all n nodal voltages and currents of the circuit

2. $\mathcal{S}_s$: represents the transformed state space of $\boldsymbol{x}$ as given by Eq. (3.16), the state vector is $\boldsymbol{x}_s \in \mathbb{R}^n$
3. $\mathcal{S}_\lambda$: represents the reduced state space spanned by the state vector $\boldsymbol{x}_\lambda \in \mathbb{R}^m$, such that the relationship between this vector to the state vector $\boldsymbol{x}_s$ is given by Eq. (3.22). Thus, $\mathcal{S}_\lambda \subset \mathcal{S}_s$
4. $\mathcal{S}_{virt}$: represents a state space of special interest, such that $\mathcal{S}_{virt}$ is a subspace of $\mathcal{S}_o$ i.e. $\mathcal{S}_{virt} \subset \mathcal{S}_o$. $\mathcal{S}_{virt}$ is referred to as the virtual state space. The state variables in the state vector $\boldsymbol{x}_{virt} \in \mathbb{R}^m$ represent the voltages across capacitors and the currents through inductors resulting in the states of the system. Note that the dimension of this state vector is, similarly to $\boldsymbol{x}_\lambda$, is defined by the model order reduction which results in $m < n$ states

The options, including for example the frequency range or order of the reduced system, are provided to *Vera* via a *.msl* file. This file hosts as well the method used for the calculation of the transformation matrices $\boldsymbol{F}$ and $\boldsymbol{H}$, the maximal allowed slew rate, the range of interest in the reduced state space, the state space step (calculate or fixed), the range of the inputs, the inputs step, the reachability method, and the output node of the system. Other options specify as well how the sampling in the $\mathcal{S}_\lambda$ space is performed, which in turn influences the overlapping of the points in this space.

On the other, in case only one circuit is provided to *Vera*, *Vera* samples the netlist as described in this chapter. In case two circuits are provided, *Vera* performs a formal equivalence checking in the analog domain. This is done by extending the sampling procedure presented in Algorithm 3 for two netlist, A and B. The process executes this time lines 2 till 12 for both circuits simultaneously. Between lines 12 and 13, two additional steps are executed; the derivative error $\delta_{\dot{x}}$ as well as the output error δ_y are computed:

$$\delta_{\dot{x}} = max(\dot{\boldsymbol{x}}_{cons,A} - \dot{\boldsymbol{x}}_{cons,B}) \qquad \delta_y = max(\boldsymbol{y}_{cons,A} - \boldsymbol{y}_{cons,B}) \tag{3.29}$$

Thus, after computing the consistent solutions for both circuits A and B and the corresponding output voltages, both errors are calculated by finding the maximum values over all corresponding dimensions. After the analysis completes, the relative derivative error $\boldsymbol{\delta}_{\dot{x},r}$ and the relative output error $\boldsymbol{\delta}_{y,r}$ are computed by considering the maximum voltage spectrum of the signals over all sampled points:

$$\delta_{\dot{x},r} = \frac{2 \cdot |\delta_{\dot{x}}| \cdot 100}{max(|\dot{\boldsymbol{x}}_{cons,A}|) + max(|\dot{\boldsymbol{x}}_{cons,B}|)} \qquad \delta_{y,r} = \frac{2 \cdot |\delta_y| \cdot 100}{max(|\boldsymbol{y}_{cons,A}|) + max(|\boldsymbol{y}_{cons,B}|)} \tag{3.30}$$

Finally, the maximum values of errors $\delta_{\dot{x}}$ and δ_y, as well as maximum values of relative errors $\delta_{\dot{x},r}$ and $\boldsymbol{\delta}_{y,r}$ are computed over all sampled points.

This verification, which calculates of the deviations between two models, has a significant importance and is therefore demonstrated later in Section 6.1. Using this methodology, the generated abstract model can be compared against the original Spice netlist. This closes the modeling and verification loop by verifying the correctness of the generated model. Consequently, error margins can be defined.

3.4 From Vera to Elsa: spaceM

The generated *acv* file from *Vera* needs to be imported to Matlab, as *Elsa* (Chapter 4) is written in Matlab syntax. This is done via *spaceM*, a MEX code that calls a C++ parser that uses LEX and

YACC to generate a Matlab structure named *space*. The contents of this structure are, as stated in Section 3.3, the contents of the *acv* file.

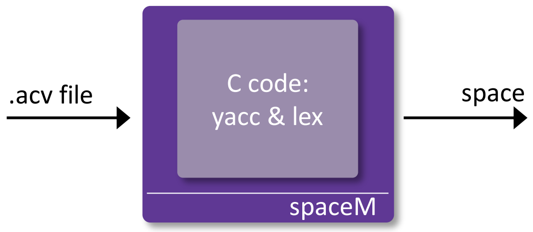


Fig. 3.2. Overview of *spaceM*.

The main function of *spaceM* is stated in appendix B. Note that at line 55 the C++ parser is called via the function *readACVFile*.

4 Elsa: Eigenvalues Based Hybrid Linear System Abstraction

In this chapter, the automated abstraction approach is examined in detail. As the eigenvalues play a significant role in the abstraction process, the approach has been named *Elsa*: **e**igenvalue-based hybrid **l**inear **s**ystem **a**bstraction. The abstraction process performed by *Elsa* can be divided into four building blocks as indicated in Fig. 4.1. In the following, each of these blocks as well, as the resultant hybrid automaton (HA), will be analyzed in detail.

After the initialization of *Elsa* in the first block, the locations of the HA are identified in the second block. For each of the identified locations, a system description is found. The result is a HA with a linear behavior in each of its location. On top to the system modeling, the guards as well as the invariant for each location are identified in the system modeling block. Finally, the model is generated according to the specified output language and the corresponding modeling methodology. The abstract model can be generated as a Verilog-A, Matlab (*Cora*), or SystemC-AMS behavioral model.

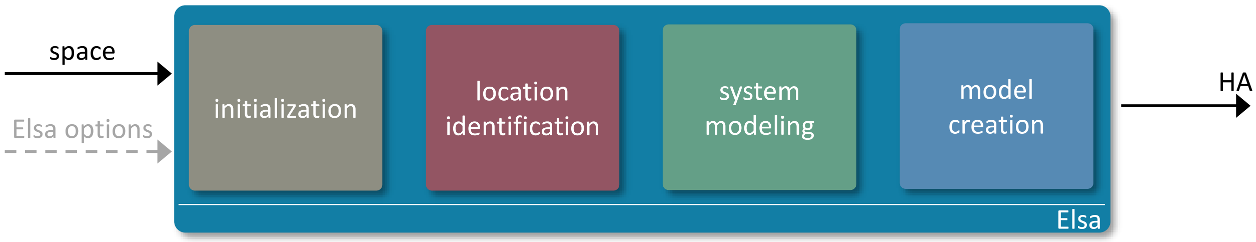


Fig. 4.1. Overview of *Elsa* and its 4 building blocks.

As indicated in Fig. 4.1, along with the Matlab structure *space* (see Section 3.4), an option file is passed to *Elsa*. This option file is called *SpaceOptions*. Hence, the start point of the abstraction process is the in Matlab loaded structure *space* and the option file *SpaceOptions* located in the current directory, while the end point is a generated HA in the specified output language. Note that the whole abstraction process starting with the sampling executed by *Vera*, till the creation of the HA by *Elsa* is automated. However, if a circuit has already been sampled and an *acv* file is at hand (section 3.3), *Elsa* can be launched separately.

In the following, the building blocks of *Elsa* shown in Fig. 4.1 are examined along with the possible modifications that can be set to improve the obtained HAs.

4.1 Running Example

To demonstrate the approach, a running example will be handled throughout this chapter. For that purpose, consider the schematic diagram of a second order lowpass filter shown in Fig. 4.2.

The lowpass filter exhibits a nonlinear limiting behavior at the output voltage V_{nout} as soon as this voltage reaches its maximum or minimum value given by V_{dd} or V_{ss}, respectively. V_{dd} is set to $+1.65$ V, while V_{ss} is set to -1.65 V. All voltages are given with respect to the reference voltage $V_{ref} = V_{gnd} = 0$ V. By that, V_{nout} is limited to the range $[-1.65, 1.65]$ V.

The operation amplifier has a gain of -0.8869, which is negative as the operation amplifier is used in an inverting configuration. Thus, for $V_{in} > 1.8604$ V the operation amplifier goes into the negative saturation $V_{nout} = -1.65$ V, while for $V_{in} < -1.8604$ V the operation amplifier goes into the positive saturation $V_{nout} = 1.65$ V.

In order to achieve this gain, the resistors R1, R2, and R3 are chosen to be 9.5 kΩ, 10 kΩ, and 10 kΩ, while the capacitors C1 and C2 are set to 0.01 μF and 0.1 μF, respectively. The operational amplifier is described in Spice at transistor level with full BSIM4 accuracy. Note that in Appendix C, the schematic of the netlist describing the operation amplifier is shown in Fig. C.1, while the test bench containing this operation amplifier is described in Listing C.1 and shown in Fig. 4.2.

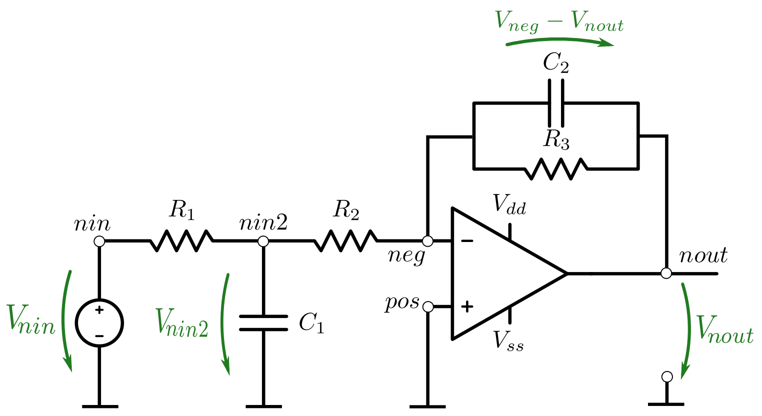


Fig. 4.2. A second order lowpass filter with a nonlinear limitation for $V_{nout} \in [-1.65, 1.65]$. The operational amplifier is a Spice file consisting of 17 transistors in a 350 nm CMOS technology.

For the running example from Fig. 4.2, *Vera* has sampled the netlist and generated an *acv* file as stated in Chapter 3. Note that the DAE system has $n = 24$ variables, which represent the nodal voltages and currents of the circuit from Fig. 4.2 including the internal ones from the operation amplifier (see Appendix C). With the nodal voltages (V_*) and currents ($I_{*.br}$), the $\boldsymbol{x} \in \mathbb{R}^{24}$ state vector is:

$$\boldsymbol{x} = \begin{bmatrix} V_{xI2.net1} & V_{xI2.net2} & V_{xI2.net3} & V_{xI2.vbias4} & V_{xI2.net7} & V_{xI2.net5} \\ V_{xI2.net6} & V_{xI2.net4} & V_{xI2.vbias1} & V_{xI2.vbias2} & I_{vpsub.br} & I_{vvdd.br} \\ I_{vvref.br} & I_{vvdd.br} & I_{V1.br} & V_{psubE} & V_{bbias} & V_{vss} \\ V_{vref} & V_{vdd} & V_{nout} & V_{neg} & V_{nin2} & V_{nin} \end{bmatrix}^T \tag{4.1}$$

For this example, *Vera* detects that the system has an order of $r = 14$. The result of this dynamic state identification is presented in Table 4.1.

An order reduction is performed by *Vera* resulting in a reduced order of $m = 2$. This corresponds to selecting the first two rows from Table 4.1. Hence, at each sample point, $\boldsymbol{\Lambda}$ from Eq. (3.23)

Table 4.1: Result of the dynamic state identification performed by *Vera*

#	Node 1	Node 2	*Eigenvalues* (λ_i)
1	$nout$	neg	-1.01×10^1
2	$nin2$	not set	-1.09×10^3
3	$xI2.vbias1$	$xI2.vbias2$	-1.71×10^7
4	$xI2.vbias4$	$bbias$	-3.24×10^7
5	$xI2.vbias1$	vdd	-4.36×10^7
6	$xI2.vbias4$	vss	-5.84×10^7
7	$xI2.net1$	neg	$(-2.77 - 17.02j) \times 10^8$
8	$xI2.net1$	not set	$(-2.77 + 17.02j) \times 10^8$
9	$xI2.net7$	$bbias$	-2.88×10^8
10	$xI2.net3$	$xI2.net6$	-4.67×10^8
11	$xI2.net4$	vdd	-1.36×10^9
12	$xI2.net2$	vss	-3.39×10^9
13	$xI2.net3$	vss	$(-8.21 - 2.62j) \times 10^9$
14	$xI2.net2$	$xI2.net6$	$(-8.21 + 2.62j) \times 10^9$

contains only two eigenvalues and is thereby a 2×2 matrix. Moreover, this defines the size of the reduced state space vector $\boldsymbol{x}_\lambda \in \mathbb{R}^2$ as well as the size of the virtual state space vector $\boldsymbol{x}_{virt} \in \mathbb{R}^2$.

Considering again the first two rows of Table 4.1, the dimensions of the $\boldsymbol{x}_{virt}$ vector can be identified. In this case, this vector consists of three entities from the $\boldsymbol{x}$ vector: V_{nout}, V_{neg}, and V_{nin2}. More precisely, the $\boldsymbol{x}_{virt}$ vector is:

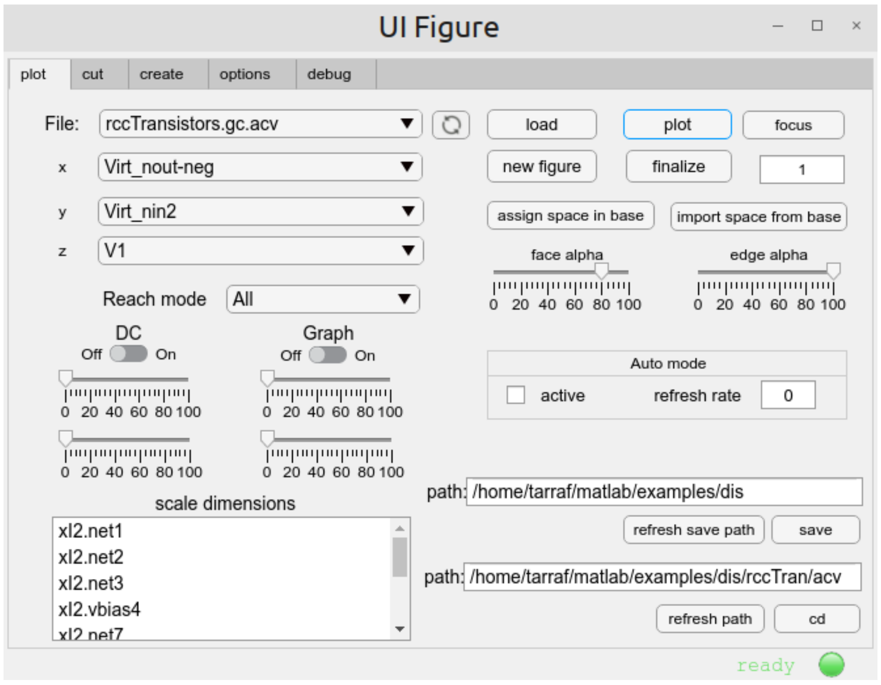
$$\boldsymbol{x}_{virt} = \begin{bmatrix} V_{nout} - V_{neg} \\ V_{nin2} \end{bmatrix} \tag{4.2}$$

Moreover, the dimensions of the transformation matrices are $\boldsymbol{F}_\Lambda \in \mathbb{R}^{24\times2}$, $\boldsymbol{F}_\infty \in \mathbb{R}^{24\times22}$, $\boldsymbol{H}_\Lambda \in \mathbb{R}^{2\times24}$, and $\boldsymbol{H}_\infty \in \mathbb{R}^{22\times24}$. The circuit is a SISO system with an output matrix $\boldsymbol{C} \in \mathbb{R}^{1\times24}$ and an input matrix $\boldsymbol{B} \in \mathbb{R}^{24\times1}$. Note that even though $\boldsymbol{C}$ and $\boldsymbol{B}$ are both vectors, they will be handled in the following as matrices to illustrate the approach in general.

Considering the sampling performed by *Vera*, the input range was specified as $V_{nin} \in [-5, 5]$ V with an input step of 0.5 V. The state space range was specified as $\boldsymbol{x}_{\lambda,1}, \boldsymbol{x}_{\lambda,2} \in [-5, 5]$ with a state space step of 0.25. As mentioned, the reduction order was specified as $m = 2$. For these options, *Vera* sampled this example with roughly $l = 18500$ points.

4.2 Examining the Results of Vera With Amcvis

Before starting with *Elsa*, a powerful tool will be briefly introduced. A Matlab application named *Amcvis* has been implemented that represents a graphical interface to *Elsa*. Moreover, *Amcvis* can be used for various debugging options, like for example plotting the sampled points in any of the specified state spaces from Section 3.3. Up to three dimensions can be plotted simultaneously. Additional dimensions affect the color scaling. In what follows, *Elsa* will be examined on code level and not through this interface. But before generating an abstract model with *Elsa*, it makes sense to analyze the data first with this tool. Fig. 4.4 presents the front end of *Amcvis*.



File: rccTransistors.gc.acv
x Virt_nout-neg
y Virt_nin2
z V1
Reach mode All
1
refresh rate 0
path: /home/tarraf/matlab/examples/dis
path: /home/tarraf/matlab/examples/dis/rccTran/acv
xl2.net1
xl2.net2
xl2.net3
xl2.vbias4
xl2.net7

Fig. 4.3. Amcvis implemented as a Matlab application.

After *space* has been loaded into Matlab via *spaceM* (Section 3.4), different plots can be generated by using *Amcvis*. Fig. 4.4a shows the real part of the eigenvalues of the reduced system. Note that the eigenvalues are purely real for this example. With these two eigenvalues, the color of the sampled points in the remaining figures is specified. Fig. 4.4c shows the $\mathcal{S}_{virt}$ space, while Fig. 4.4g the $\mathcal{S}_\lambda$ space. The red points represent the calculated DC points (see Section 3.2.1) for an input range $V_{nin} \in [-5, 5]$ V with an input step of 0.5 V. The remaining subfigures of Fig. 4.4 show various plots with randomly chosen constellations of variables. The voltage labels correspond to the node names shown in Fig. 4.2 as well as to the internal nodes of the operation amplified from Appendix C. As observed, especially in Fig. 4.4f, the eigenvalues are well suited for the distinction of the behavior of the pointwise linearized system.

Unlike the $\mathcal{S}_\lambda$ space, the $\mathcal{S}_{virt}$ space shown in Fig. 4.4c, clearly shows the different dynamic behaviors of the system without an overlapping of the sample points with different eigenvalues. However, the task of *Elsa* is to build a HA in the $\mathcal{S}_\lambda$ domain and not the $\mathcal{S}_{virt}$ domain, even though the overlapping of these points might challenges the model abstraction approach.

A question that comes up when observing the $\mathcal{S}_\lambda$ and $\mathcal{S}_{virt}$ spaces is: can the HA be build in the $\mathcal{S}_{virt}$ space? As $\mathcal{S}_{virt} \subset \mathcal{S}_o$, building a HA in the $\mathcal{S}_{virt}$ domain is equivalent to selecting specific entities from the $\boldsymbol{x}$ vector and abstracting the model to these values. Obviously, this would result in a model with a bad accuracy as well as in the loss of the ability to reconstruct precisely the elements of $\boldsymbol{x}$. However, the $\mathcal{S}_{virt}$ space can be used to aid the model abstraction process as will be explained in Section 4.5.

Additional to building the HA in the $\mathcal{S}_\lambda$ domain, the generated model is simulated in this domain as well. As observed in Fig. 4.4g, building as well as simulating the HA in this domain is not an

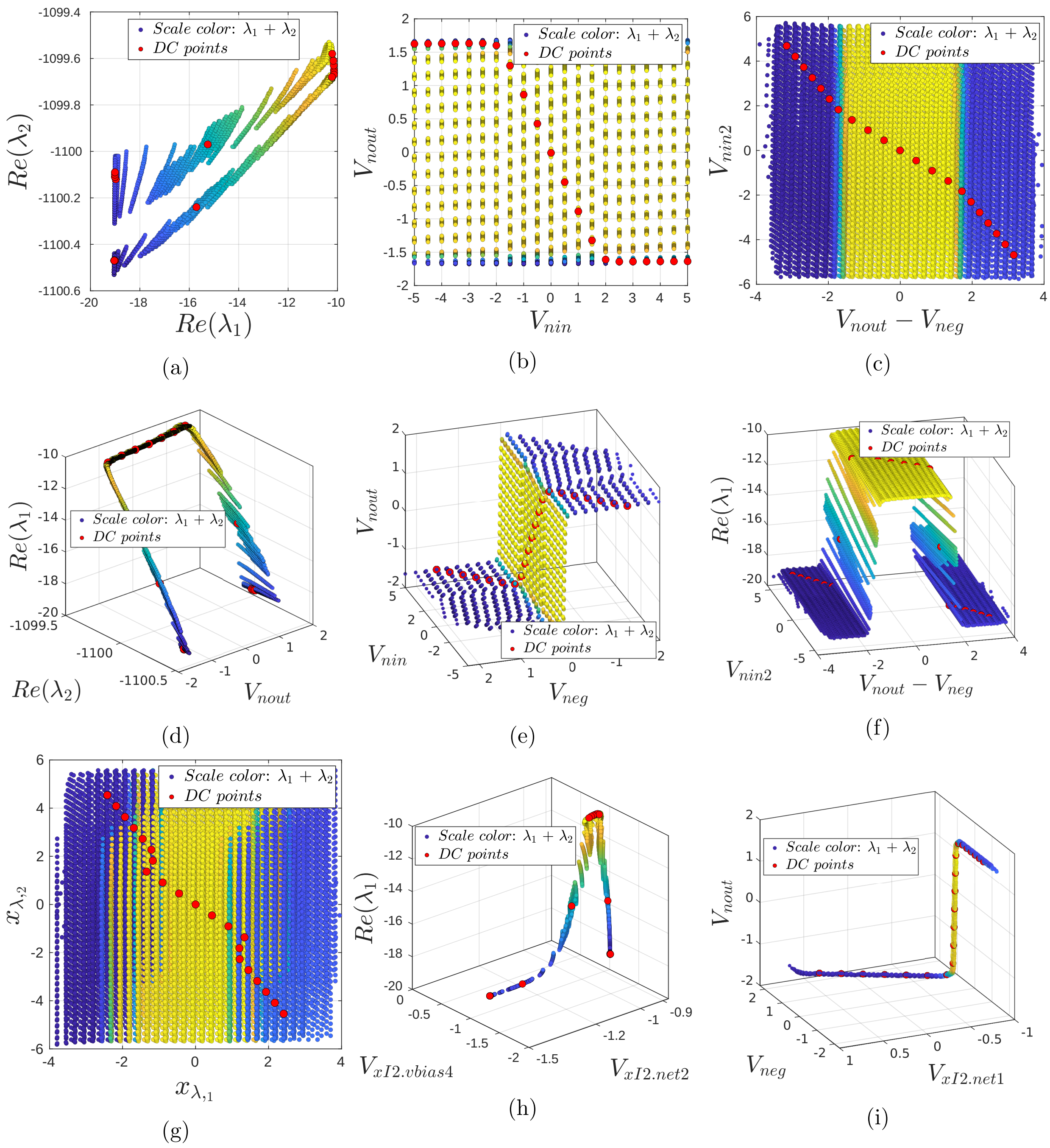


Fig. 4.4. Various plots generated with *Amcvis*. In (a) the eigenvalues of the reduced system are illustrated, while in (c) the $\mathcal{S}_{virt}$ space and in (g) the $\mathcal{S}_{\lambda}$ space are presented. In the remaining figures, randomly chosen constellations of variables are shown.

easy task. Considering the building aspect, each sampled point generally exhibits a different system behavior, which can be traced back to the eigenvalues (Section 2.3.1). Moreover, one can simply imagine that projecting a high dimensional system from the $\mathcal{S}_o$ domain, in this case with $r = 14$ dynamic states, to a space ($\mathcal{S}_\lambda$) with a lower order, in this case $m = 2$, will in some cases result in points overlapping with different dynamic behaviors. Additionally, for each point in the $\mathcal{S}_\lambda$ domain the transformation matrices $\boldsymbol{F}$ and $\boldsymbol{H}$ (Section 3.2.2) vary, which favors this overlapping, and thus challenges the abstraction process.

There are different settings that can be provided to *Vera* to control the overlapping of the sampled points. On the other hand, this generally result in greater variations in the transformation matrices, which challenges the abstraction process as well. However, as will be examined in Section 4.5, the presented approach can overcome such challenges with a little additional computational effort. Nonetheless, in the following the worst sampling case will be considered, that is the presence of strongly overlapping points with different behaviors in the $\mathcal{S}_\lambda$ space.

4.3 Initialization

As stated at the beginning of this chapter, *Elsa* is provided with the structure *space* and an option file called *SpaceOptions*. The initialization process is illustrated in Fig. 4.5.

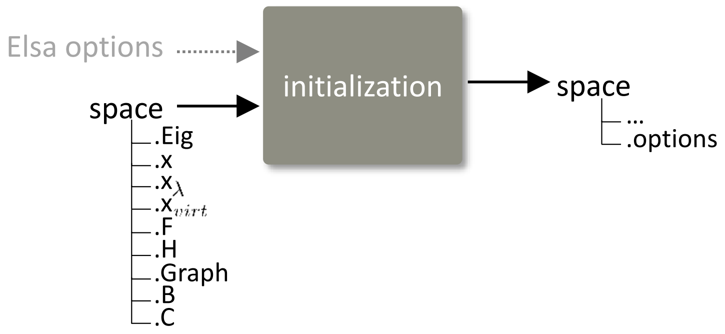


Fig. 4.5. Initiation block of *Elsa*.

During this initialization phase, some basic functions are called to check if the options provided are up to date, valid, and compatible. If this is not the case, the program is terminated and possible fixes are displayed. If no errors are detected, the structure *options* is created from *SpaceOptions* and assigned as an additional field in *space* named *space.options* as shown in Fig. 4.5. Note that, the variables contained in *space* are as described in Section 3.3.

4.4 Location Identification

The next step in the abstraction process is to identify the locations *Loc* of the HA. Each location $loc \in Loc$ is defined as a pair consisting of a group and a region, such that:

$$loc = g(j)r(k), \tag{4.3}$$

where j is the group counter, while k is the region counter.

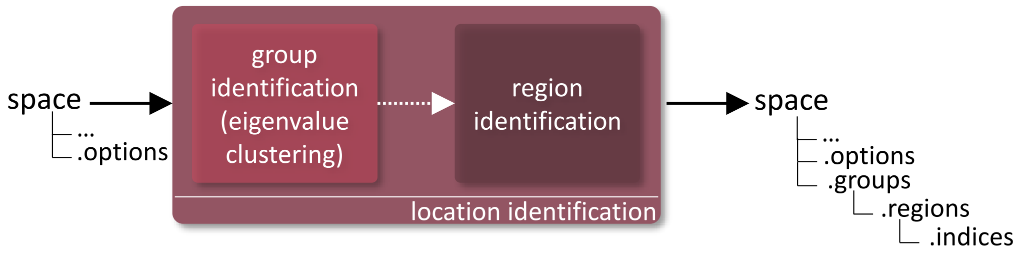


Fig. 4.6. Overview of the location identification block of *Elsa*.

The location identification can be divided into two blocks: the group identification and the region identification. Fig. 4.6 shows the location identification block of *Elsa* and its consisting subblocks.

As illustrated, the location identification starts with the group identification. For each of the found groups, a region identification is performed. By that, a group is divided into one or several regions. Hence, the indices of the underlying sampled points are labeled according to the group and region they belong to. Why a group followed by a region identification is necessary, will be clear at the end of this section.

4.4.1 Group Identification: Eigenvalue Clustering

The group identification consists of grouping the sampled points with similar eigenvalues obtained from the pointwise linearization of the system (see Section 3.2.2). More precisely, for all sampled points, the eigenvalues presented in $\mathbf{\Lambda}$ (Eq. (3.21)) are clustered. The clustering of these eigenvalues is processed through several steps:

1. scale the data set
2. find the optimal number of clusters if necessary
3. perform k-means clustering on the scaled data set

These steps are necessary as various options can be applied to optimize the accuracy of the generated HA.

Scaling the Data Set

In order to avoid working with complex values, each eigenvalue is divided into a pair of two elements, one representing the real part and one representing the imaginary part. As *Vera* demanded that all finite eigenvalues of the system are distinct (Assumption 1), $\mathbf{\Lambda}$ is a diagonal matrix. Although *Elsa* is generally able to handle eigenvalue matrices in the JCF form (see Section 4.5), for simplicity the approach will be described based on a diagonal $\mathbf{\Lambda}$ matrix.

By dividing each eigenvalue into two parts, the matrix $\boldsymbol{Eig}$ can be created that hosts the eigenvalues of a single point row-wise. Therefore, with a specified reduction order m and the l sampled points, the matrix hosting the eigenvalues is $\boldsymbol{Eig} \in \mathbb{R}^{l \times 2m}$:

$$\boldsymbol{Eig} = \begin{bmatrix} Re(\lambda_{1,1}) & Im(\lambda_{1,1}) & Re(\lambda_{1,2}) & Im(\lambda_{1,2}) & \dots & Re(\lambda_{1,m}) & Im(\lambda_{1,m}) \\ Re(\lambda_{2,1}) & Im(\lambda_{2,1}) & Re(\lambda_{2,2}) & Im(\lambda_{2,2}) & \dots & Re(\lambda_{2,m}) & Im(\lambda_{2,m}) \\ \vdots & \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ \underbrace{Re(\lambda_{l,1})}_{Re(\boldsymbol{\lambda}_1)} & \underbrace{Im(\lambda_{l,1})}_{Im(\boldsymbol{\lambda}_1)} & Re(\lambda_{l,2}) & Im(\lambda_{l,2}) & \dots & Re(\lambda_{l,m}) & Im(\lambda_{l,m}) \end{bmatrix} \tag{4.4}$$

For the running example from Section 4.1 with $l = 18500$ sampled data points and the reduced order $m = 2$, the diagonal matrix for each sampled point is $\boldsymbol{\Lambda}_i \in \mathbb{R}^{2\times 2}$, and thus the data set becomes $\boldsymbol{Eig} \in \mathbb{R}^{l\times 4}$ as given by Eq. (2.42).

There exist different options to scale the sampled eigenvalues. Four methods have been implemented that perform this task. All methods scale each column of Eq. (4.4) separately. Hence, the eigenvalues in $\boldsymbol{Eig}$ are scaled by the scaling matrix $\boldsymbol{K} \in \mathbb{R}^{l\times 2m}$ such that the new data set $\widetilde{\boldsymbol{Eig}}$ is given by the element-wise multiplication of these matrices:

$$\widetilde{\boldsymbol{Eig}} = \boldsymbol{Eig} \odot \boldsymbol{K}$$

The default method is to scale the row elements of each column of the data set $\boldsymbol{Eig}$ with the norm of the column. For example, considering the i^{th} column with i being an odd number, the magnitude of the column vector $Re(\boldsymbol{\lambda}_i)$ (see Eq. (4.4)) is:

$$||Re(\boldsymbol{\lambda}_i)|| = \sqrt{Re(\lambda_{1,i})^2 + Re(\lambda_{2,i})^2 + \cdots + Re(\lambda_{l,i})^2}$$

This can be thought of as if each row element of the column vector $Re(\boldsymbol{\lambda}_i)$ represents a dimension. Next, each row element of the column vector $Re(\boldsymbol{\lambda}_i)$ is divided by this magnitude. Hence, $\boldsymbol{K}$ is:

$$\begin{aligned} \boldsymbol{K} &= \begin{bmatrix} \frac{1}{||Re(\boldsymbol{\lambda}_1)||} & \frac{1}{||Im(\boldsymbol{\lambda}_1)||} & \frac{1}{||Re(\boldsymbol{\lambda}_2)||} & \frac{1}{||Im(\boldsymbol{\lambda}_2)||} & \cdots & \frac{1}{||Re(\boldsymbol{\lambda}_m)||} & \frac{1}{||Im(\boldsymbol{\lambda}_m)||} \\ \frac{1}{||Re(\boldsymbol{\lambda}_1)||} & \frac{1}{||Im(\boldsymbol{\lambda}_1)||} & \frac{1}{||Re(\boldsymbol{\lambda}_2)||} & \frac{1}{||Im(\boldsymbol{\lambda}_2)||} & \cdots & \frac{1}{||Re(\boldsymbol{\lambda}_m)||} & \frac{1}{||Im(\boldsymbol{\lambda}_m)||} \\ \vdots & \vdots & \vdots & \vdots & \ldots & \vdots & \vdots \\ \frac{1}{||Re(\boldsymbol{\lambda}_1)||} & \frac{1}{||Im(\boldsymbol{\lambda}_1)||} & \frac{1}{||Re(\boldsymbol{\lambda}_2)||} & \frac{1}{||Im(\boldsymbol{\lambda}_2)||} & \cdots & \frac{1}{||Re(\boldsymbol{\lambda}_m)||} & \frac{1}{||Im(\boldsymbol{\lambda}_m)||} \end{bmatrix} \\ &= \begin{bmatrix} \boldsymbol{k}_1 & \boldsymbol{k}_2 & \ldots & \boldsymbol{k}_{2m} \end{bmatrix} \end{aligned} \tag{4.5}$$

Note that for the case the magnitude of a column vector from Eq. (4.4) is zero, the corresponding column in $\boldsymbol{K}$ is set to one. This scaling brings the advantage of comparing the eigenvalues regardless of their size.

On top to the default method (a), the remaining three implemented scaling methods are:

(b) scaling performed similar to the dominant pole reduction i.e. poles closes to zero are more weighted than poles further to the left in the complex s-plane

(c) scaling the eigenvalues till they have the same exponent pow in the scientific notion $\{d\times 10^{pow} \mid 1 \leq d \leq 10\}$

(d) scaling the eigenvalues by a vector provided by the user

In the simplest case, the scaling method (b) first computes the mean values across the columns of Eq. (4.4), that is for the i^{th} column with i being an odd number:

$$\overline{Re(\boldsymbol{\lambda}_i)} = mean(Re(\lambda_{1,i}), Re(\lambda_{2,i}), \ldots, Re(\lambda_{l,i}))$$

The corresponding i^{th} column vector $\boldsymbol{k}_i \in \mathbb{R}^{l\times 1}$ in $\boldsymbol{K}$ is then:

$$\boldsymbol{k}_i = \begin{bmatrix} 10^{\kappa} & 10^{\kappa} & \ldots & 10^{\kappa} \end{bmatrix}^T, \tag{4.6}$$

such that:

$$\kappa = k_p \cdot floor(log_{10}(|\overline{Re(\boldsymbol{\lambda}_i)}|)) \tag{4.7}$$

Where k_p is a constant that is specified by the user, with the default value set to $k_p = -2$. Note that, in this case the corresponding column $Im(\lambda_{1,i+1})$ is scaled with the same vector, that is $\boldsymbol{k}_{i+1} = \boldsymbol{k}_i$. Moreover, if Eq. (4.7) returns infinity, the exponents in Eq. (4.6) are set to 0.

The scaling method (c) uses Eqs. (4.6, 4.7) with $k_p = -1$, while method (d) uses an input vector provided by the user to assign the column vectors from Eq. (4.5) directly. Note that only the simplest versions of the scaling methods were discussed here. Nonetheless, at end of scaling process, $\boldsymbol{Eig}$ has been scaled by $\boldsymbol{K}$ to form $\widetilde{\boldsymbol{Eig}}$.

In addition to scaling $\boldsymbol{Eig}$, columns can also be removed. As complex eigenvalues always come in pairs, they can be optionally considered only once during the clustering process. Hence, the corresponding columns in Eq. (4.4) are removed.

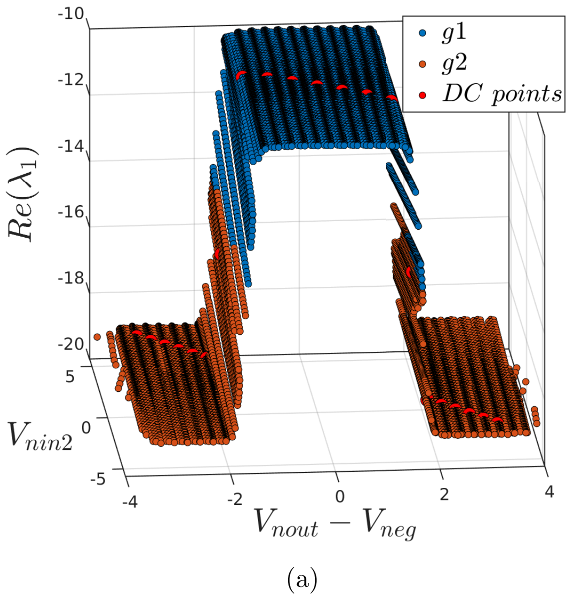
Finding the Optimal Number of Clusters

The next step in the group identification involves finding the optimal number of clusters for this data set. If this number has not been specified in the options provided to *Elsa*, the optimal number of clusters is calculated according to the standard *evalclusters* function (see Section 2.5.3) provided by Matlab. In this function call, the clustering algorithm is set to k-means (Section 2.5.1) and the clustering evaluation criterion is set to use the silhouette coefficient (Section 2.5.3). The function then launches k-means several times, each instance with a different number of clusters, up till a specified maximum. The results from the various clustering are then compared using the silhouette coefficient. Finally, the optimal number of clusters is returned from the clustering that yielded the best silhouette coefficient.

To illustrate this process, consider the running example from Section 4.1. In fact, an extensive clustering analysis on this example has been handled in Section 2.5. More precisely, for the running example, the previous mentioned scaling (see Eq. (4.4)) is not needed. Moreover, as the eigenvalues in $\boldsymbol{Eig}$ are purely real, the second and fourth columns of this matrix are zero vectors. Hence, the eigenvalues of the reduced system can be illustrated in a 2D space (Fig. 2.3). By using the data set $\boldsymbol{Eig}$, which is illustrated in Fig. 2.3 (first and third column vectors), the optimal number of clusters is evaluated using the silhouette coefficient as an evaluation criteria. The result of this analysis is shown in Fig. 2.11b. As observed, the optimal number of clusters is two. Thus, k-means is launched as stated in Section 2.5.1 with $k_m = 2$ to partition the eigenvalues as shown in Fig. 2.4. The corresponding silhouette values of all sample points are shown in Fig. 2.11a.

Clustering the Data Set

With the optimal number of clusters at hand, the matrix of scaled eigenvalues $\widetilde{\boldsymbol{Eig}}$ is clustered using the k-means clustering algorithm presented in Section 2.5.1. Of course other clustering algorithms can be used here as well. Nonetheless, as stated in Section 2.5.1, k-means is a partitioning clustering algorithm. As our aim in the group identification is to partition the eigenvalues of the linearized system into groups with similar eigenvalues, k-means represent an optimal approach compared to other clustering methods. This can be clearly visualized in the $\mathcal{S}_{virt}$ or $\mathcal{S}_{\lambda}$ space as illustrated in Fig. 4.7, where k-means has been used to identify the cluster groups $g1$ and $g2$. Note that the unlabeled data in each state spaces, $\mathcal{S}_{virt}$ and $\mathcal{S}_{\lambda}$, was previously illustrated in Fig. 4.4c and Fig. 4.4g, receptively.



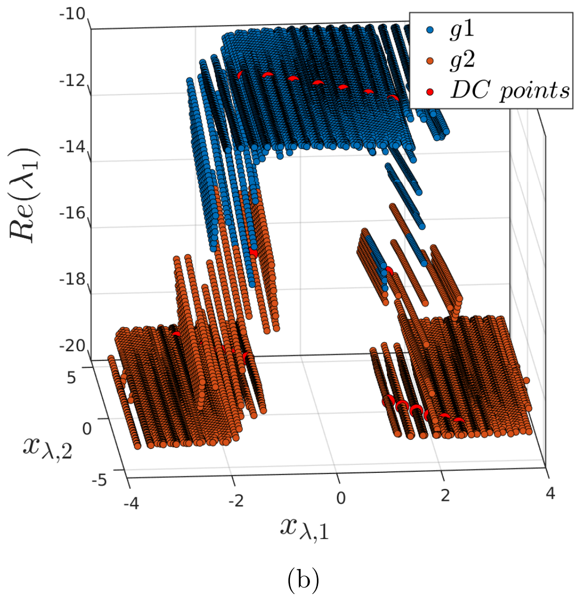


Fig. 4.7. Result of the group identification in the (a) $\mathcal{S}_{virt}$ and (b) $\mathcal{S}_{\lambda}$ space plotted against the real part of the first eigenvalue. The red points show the calculated DC points.

According to Section 2.3, eigenvalues are directly linked to the system behavior. Hence, sampled points with different eigenvalues from the linearized system correspond to different dynamic behaviors of the system. This implies that eigenvalue clustering can be used to distinguish the system behaviors. This becomes clear when the cluster groups identified are visualized in the $\mathcal{S}_{virt}$ space or $\mathcal{S}_{\lambda}$ space as shown in Fig. 4.7. According to Section 4.1, the circuit exhibits a limiting behavior. This limiting behavior, which can be observed in group $g2$ of Fig. 4.7, can as well as be observed in Fig. 4.8, which shows one of the $\boldsymbol{x}_{virt}$ state variables (see Eq. (4.2)) drawn against the output of the system.

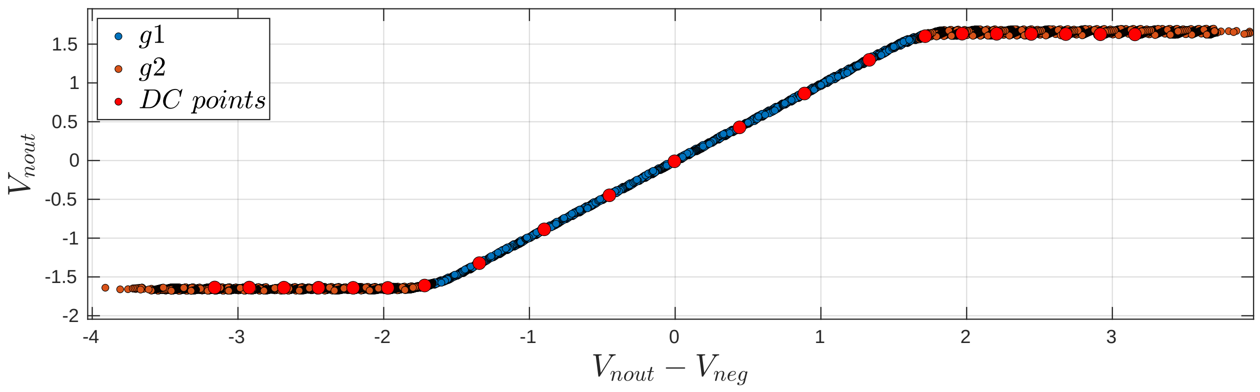


Fig. 4.8. The first $\boldsymbol{x}_{virt}$ state, $V_{nout} - V_{neg}$, drawn against the output of the system V_{nout} after the group identification. The red points show the calculated DC points.

What is important to notice in Fig. 4.7a, is that different regions of the group $g2$, the region with

$V_{nout} - V_{neg} \leq -1.65$ and the other region with $V_{nout} - V_{neg} \geq 1.65$, have the same eigenvalues. Thus, these points exhibit a similar homogeneous system responses (Section 2.3.1), even though they belong to different portions of the state spaces. On the other hand, the DC points (Section 3.2.1), colored in red in the $\mathcal{S}_{virt}$ space from Fig. 4.7a, show that the two regions belonging to the same group $g2$ contain different operating points. The same can be observed in the $\mathcal{S}_{\lambda}$ space from 4.7b. Interpreting this aspect, it becomes clear that sampled points belonging to different regions of a group can exhibit different system responses which can be traced back to their particular solutions.

This becomes even more obvious when considering Fig. 4.8. Keeping in mind that the output response of a linear system is as given in Eq. (2.29), it becomes clear that the particular solution in the regions of $g2$ differ. Moreover, as indicated by the DC points, the operating points differ as well, which can be traced back to the input of the system. Hence, after performing an eigenvalue clustering and thus clustering the sampled points into groups, a region identification must be performed on the groups to distinguish them into regions, thereby differentiating the overall behavior of the sampled points.

4.4.2 Region Identification

For every identified group, a region identification is performed. Several algorithms have been deployed that execute this task. In general, these algorithms can be classified into two categories depending on the type of analysis they perform:

- graph analysis: distance or polytope
- clustering analysis: k-means, DBSCAN, mean shift, or OPTICS

The graph-based methods process the directed connection graph *Vera* generated during sampling (see Section 3.3). The clustering-based methods simply perform a cluster analysis on the points belonging to the same groups. In all methods, the state space in which the analysis is performed can be chosen. This will be deeper examined in Section 4.5. For the remaining of this section, the $\mathcal{S}_{virt}$ space will be used. For the clustering methods, on top of the specified state space, additional data such as the inputs of the system can be used in the analysis. In the following, several region identification methods will be examined, that effectively divide groups into regions.

Graph Based Methods

The graph-based methods make use of the directed graph generated by *Vera*. For the running example, this graph is shown in Fig. 4.9.

This type of analysis first determines the subgraphs of each group. For that, the connection graph of each group is analyzed separately. Note that, the connection graph of each group is formed by the nodes that belong to the group. If a connection graph is not well-connected, subgraphs can be determined. Hence, subgraphs are parts of a connection graph that are not connected.

Usually, *Vera* returns a well-connected graph for the entire state space as observed in Fig. 4.9. Upon performing a group identification, the directed graph from *Vera* is divided into connection graphs for each group. As observed in Fig. 4.9 for the running example, the directed graph from *Vera* is divided into two connection graphs corresponding to the groups $g1$ and $g2$. Looking at this from a different perspective, when a region identification is performed on $g2$, the points belonging

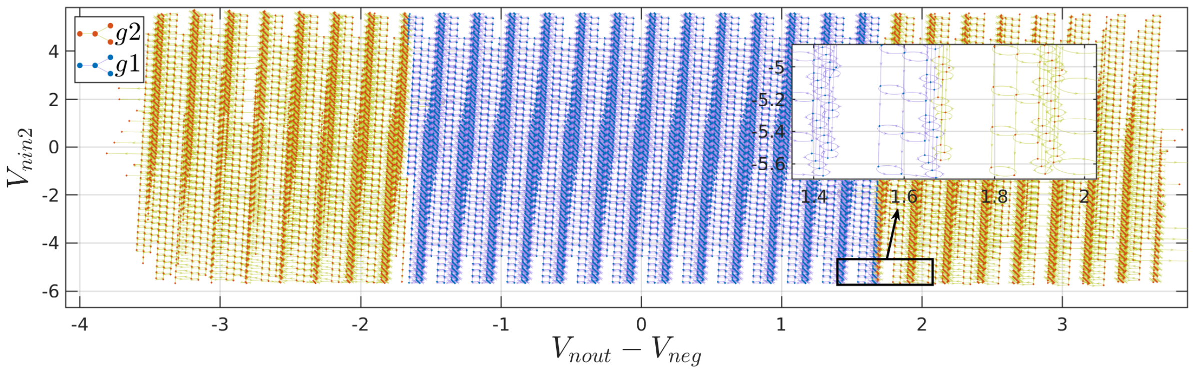


Fig. 4.9. Connection graph generated by *Vera* in the $\mathcal{S}_{virt}$ space.

to $g1$ are not considered, which corresponds to removing these points from the directed graph of the entire sampled space. Thus, the remaining points that belong to $g2$ form at least two subgraphs as observed in Fig. 4.9. In contrast, $g1$ has probably only one subgraph which is at the same time the connection graph of this group. Additional subgraphs may exist depending on the connections.

If the algorithm detects that a group has multiple subgraphs, the region identification is launched. If this is not the case, the region identification is skipped. As observed in Fig. 4.9, the points belonging to $g1$ are well-connected, while the point belonging to $g2$ are not and are therefore subjected to a region identification. Hence, the task of the graph-based methods becomes finding the regions of $g2$ from the identified subgraphs. Two graph-based methods have been implemented that execute the region identification:

Gdist method uses the Euclidean distance

Gpoly method examines the volumes of the identified polytopes

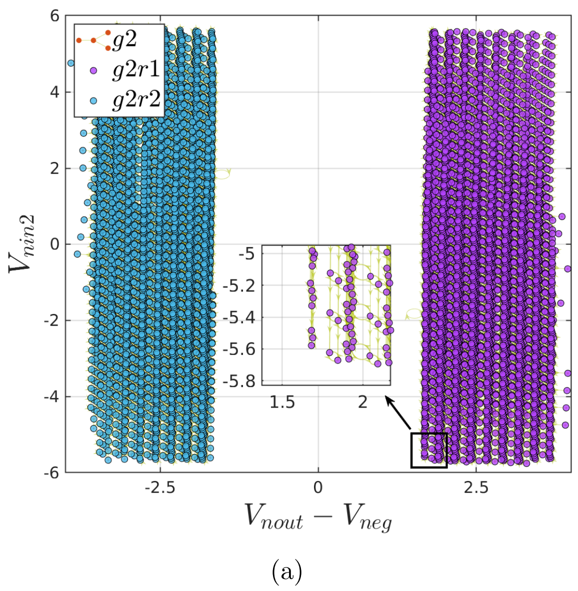
The basic algorithm of the both methods is stated in Algorithm 5 using the notation from Eq. (4.3).

Algorithm 5 Graph-based region identification methods

```
 1: procedure $r(k) = regions(g(j), graph)$
 2:     find subgraphs in $graph$
 3:     if $g(j)$ has more than one subgraph then
 4:         find the largest two subgraphs and denote them as the main subgraphs
 5:         add the remaining subgraphs to the main subgraphs based on the selected method
 6:         label the nodes of the subgraphs to the two locations $g(j)r1$ and $g(j)r2$
 7:     else
 8:         all nodes of the $graph$ belong to one location $g(j)r1$
 9:     end if
10: end procedure
```

As described in Algorithm 5, first the connection graph of a group is analyzed (line 2). If the connection graph can be divided into subgraphs (line 3), the region identification is launched and the largest two subgraphs are identified. As given in line 5, all remaining subgraphs are merged into the largest two subgraphs of the current group using either the *Gdist* or *Gpoly method*. Finally, the nodes of each subgraph are label as points belonging to the same region. In the case only one subgraph was determined, all underlying nodes belong to same region. Hence, the graph based methods either divide a group into two regions, are assign all points of a group to the same region based on the graph properties.

The *Gdist method* calculates in the $\mathcal{S}_{virt}$ space the distances from all nodes of the two main subgraphs to the nodes of the current inspected one. This subgraph is then merged into the closest main subgraph. Fig. 4.10a shows the result of this method on the running example.



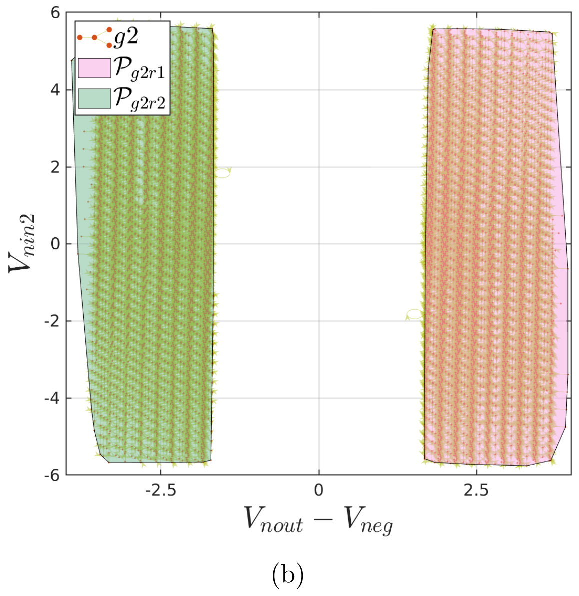


Fig. 4.10. Results of the region identification in the $\mathcal{S}_{virt}$ using the graph-based methods. In (a) the result of the *Gdist method* is presented, while (b) shows the result of the *Gpoly method*.

The *Gpoly method* creates from the nodes of each of the two main subgraphs a polytope in the $\mathcal{S}_{virt}$ space. At each iteration, one of the remaining subgraphs is examined, by enlarging the two previously identified polytopes by the vertices of this subgraph. This usually increases the volume of these polytopes. The new volume of each polytope is then computed, and the nodes of the subgraph are added to the polytope which yields the smallest volume increase. The result of this method is shown in Fig. 4.10b.

The graph-based methods are sensitive to a bad connected sampled state space and to a group identification that yielded bad connection graphs. Even though several correcting algorithm exist, there is no guarantee that these methods will yield the best results. Besides this, these methods consume a fair amount of time. Yet, for the case that each group has only two regions, which is often the case for a SISO system, these methods are quite reliable.

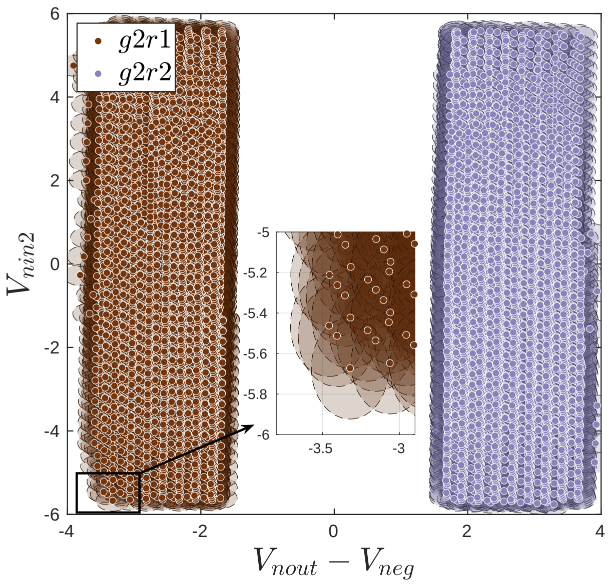
Clustering Based methods

Due to the stated disadvantages of the graph-based methods, the clustering-based methods were developed. Moreover, for the case that the system has more than one input, which often implies that the groups can have more than two regions, the cluster based methods should be used. Four clustering-based methods have been deployed that can identify the regions of a group: k-means, DBSCAN, mean shift, and OPTICS.

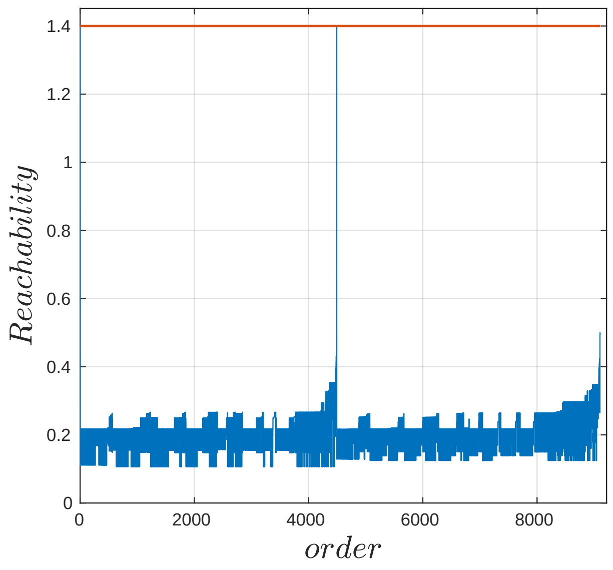
K-means and DBSCAN have been previously explained in Section 2.5. The remaining clustering methods will not be presented in this dissertation. For more details see [ABKS] for OPTICS and [Yiz95] for mean shift.

In Fig. 4.11a, the result of the region identification performed by DBSCAN is illustrated for the running example. With ϵ_s representing the state space step in the $\mathcal{S}_{virt}$ domain, DBSCAN (see Section 2.5.2) with $N = 1$ and $\epsilon = 1.1 * \epsilon_s$, identified the two regions of $g2$ as shown.

This clustering can also be performed with k-means, OPTICS, or mean shift. In case OPTICS is used, *Elsa* can be launched in an interactive mode. The user is prompted to input a reachability distance. OPTICS is launched with the specified distance, and the number of clusters found is displayed to the user. At this point, the user can either change the reachability distance to obtain more or less clusters, or continue the analysis with the found clusters. For the running example, the reachability distance was set to 1.4, the corresponding reachability plot generated by OPTICS for group $g2$ is illustrated in Fig. 4.11b. The ordering of the points as processed by OPTICS is illustrated on the x-axis, while the y-axis illustrates the reachability distance. Note that, points that belong to the same cluster have a low reachability distance to their nearest neighbor. Hence, cluster are illustrated by valleys in the reachability plot. The wider the valley is, the denser the cluster. For the selected reachability distance, two large valleys can be observed in Fig. 4.11b. The result of this clustering is similar to the result of DBSCAN presented in Fig. 4.11a.



(a)



(b)

Fig. 4.11. In (a) the result of the region identification in the $\mathcal{S}_{virt}$ using the cluster method DBSCAN is shown. In (b) the reachability plot generated by OPTICS is presented. The red horizontal line indicates the reachability distance specified by the user.

After the regions of each group are identified, the location identification is complete. Each location can be described by a pair consisting of a group and a region as shown in Eq. (4.3). For the running example, group $g1$ has only one region $r1$ while group $g2$ has two regions, $r1$ and $r2$. Thus, this examples has 3 locations: $g1r1$, $g2r1$ and $g2r2$, yielding $Loc = \{g1r1, g2r1, g2r2\}$.

The region identification is not always necessary when the group identification is modified. In some cases the results of the group identification can be improved by clustering additionally to the eigenvalues, the $\boldsymbol{x}_{virt}$ values from the $\mathcal{S}_{virt}$ space. In this case, the clustering data set is formed from the eigenvalues stored in $\boldsymbol{Eig}$ and the $\boldsymbol{x}_{virt}$ sampled values. This enhances the group identification, and in some cases, replaces the region identification by additional identifying groups. As presented in Section 2.5.1, when k-means is used on the data set consisting of one of the $\boldsymbol{x}_{virt}$ states along with eigenvalues, three locations could be determined as shown in Fig. 2.7. In this case three groups were identified, with each group containing only one region. This can be considered a special case and does not always guarantee the best solution. However, performing first a group identification on the eigenvalues, followed by a region identification on the states, decreases the clustering overhead, as well as separates the identification problem into first distinguishing points according to their system behavior (eigenvalues), followed by distinguishing the labeled points based on their positions or connections in the state space. On top of that, additionally considering the inputs of the system in the region identification often provides even better results, but again increases the overhead. Hence, in the following these options will be skipped unless stated otherwise.

4.5 System Modeling

According to Section 2.4, a HA is defined as:

$$\text{HA} = (Loc, loc_0, \boldsymbol{x}_\lambda, \boldsymbol{x}_{\lambda,0}, inv, tran, grd, \boldsymbol{J}, \boldsymbol{u}, \boldsymbol{f})$$

Until this point, the locations Loc have been determined (Section 4.4), while the input vector $\boldsymbol{u}$ is usually given. The third block of *Elsa* as presented in Fig. 4.1 performs the system modeling. This involves describing the system behavior in each location, determining the validity of the location as specified by the invariants, and modeling the transitions between the locations using the guards and the corresponding jump functions.

One of the properties of *Elsa* is that the created HA has a linear system description. Therefore, the flow function $\boldsymbol{f}$ in each location is described using the linear state space representation introduced in Section 2.3.1. Since the HA is created in the $\mathcal{S}_\lambda$ space, the states space vector of the automaton is $\boldsymbol{x}_\lambda \in \mathbb{R}^m$, with a given initial vector $\boldsymbol{x}_{\lambda,0}$. Thus, to describe the flow function, the system matrix $\boldsymbol{A}_{loc}$ and the input matrix $\boldsymbol{B}_{loc}$ for each location $loc \in Loc$ are needed, as well as some reference linearization points as will be seen later.

Moreover, since the goal is to reconstruct all nodal voltages and currents in the $\mathcal{S}_o$ domain, a back-transformation is needed that transforms the result of a simulation or reachability analysis of HA in the $\mathcal{S}_\lambda$ space back to the original state space $\mathcal{S}_o$ of the system. Hence, the result of the system modeling block described in this section, as shown in Fig. 4.12, is a HA accompanied by additional matrices that define the back-transformation in each location. Note that, blocks in Fig. 4.12 with dashed borders represents blocks that can be skipped depending on the specified options.

According to Fig. 4.12, the system modeling occurs in the $\mathcal{S}_{virt}$ space as well as in the $\mathcal{S}_\lambda$ space. Nonetheless, the modeling can be also exclusively performed in the $\mathcal{S}_\lambda$ space if desired. In this case the underlying blocks which are executed in the $\mathcal{S}_{virt}$ are processed in the $\mathcal{S}_\lambda$ space. Moreover, in this case the transformation block is skipped. Note that, performing the system modeling exclusively in the $\mathcal{S}_{virt}$ space is not possible, as the modeling process is based on KCF (see Section 2.2.2).

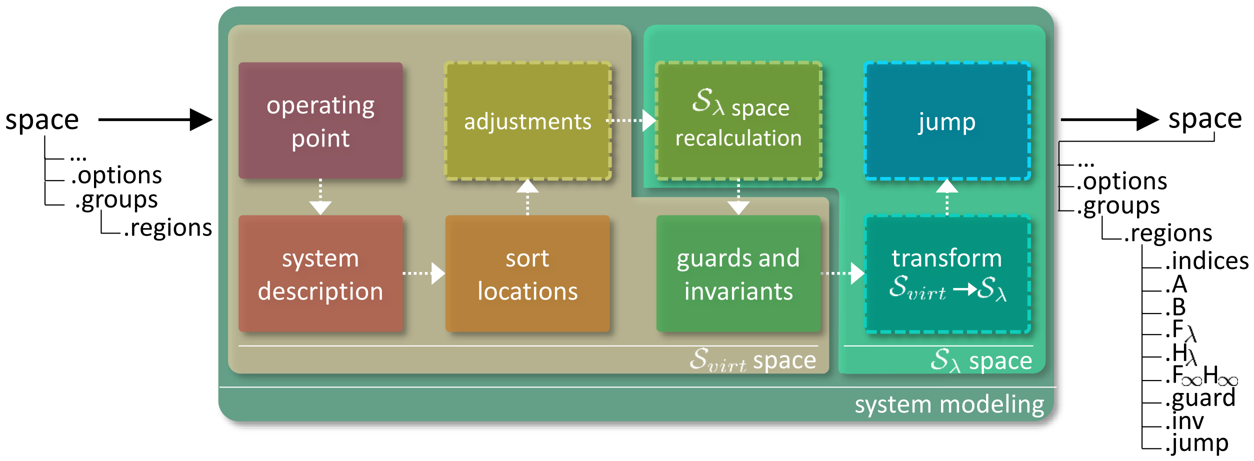


Fig. 4.12. Overview of the system modeling block of *Elsa*.

As mentioned, one of the current tasks in this section is to create a HA with a linear flow function $\boldsymbol{f}$ restricted to the linear state space representation from Section 2.3.1. Previously, *Vera* sampled the nonlinear circuit and resulted in Eq. (3.23) for each sampled point. Considering only the first row (dynamic part) of this equation:

$$\Delta\dot{\boldsymbol{x}}_\lambda = \boldsymbol{\Lambda}\Delta\boldsymbol{x}_\lambda + \tilde{\boldsymbol{B}}_\lambda\Delta\boldsymbol{u}\,, \tag{4.8}$$

this equation states the linearized dynamic behavior of each sampled point in the $\mathcal{S}_\lambda$ space. With Assumption 2 (see Section 3.2.2), and by inserting the algebraic part stated in Eq. (3.25) into Eq. (3.27), the following equation can be obtained:

$$\begin{aligned}\Delta\boldsymbol{x} &= \boldsymbol{F}_\lambda\Delta\boldsymbol{x}_\lambda - \boldsymbol{F}_\infty\tilde{\boldsymbol{B}}_{\infty,red}\Delta\boldsymbol{u}\\ &= \boldsymbol{F}_\lambda\Delta\boldsymbol{x}_\lambda - \boldsymbol{F}_\infty\boldsymbol{H}_\infty\boldsymbol{B}_{\infty,red}\Delta\boldsymbol{u}\end{aligned} \tag{4.9}$$

Eq. (4.9) shows that for each sampled point a transformation exists that allows to calculate its value in the $\mathcal{S}_o$ space by using only the $\mathcal{S}_\lambda$ space along with the input of the system. The only problem is that Eqs. (4.8, 4.9) are only valid in a small region around the linearization points, as at every sampled point, the system was linearized (see Algorithm 3 and Algorithm 4).

Thus, the current tasks for the flow function $\boldsymbol{f}$ in each location of the HA becomes clear: describe the pointwise linear equations in each location of a HA by a well-chosen representing equation which is valid for all sampled points belonging to the location.

During system linearization, a linearization point is chosen, and the system is linearized around it. As long as the system stays in a close range to the linearization point, the linearized equations are valid. Here, we went both ways. We first linearized the system for the sampled points. Secondly, instead of choosing one that represents the remaining points, we analyzed the eigenvalues of the linearized system and clustered them to locations of different behaviors (Section 4.4). By that, we identified the location where the sampled points have similar behaviors. Next, a representing equation in each location is determined that describes the behavior of all points belonging to the location. Moreover, the range in the $\mathcal{S}_\lambda$ space where these equations are valid are defined by the invariants of the HA. The transitions between these invariants are defined by the guards and the jump functions. Hence, it becomes clear that the approach is actually building a HA step by step.

In the following, this procedure will be analyzed analogously to Fig. 4.12. Moreover, as illustrated in this figure, the approach is examined for the case $\mathcal{S}_{virt}$ is considered along with the $\mathcal{S}_{\lambda}$ space in the modeling process, as this usually yields more reliable results. For the case the modeling was performed entirely in the $\mathcal{S}_{\lambda}$ space, only the sections as specified in Fig. 4.46 are used.

4.5.1 Calculating the Operating Points

After the locations of the HA have been identified (Section 4.4), the sampled points are associated to the corresponding locations. Among these points are also the DC points as shown in Fig. 4.13.

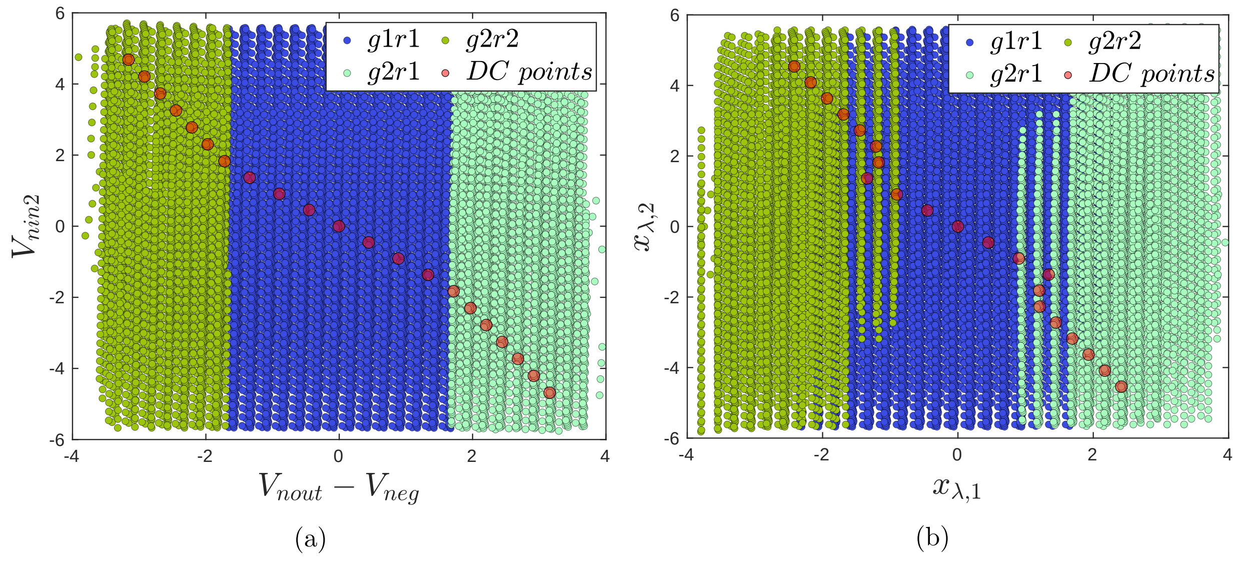


Fig. 4.13. Result of the location identification presented (a) in the $\mathcal{S}_{virt}$ and (b) in the $\mathcal{S}_{\lambda}$ space for the running example.

Consider the sampled $\mathcal{S}_{virt}$ space of the running example illustrated in Fig. 4.13a. From the set of DC points in each location, a representing operating point needs to be chosen. For this, first a global reference point must be computed. Unless specified otherwise, the global reference is assumed to be the point at which there is no input voltage, that is $\boldsymbol{u} = \mathbf{0}$. The location containing the global reference point is referred to as the center location. For the running example, the global reference point is $\boldsymbol{x}_{virt,glob} = \mathbf{0}$ in the $\mathcal{S}_{virt}$ domain, implying that the center location is $g1r1$.

For each location the operating point is chosen as the DC point that is closest to the global reference by using the Euclidean distance in the $\mathcal{S}_{virt}$ space:

$$d = || \boldsymbol{x}_{virt,DC} - \boldsymbol{x}_{virt,glob} ||_2$$

For the center location the reference point is identical with the operating point. Moreover, several options exist that can affect the identification of the operating points. The most important one specifies which operating point should be considered i.e. the i^{th} closest, the i^{th} furthest, or the nearest to the global reference which is the default case used here.

The result of the operating points identification for the running example is illustrated in Fig. 4.14. For each of the three locations of the HA, the yellow marked DC points are chosen as the representing operating points. The black circles in Fig. 4.14 show the distance from the selected orating points to the global reference at $\boldsymbol{x}_{virt,glob} = \mathbf{0}$.

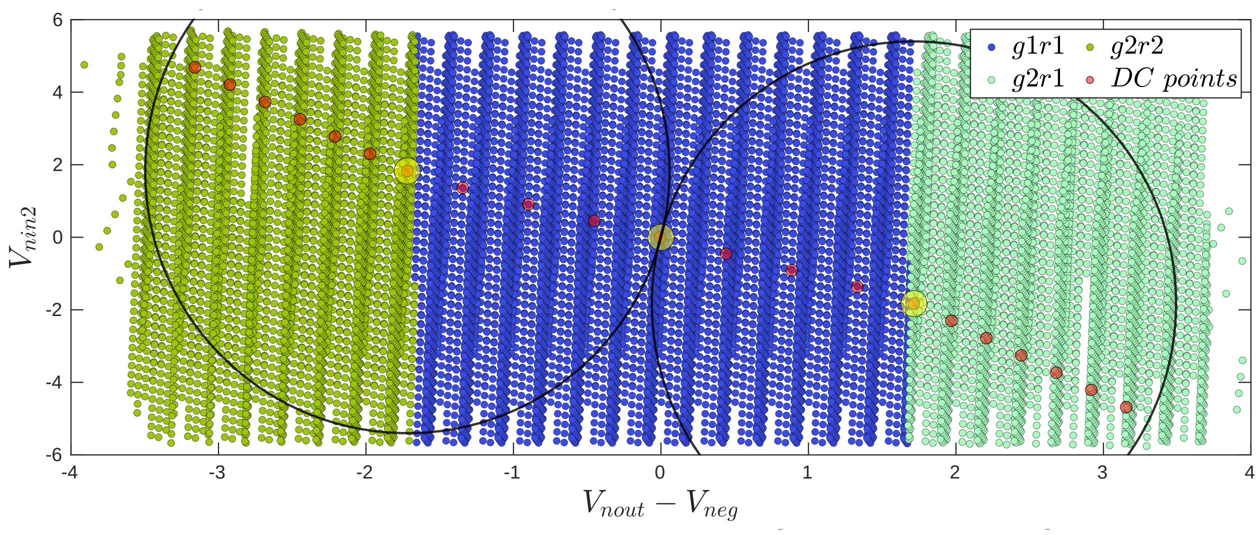


Fig. 4.14. The operating point identification performed in $\mathcal{S}_{virt}$. The yellow points indicated the chosen DC points in each location. The black circles indicate the distance from these points to the global reference point.

By identifying an operating point, all the values stated in Section 3.3 for this point are obtained. This includes, at the operating input voltage $\boldsymbol{u}_{op}$, the operating point in the $\mathcal{S}_{virt}$ domain denoted $\boldsymbol{x}_{virt,op}$, the operating point in the $\mathcal{S}_{\lambda}$ domain denoted $\boldsymbol{x}_{\lambda,op}$, and the operating point in the $\mathcal{S}_{o}$ domain denoted $\boldsymbol{x}_{op}$.

Several aspects need to be considered when identifying the operating points. This includes the case when there are DC points that are overlapping or equidistant from the global reference point. In this case the mean of these points is taken, and the operating point is identified by the sampled data point that is closest to this mean. Another case that needs to be considered is when there are no DC points in a location, as shown later in Section 7.4.2. In this case, first a test is performed to see if a DC point from the closest surrounding regions can be taken. If the surrounding DC points are too far, which is the case when the minimum distance from all sampled points of a location to each of the examined DC points is greater than the state space step, either a random sampled point is chosen as the operating point, or the mean of all points in a location is calculated and the operating point is chosen as the sampled point closest to the calculated mean value.

4.5.2 Determining the System Description

In the second block shown in Fig. 4.12, the different system behaviors of the HA in each location are determined. With the sampled data set divided into the locations (see Section 4.4), and the previous identified operating points in the $\mathcal{S}_o$, $\mathcal{S}_{virt}$ and $\mathcal{S}_{\lambda}$ spaces, the current aim is to model the system behavior in each location. More precisely, the system description task involves finding for each location $loc \in Loc$:

(a) the representing operating points $\boldsymbol{x}_{op}$, $\boldsymbol{x}_{\lambda,op}$, and the operating input $\boldsymbol{u}_{op}$

(b) a linear state space representation that is valid for the whole location i.e. all sampled points belonging to the location loc. This involves determining the system matrix $\boldsymbol{A}_{loc} \in \mathbb{R}^{m \times m}$ and the input matrix $\boldsymbol{B}_{loc} \in \mathbb{R}^{m\times k}$, such that the linear system behavior is given by:

$$\begin{aligned} \Delta\dot{\boldsymbol{x}}_\lambda &= \boldsymbol{A}_{loc}\Delta\boldsymbol{x}_\lambda + \boldsymbol{B}_{loc}\Delta\boldsymbol{u} \\ \implies \dot{\boldsymbol{x}}_\lambda - \cancel{\dot{\boldsymbol{x}}_{\lambda,op}} &= \boldsymbol{A}_{loc}(\boldsymbol{x}_\lambda - \boldsymbol{x}_{\lambda,op}) + \boldsymbol{B}_{loc}(\boldsymbol{u} - \boldsymbol{u}_{op}), \end{aligned} \tag{4.10}$$

for the case the operating point is a DC point, as:

$$\dot{\boldsymbol{x}}_{\lambda,op} = \boldsymbol{0}$$

(c) a back-transformation that is valid for the whole location *loc*. This involves finding the transformation matrices $\boldsymbol{F}_{loc} \in \mathbb{R}^{n\times m}$ and $\boldsymbol{L}_{loc} \in \mathbb{R}^{n\times k}$ that transforms the solution of Eq. (4.10) from the $\mathcal{S}_\lambda$ to the $\mathcal{S}_o$ space via:

$$\boldsymbol{x} = \boldsymbol{x}_{op} + \boldsymbol{F}_{loc}(\boldsymbol{x}_\lambda - \boldsymbol{x}_{\lambda,op}) - \boldsymbol{L}_{loc}(\boldsymbol{u} - \boldsymbol{u}_{op}) \tag{4.11}$$

Point (a) has already been covered in Section 4.5.1. The two remaining points ((b) and (c)) can be solved with the information at hand. For this, several methods have been deployed, which vary according to the points that are considered in determining the matrices $\boldsymbol{A}_{loc}$, $\boldsymbol{B}_{loc}$, $\boldsymbol{F}_{loc}$, and $\boldsymbol{L}_{loc}$:

op method	uses only the operating point
mean method	uses all sampled points belonging to the location
dc method	uses only the DC points belonging to a location
weight method	uses all sampled points belonging to the location with different specified weights

Points in a location can be divided into two categories: normal sample points and DC points. While DC points were computed by the DC analysis in *Vera* (see Section 3.2.1), the remaining sampled points were computed by stepping through the reduced state space $\mathcal{S}_\lambda$ and calculating the consistent solutions (see Section 3.2.3). Previously in Section 4.5.1, the operating points in each location were identified from the DC points. Considering Eqs. (4.8, 4.9), the *op method* which uses only the operating point of a location, basically assigns for each location $loc \in Loc$ the system, input, and transformation matrices to their values at the operating point:

$$\boldsymbol{A}_{loc} = \boldsymbol{\Lambda}_{op} \qquad \boldsymbol{B}_{loc} = \tilde{\boldsymbol{B}}_{\lambda,op} \tag{4.12}$$

$$\boldsymbol{F}_{loc} = \boldsymbol{F}_{\lambda,op} \qquad \boldsymbol{L}_{loc} = \boldsymbol{F}_{\infty,op}\boldsymbol{H}_{\infty,op}\boldsymbol{B}_{\infty,red,op} \tag{4.13}$$

The *mean method* computes the mean of the system, input, and transformation matrices over all l_{loc} sampled points belonging to a location:

$$\boldsymbol{A}_{loc} = mean(\boldsymbol{\Lambda}_1, \ldots, \boldsymbol{\Lambda}_{l_{loc}}) \qquad \boldsymbol{B}_{loc} = mean(\tilde{\boldsymbol{B}}_{\lambda,1}, \ldots, \tilde{\boldsymbol{B}}_{\lambda,l_{loc}}) \tag{4.14}$$

$$\boldsymbol{F}_{loc} = mean(\boldsymbol{F}_{\lambda,1}, \ldots, \boldsymbol{F}_{\lambda,l_{loc}}) \qquad \boldsymbol{L}_{loc} = mean(\boldsymbol{L}_1, \ldots, \boldsymbol{L}_{l_{loc}}) \tag{4.15}$$

With:

$$\boldsymbol{L}_i = \boldsymbol{F}_{\infty,i}\boldsymbol{H}_{\infty,i}\boldsymbol{B}_{\infty,red,i}$$

Similarly to the *mean method*, the *dc method* computes the mean of the system, input, and transformation matrices. However, only the DC points belonging to a location are considered in this calculation. With $(l_{DC}+1)$ DC points in a location, representing one operating point and l_{DC} remaining DC points, the matrices are calculated by:

$$\boldsymbol{A}_{loc} = mean(\boldsymbol{\Lambda}_1, \ldots, \boldsymbol{\Lambda}_{l_{DC}}, \boldsymbol{\Lambda}_{op}) \qquad \boldsymbol{B}_{loc} = mean(\tilde{\boldsymbol{B}}_{\lambda,1}, \ldots, \tilde{\boldsymbol{B}}_{\lambda,l_{DC}}, \tilde{\boldsymbol{B}}_{\lambda,op}) \tag{4.16}$$

$$\boldsymbol{F}_{loc} = mean(\boldsymbol{F}_{\lambda,1}, \ldots, \boldsymbol{F}_{\lambda,l_{DC}}, \boldsymbol{F}_{\lambda,op}) \qquad \boldsymbol{L}_{loc} = mean(\boldsymbol{L}_1, \ldots, \boldsymbol{L}_{l_{DC}}, \boldsymbol{L}_{op}) \tag{4.17}$$

The *weight method* adds weights in the calculation of the matrices. Three values can be weighted in a location: the operating point by w_1, the remaining DC points by w_2, and the remaining points by w_3, such that:

$$\sum_{i=1}^{3} w_i = 1 \tag{4.18}$$

With $l_{pts} = l_{loc} - (l_{DC}+1)$ remaining points belonging to location, each of the four representing matrix from Eqs. (4.16, 4.17) is computed by:

$$\boldsymbol{M}_{loc} = w_1 \cdot \boldsymbol{M}_{op} + w_2 \cdot mean(\boldsymbol{M}_{DC,1}, \ldots, \boldsymbol{M}_{DC,l_{DC}}) + w_3 \cdot mean(\boldsymbol{M}_{pts,1}, \ldots, \boldsymbol{M}_{pts,l_{pts}})$$

Where $\boldsymbol{M}$ can be replaced with the desired matrix and its values corresponding to the operating, DC, and remaining points, respectively.

Unless stated otherwise, the *mean method* is used as the default system description calculation method. The *op method* usually yields worse results, compared to the default method. In contrast, the *dc method* yields similar results to the *mean method*. However, upon changing the locations, the *mean method* results in smother transitions. The *weight method* can yield better results than the *mean method*, however, requires a good specification of the weights.

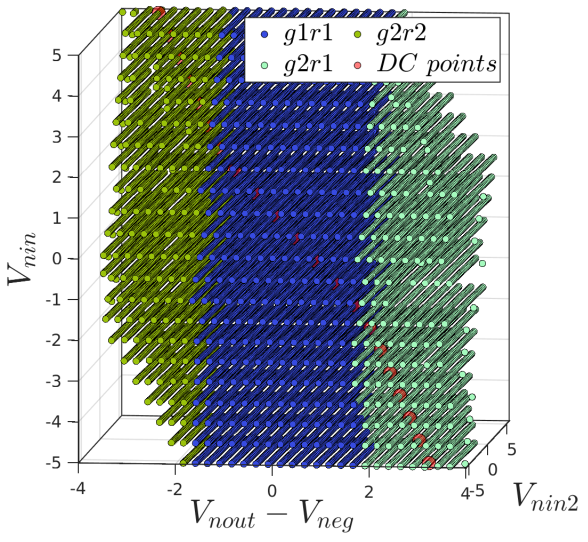
Before the system description specified by the matrices $\boldsymbol{A}_{loc}$ and $\boldsymbol{B}_{loc}$ and the back-transformation described by the transformation matrices $\boldsymbol{F}_{loc}$ and $\boldsymbol{L}_{loc}$ are calculated, the sampled points in each location can be filtered [1]. This filtering can remove points, according to a test with a specified tolerance margin, by classifying them as bad points. In this case, the obtained filtered data set can then be used for the calculation of the previous matrices, as well as for the proceeding calculations of the guards and invariants.

Filtering the Sampled Points

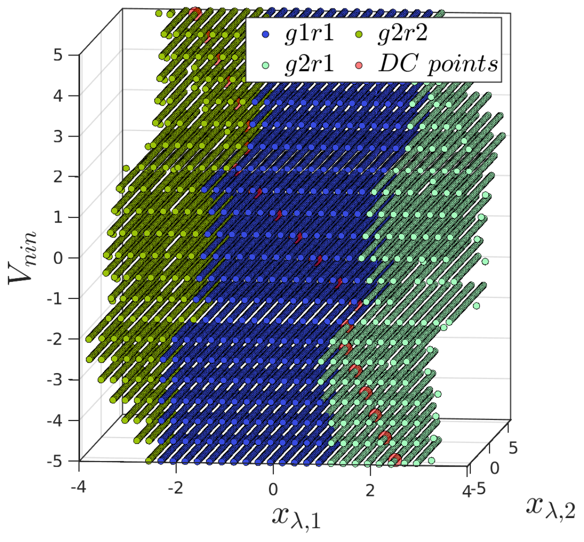
Consider Fig. 4.15a and Fig. 4.15b which present the $\mathcal{S}_{virt}$ and $\mathcal{S}_\lambda$ space drawn against the input voltage V_{nin} for the running example, respectively. For a specific input value V_{nin}, a plane containing one DC point can be observed in each space along the z-axis. This is due to the way *Vera* samples the state spaces. As stated in Chapter 3, and specifically in Algorithm 3 between lines 5 to 15, for a specific input value, *Vera* first computes a DC point (red points in Fig. 4.15). Starting from this point, *Vera* then steps though the $\mathcal{S}_\lambda$ with a specified state space step calculating thereby the surrounding points. From these points, and by taking again a state step, *Vera* continuous to sample the surrounding points till the specified borders are reached. Hence, *Vera* samples for each DC point a plane in the $\mathcal{S}_\lambda$ space. Moreover, *Vera* linearizes the system at every sampled point.

As stated earlier, when a linearization is performed at a specific point, the linearized system behavior is only valid in a range *close* to this linearization point. If for each plane in the $\mathcal{S}_\lambda$ space

[1]The results of *Vera* can sometimes contain non-plausible data resulting from the inverse integration scheme used in *Vera* or from numerical inaccuracies. These point are candidates for exclusion.



(a)



(b)

Fig. 4.15. The (a) $\mathcal{S}_{virt}$ and (b) $\mathcal{S}_\lambda$ space drawn against the input voltage V_{in}. As observed, for every input value there exists a plane containing one DC point.

the DC point is considered as the only linearization point, a question can be stated regarding the range of validity of the linearized model around these points. For this, consider again Eq. (4.9) which states the pointwise valid transformation between the $\mathcal{S}_o$ and $\mathcal{S}_\lambda$ space. When considering only one plane in the $\mathcal{S}_\lambda$ space obtained for a single input value $\boldsymbol{u}_{DC}$, the DC point $\boldsymbol{x}_{DC}$ in the $\mathcal{S}_o$ space and its corresponding DC point $\boldsymbol{x}_{\lambda,DC}$ in the $\mathcal{S}_\lambda$ space can be used as reference values for the entire plane. Thus, for every calculated DC points corresponding to a specific input value, the following equation is obtained by generalizing Eq. (4.9):

$$\boldsymbol{F}_\lambda \boldsymbol{x}_\lambda = (\boldsymbol{x} - \boldsymbol{x}_{DC}) + \boldsymbol{F}_\infty \boldsymbol{H}_\infty \boldsymbol{B}_{\infty,red}(\boldsymbol{u} - \boldsymbol{u}_{DC}) + \boldsymbol{F}_\lambda \boldsymbol{x}_{\lambda,DC} \tag{4.19}$$

Note that the matrices in Eq. (4.19) in general vary for every sampled point. *Vera* calculated and corrected these matrices between every sampled point and the next points as given by the connection graph. Obviously when considering the DC points as a constant references, only the points that are a state step away from the DC points and have valid connection according to the connection graph obey Eq. (4.19). For the remaining points of the plane, Eq. (4.19) proposes an estimation. Still, a question can be stated regarding which transformation matrices should be used in Eq. (4.19): those of the DC point of a plane or those of the currently inspected point. Therefore, there are two options available to proceed with the abstraction of Eq. (4.9) through Eq. (4.19):

transPt method choose the transformation matrices as those belonging to the current analyzed sample point. Thus, an error is performed for every considered point further that one state step away from the DC point of the current plane

transDC method choose the transformation matrices belonging to the current DC point of the plane, ignoring thereby the transformation matrices of the current analyzed point. Hence, an error is made for every analyzed point that has transformation matrices that differ from those of the current used DC point, as well as for those points that are further than one step away from the DC point

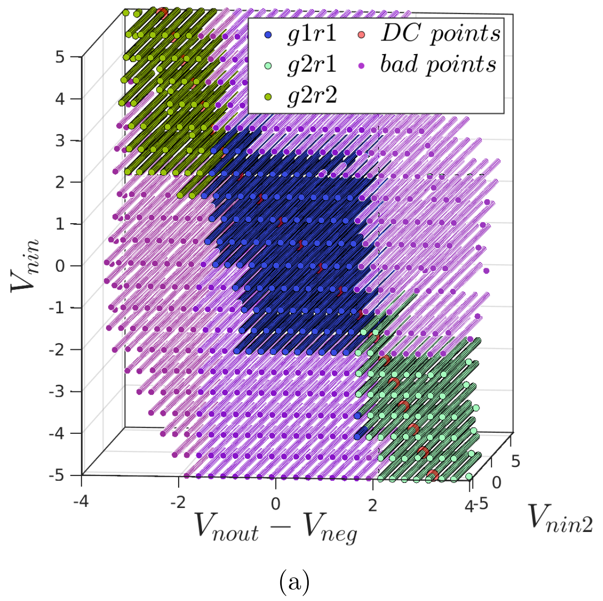
The *transPt method* provides additional information containing the variation of the transformation matrices between the points in a plane. As the *transDC method* injects additional errors into the modeling process, this method in not handled. For every $\boldsymbol{x}_\lambda$ belonging to the plane containing the DC point, Eq. (4.19) is solved for $\boldsymbol{x}_{\lambda,lsq}$ using the least-square method:

$$\boldsymbol{x}_{\lambda,lsq} = lsq(\boldsymbol{F}_\lambda, (\boldsymbol{x} - \boldsymbol{x}_{DC}) + \boldsymbol{F}_\infty \boldsymbol{H}_\infty \boldsymbol{B}_{\infty,red}(\boldsymbol{u} - \boldsymbol{u}_{DC})) + \boldsymbol{x}_{\lambda,DC} \tag{4.20}$$

The obtained result is compared to the $\boldsymbol{x}_\lambda$ value calculated by *Vera*, thereby marking points that have large deviations. This is done by calculating the error δ_{lsq} in a similar fashion to the root mean square error for every sampled point using the reduced order m and the Euclidean distance:

$$\delta_{lsq} = \frac{||\, \boldsymbol{x}_\lambda - \boldsymbol{x}_{\lambda,lsq}\, ||_2}{\sqrt{m}}, \tag{4.21}$$

For the running example, this comparison is performed by marking points which have an error $\delta_{lsq} \geq 0.1$. The result is illustrated in Fig. 4.16. As observed, several points are marked as bad points and can be dropped out if desired for the preceding calculations.



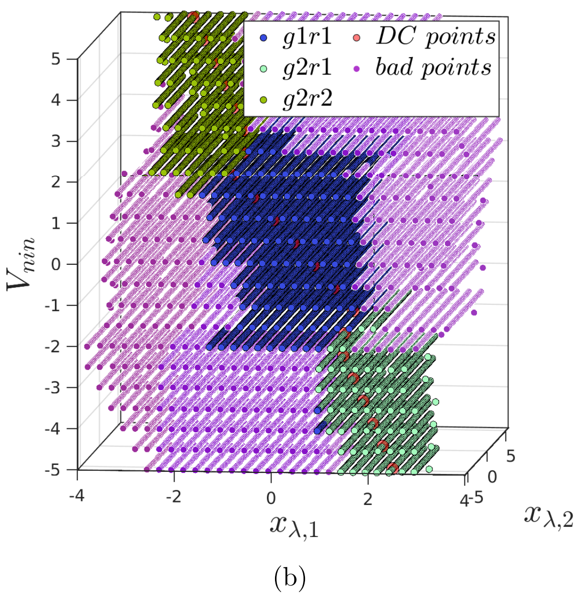


Fig. 4.16. Identifying points that yield a great linearization error if considered with respect to the DC point at a given input voltage.

Hence, the technique of filtering the sampled point based on Eq. (4.20) and a specified error margin δ_{lsq} (Eq. (4.21)) is referred to as the *lsq filter*. A formal definition is:

lsq filter filters the sampled points according to a specified error margin δ_{lsq} by considering for each DC point a plane in the $\mathcal{S}_\lambda$ or $\mathcal{S}_{virt}$ space, and applying Eqs. (4.20, 4.21) along using the *transPt method* or *transDC method* on all sampled points belonging to this plane

This filter can be seen as the removing points that are not suitable to be considered from the DC points, even though their transformation matrices are used (*transPt method*). Note that, the filter can be applied for removing points during the calculation of the matrices $\boldsymbol{A}_{loc}$, $\boldsymbol{B}_{loc}$, $\boldsymbol{F}_{loc}$ and $\boldsymbol{L}_{loc}$, as well as for removing points for further calculations such as performed in Section 4.5.5. For the running example the effect of applying *lsq filter* is demonstrated on the calculation of the transformation matrices. For the original system with $n = 24$ variables stated in Eq. (4.1) and a reduced order of $m = 2$, the transformation matrix $\boldsymbol{F}_{loc} \in \mathbb{R}^{24\times2}$ has two columns. Considering only the 21^{th} row, which corresponds to the output of the system V_{nout}, the distribution of $\boldsymbol{F}_{loc}$ is illustrated in Fig. 4.17 for the three previous identified location $g1r1$, $g2r1$, and $g2r2$. The first row of this figure presents the first column of $\boldsymbol{F}_{loc}$, while the second row of Fig. 4.17 represent the second column of $\boldsymbol{F}_{loc}$. The values of $\boldsymbol{F}_{loc}$ at the operating point are marked in red (*op method*), while the calculated mean values of $\boldsymbol{F}_{loc}$ are marked in green when all sampled points are used (*mean method*), and in magenta when only the DC points are used (*dc method*).

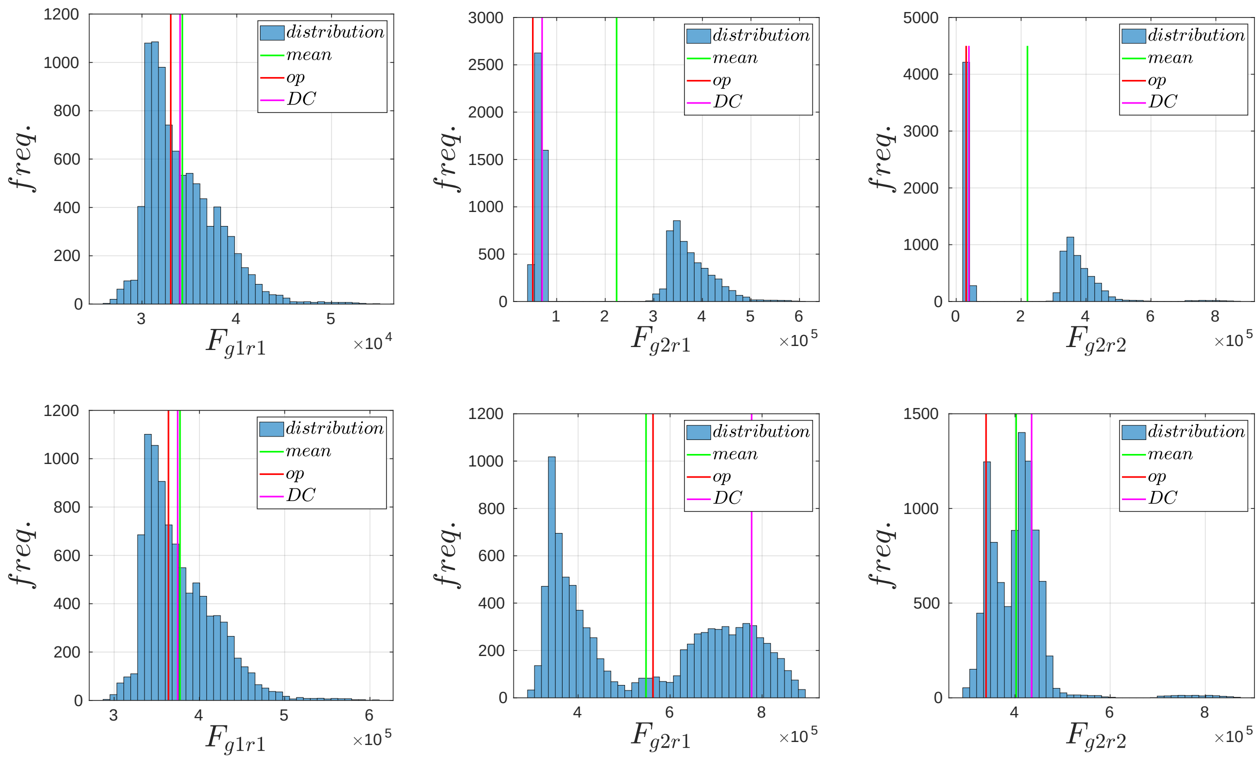


Fig. 4.17. Distribution of the transformation matrix $\boldsymbol{F}_{loc}$ for the locations $loc \in \{g1r1, g2r1, g2r2\}$ without the application of the *lsq filter*. The results are illustrated upon the 21^{th} row of $\boldsymbol{F}_{loc}$. The first row of this figure represents the first column of $\boldsymbol{F}_{loc}$, while the second row represents the second column of $\boldsymbol{F}_{loc}$.

As observed in Fig. 4.17, in the limiting locations of the system ($g2r1$ and $g2r2$) the transformation matrix $\boldsymbol{F}_{loc}$ attains various values with large differences. This can be traced back to the sampling performed by *Vera*, and the methodology used to calculation transformation matrix $\boldsymbol{F}_{loc}$. Nonetheless, selecting the correct representing value out of this large range is a challenging task.

When considering $g1r1$ and the corresponding first column in Fig. 4.17, one can notice that there exists a difference between the value of $\boldsymbol{F}_{loc}$ at the operating point (red line) and the mean value of this matrix computed with the DC points only (magenta line). Even though $g1r1$ is the linear location of the system, a difference is still presented. This difference becomes even larger in the nonlinear location ($g2r1$ and $g2r2$) especially in the second dimension of $\boldsymbol{F}_{loc}$, as illustrated in the second row of Fig. 4.17.

With an increasing number of locations, the values shown in Fig. 4.17 are distributed on more locations. This is often accompanied by a shortage of the distance between the operating and mean values (red and green lines). More precisely, better results are obtained with an increasing number of locations as will be later seen in Section 6.1.

When the *lsq filter* is applied during the calculation of the matrices, the mean value calculated (green line) changes as well as the number of points that are considered. In Fig. 4.18, the result of calculating the transformation matrix $\boldsymbol{F}_{loc}$ is illustrated for the row corresponding to the output node V_{nout} after applying the *lsq filter*. Note that, the red and magenta lines do not change. The number of bins in both figures, Fig. 4.17 and Fig. 4.18, is set to 40. The points dropped out from Fig. 4.18 compared to Fig. 4.17, are illustrated in Fig. 4.16 as bad points.

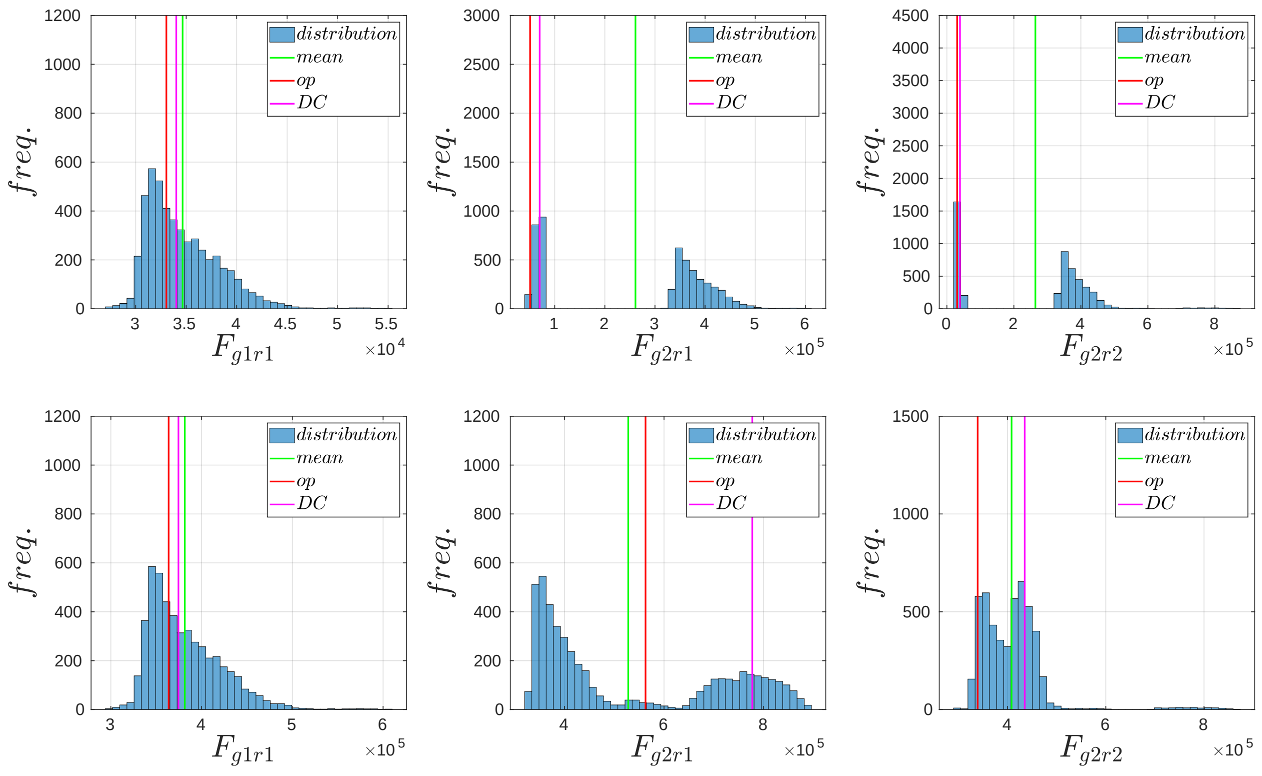
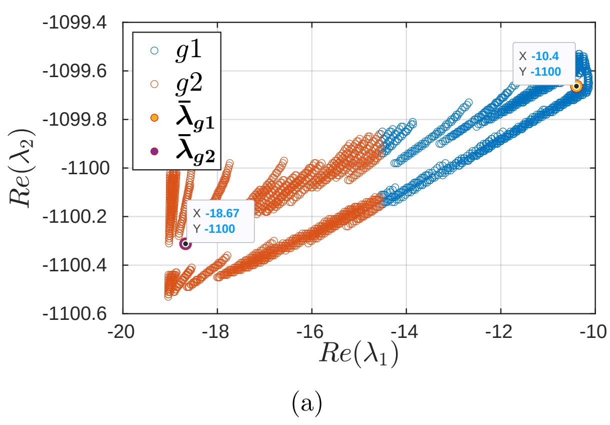


Fig. 4.18. Distribution of the transformation matrix $\boldsymbol{F}_{loc}$ as described in Fig. 4.17 after using the *lsq filter* according to Section 4.5.2.

Even after applying the *lsq filter*, the obtained results only slightly change for the running example. More precisely, the green line representing the mean value of $\boldsymbol{F}_{loc}$ changes only minimally upon comparing Fig. 4.18 to Fig. 4.17.

As the experimental results later in Chapter 7 show, using the *mean method* often yields better results especially with a smaller number of locations. Additionally, in Section 7.1.3 several models have been generated with the four different system descriptions handled in this section. As the results show, both the *mean method* and the *dc method* yield the best results. However, during the transitioning of the locations, the *mean method* yields better results. Moreover, Section 7.1.3 shows as well that using the *lsq filter* for this example does not improve the system description, but rather worsen the obtained results.

On the other hand, the eigenvalues with and without the application of the *lsq filter* are illustrated in Fig. 4.19b and Fig. 4.19a respectively. Even though the number of eigenvalues decreased, the mean of the eigenvalues did nearly not change.



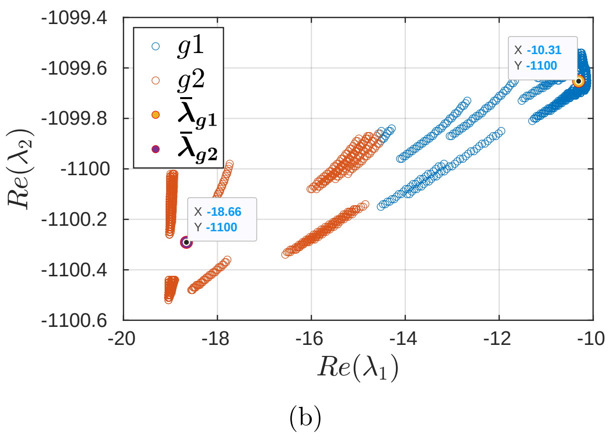


Fig. 4.19. In (a) the eigenvalues as sampled by *Vera* are illustrated, while in (b) the eigenvalues are shown after using the *lsq filter* described in Section 4.5.2.

In what proceeds in this chapter, the *lsq filter* is not applied for the calculation of the previous stated matrices, but only for filtering the bad points (see Fig. 4.16). In case the sampling performed by *Vera* yields a $\mathcal{S}_\lambda$ space with strongly overlapping regions, it is recommended to use the *lsq filter* to obtain better results.

4.5.3 Sorting the Locations

The third block shown in Fig. 4.12, sorts the identified locations. Based on the operating points $\boldsymbol{x}_{\lambda,op}$ and $\boldsymbol{x}_{virt,op}$ found previously in Section 4.5.1, the locations can be sorted. Starting with the center location (Section 4.5.1), the locations are sorted from the closest to the furthest according to the distance between their operating point to the operating point of the center location in the $\mathcal{S}_{virt}$ space. Moreover, each location is provided with an information regarding the surrounding locations. This information lists the neighbor locations from the closest neighbor to the furthest surrounding one. This information is necessary as it can speed-up the guard identification later performed in Section 4.5.6.

4.5.4 Modeling the Input as a State

The fourth block from Fig. 4.12 can be used to apply adjustments to the modeling of the system. This includes modeling the inputs of the system as states. As indicated in Fig. 4.12, this block is optional and often skipped. However, in some cases, especially when working with Matlab models that are later used with the reachability tool *Cora*, this modeling step is sometimes necessary. At the current time, the guards in *Cora* can only be defined using the states of the system, which are the contents of the vector $\boldsymbol{x}_\lambda$ for the current models. On the other hand, in some cases it is desired to include the input in the guard conditions.

To achieve this, several steps are necessary. First, k_u specified inputs $\boldsymbol{u}_+ \in \mathbb{R}^{k_u}$ are added as additional dimensions to the $\mathcal{S}_\lambda$ as well as to $\mathcal{S}_{virt}$ spaces, and removed from the input space $\mathcal{S}_u$. Thus, the system has now $k - k_u$ inputs and $m + k_u$ state variables in the $\mathcal{S}_\lambda$ space as well as in the $\mathcal{S}_{virt}$ space, with the new state variables:

$$\boldsymbol{x}_{\lambda+} = \begin{bmatrix} \boldsymbol{x}_\lambda \\ \boldsymbol{u}_+ \end{bmatrix} \qquad \boldsymbol{x}_{virt+} = \begin{bmatrix} \boldsymbol{x}_{virt} \\ \boldsymbol{u}_+ \end{bmatrix} \tag{4.22}$$

Second, all calculations including the location identification (Section 4.4), operating point calculation (Section 4.5.1), and system modeling (Section 4.5.2) handled previously, additional to the upcoming topics including the guards and invariants identification (Section 4.5.6) are executed in the extended $\mathcal{S}_{\lambda+}$ and $\mathcal{S}_{virt+}$ spaces and the shrunk input space $\mathcal{S}_u$. On the other hand, the calculations performed in Section 4.5.5 are still executed in the $\mathcal{S}_\lambda$ and $\mathcal{S}_{virt}$ spaces. The results are then extended as shown in Eq. (4.22).

Third, for each location the system matrix $\boldsymbol{A}_{loc}$ from Eq. (4.10) is extend by $\boldsymbol{B}_+$, the part of the input matrix $\boldsymbol{B}_{loc}$ that corresponds to $\boldsymbol{u}_+$:

$$\boldsymbol{A}_{loc+} = \left[\begin{array}{c:c} \boldsymbol{A}_{loc} & \boldsymbol{0} \\ \hdashline \boldsymbol{0} & \boldsymbol{B}_+ \end{array}\right] \tag{4.23}$$

Finally, the rows from $\boldsymbol{F}_\infty \boldsymbol{H}_\infty \boldsymbol{B}_{\infty,red}$ corresponding to the selected inputs are transferred to extend the transformation matrix $\boldsymbol{F}_\lambda$ by additional columns. This corresponds to removing $\boldsymbol{L}_+$ from $\boldsymbol{L}_{loc}$ and adding it to $\boldsymbol{F}_{loc}$, thereby defining $\boldsymbol{F}_{loc+}$ as:

$$\boldsymbol{F}_{loc+} = \left[\begin{array}{c:c} \boldsymbol{F}_{loc} & \boldsymbol{L}_+ \end{array}\right] \tag{4.24}$$

In order to illustrate this option, an example of a circuit with a diode is modeled with the input as a state in Section 7.2.

4.5.5 $\mathcal{S}_\lambda$ Space Recalculation

Continuing the modeling process of the system according to Fig. 4.12, the next block involves recalculating the $\mathcal{S}_\lambda$ space. Note that this block can be skipped as the $\mathcal{S}_\lambda$ space was already calculated by *Vera*. However, as will be clear at the end of this section, there are several reasons why the $\mathcal{S}_\lambda$ space should be recalculated.

Finding the $\mathcal{S}_\lambda$ space corresponds to solving Eq. (4.9) for $\boldsymbol{x}_\lambda$. The problem with this equation is that it is pointwise valid; it is only valid for each pair of points in the state space that are a state step apart. Considering the aim which is to abstract the system, Eq. (4.9) needs to be abstracted.

In Section 4.5.2, Eq. (4.19) was presented which abstracts Eq. (4.9) by using the transformation matrices belonging to either the DC point (*transDC method*) or the current analyzed point (*transPt method*) along with the DC points to calculate the remaining points belonging to the plane in the $\mathcal{S}_\lambda$ domain corresponding to the same input voltage. The recalculated space in Section 4.5.2 was only used to the filter (mark) bad points that were excluded in specific calculations. However, the current task is to abstract Eq. (4.9) even more than Eq. (4.19) did, thereby replacing the $\mathcal{S}_\lambda$ space calculated by *Vera*.

For that, two methods have been implemented that discard the $\mathcal{S}_\lambda$ space calculated by *Vera* and replace it by values found by their solutions. The first method, the *transPt method*, has already been introduced in Section 4.5.2. If the *transPt method* is used to recalculate the $\mathcal{S}_\lambda$ space, Eq. (4.20) is used along with the transformation matrices of the current inspected point. This time, the obtained $\mathcal{S}_\lambda$ space through this method replaces the $\mathcal{S}_\lambda$ space calculated by *Vera*. The second method that can be used to recalculate the $\mathcal{S}_\lambda$ space is the *transLoc method*:

transLoc method uses the operating point at each location as a reference point along with the transformation matrices previously calculated in Section 4.5.2

With the transformation matrices $\mathbf{F}_{loc}$ and $\boldsymbol{L}_{loc}$ at hand, the *transLoc method* further abstracts Eq. (4.19) to:

$$\boldsymbol{F}_{loc}\boldsymbol{x}_\lambda = \boldsymbol{x} - \boldsymbol{x}_{op} + \boldsymbol{L}_{loc}(\boldsymbol{u} - \boldsymbol{u}_{op}) + \boldsymbol{F}_{loc}\boldsymbol{x}_{\lambda,op} \tag{4.25}$$

Hence, the $\mathcal{S}_\lambda$ space is recalculated by solving Eq. (4.25) for all sampled points belonging to a location $loc \in Loc$:

$$\boldsymbol{x}_\lambda = lsq(\boldsymbol{F}_{loc}, (\boldsymbol{x} - \boldsymbol{x}_{op}) + \boldsymbol{L}_{loc}(\boldsymbol{u} - \boldsymbol{u}_{op})) + \boldsymbol{x}_{\lambda,op} \tag{4.26}$$

Eq. (4.26) recalculates the points in the $\mathcal{S}_\lambda$ space of a location to fit a single transformation described by $\boldsymbol{F}_{loc}$ and $\boldsymbol{L}_{loc}$ (see Section 4.5.2) and the found operating points $\boldsymbol{x}_{op}$, $\boldsymbol{x}_{\lambda,op}$ at the operating input $\boldsymbol{u}_{op}$ (Section 4.5.1) for each location. For the rest of this section, only the *transLoc method* will be considered.

This method can inject errors into the guard and invariant as points in the state space can be calculated that have different values in the $\mathcal{S}_\lambda$ space than those calculated by *Vera*. Looking at this fact from a different perspective, the points in the $\mathcal{S}_\lambda$ space are modified to obey the back-transformation given by Eq. (4.11). Note that, both stated methods in this section can be combined with the *lsq filter* from Section 4.5.2.

In Fig. 4.20 a comparison between the $\mathcal{S}_\lambda$ space from *Vera* and the recalculated $\mathcal{S}_\lambda$ space using Eq. (4.26) is illustrated. Note that in both figures the *lsq filter* was not applied. Fig. 4.20a and Fig. 4.20c correspond to the unchanged $\mathcal{S}_\lambda$ space as calculated by *Vera*, on the other hand, Fig. 4.20b and Fig. 4.20d show the recalculated $\mathcal{S}_\lambda$ space. In the first row of Fig. 4.20, the input was taken as an additional dimension to remove the overlapping of the points. As shown later in Section 7.1, all these variants do not yield good abstracting models in general.

In contrast, when in either case the *lsq filter* is additionally used, the resulted models exhibit a better abstraction behavior. The corresponding $\mathcal{S}_\lambda$ spaces are illustrated in Fig. 4.21.

As stated previously, the $\mathcal{S}_\lambda$ recalculation methods; the *transPt method* and the *transLoc method*, can be both accompanied by the *lsq filter* from Section 4.5.2. In this case, the bad sampled points (see Fig. 4.16) are removed from the $\mathcal{S}_\lambda$ as observed in Fig. 4.21, prior to recalculating this space with either method. Note that, Fig. 4.21b and Fig. 4.21d were generated using the *transLoc method*.

As shown in Fig. 4.12, the recalculation of the $\mathcal{S}_\lambda$ is optionally. However, there are several reasons why the $\mathcal{S}_\lambda$ space should be recalculated:

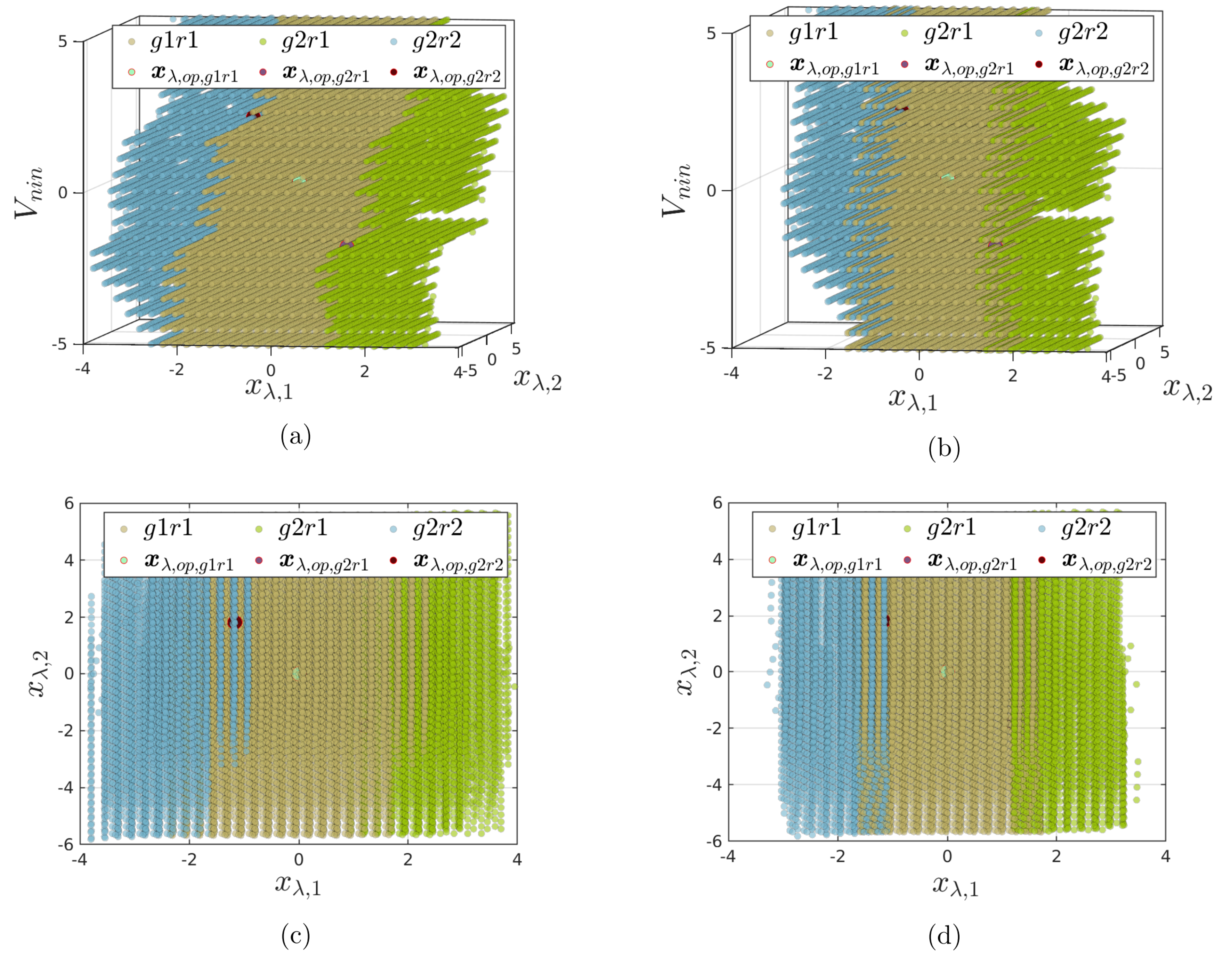


Fig. 4.20. In (a) and (c) the unmodified $\mathcal{S}_\lambda$ space by *Vera* is presented, while in (b) and (d) the recalculated $\mathcal{S}_\lambda$ using the *transLoc method* (Eq. (4.26)) is illustrated. In the first row, the input V_{nin} is taken as an additional dimension.

1. The HA has linear location described as stated in Eq. (4.10). These delta values are computed with respected to the operating points. When the system is later simulated in the $\mathcal{S}_\lambda$ space, the locations in the $\mathcal{S}_\lambda$ should be adjusted to fit the behavior specified by the calculated matrices (Section 4.5.2) and the identified operating points.
2. Points in a location should have similar transformation matrices. Points which have strongly varying transformation matrices should not belong to the same location.
3. The results of the guards and invariants identification performed later in Section 4.5.6 can be improved by reforming the $\mathcal{S}_\lambda$ space.
4. To control the discontinuities produced by the jump functions as will be later examined in Section 4.5.9.
5. We do not completely trust the calculation performed by *Vera*.

Especially the second point makes is clear why the *lsq filter* should always be used. When the $\mathcal{S}_\lambda$ space is recalculated and this filter is active, part of the behavior calculated by *Vera* is preserved and can be bounded to an error margin. In Section 7.1 varies models were created with different

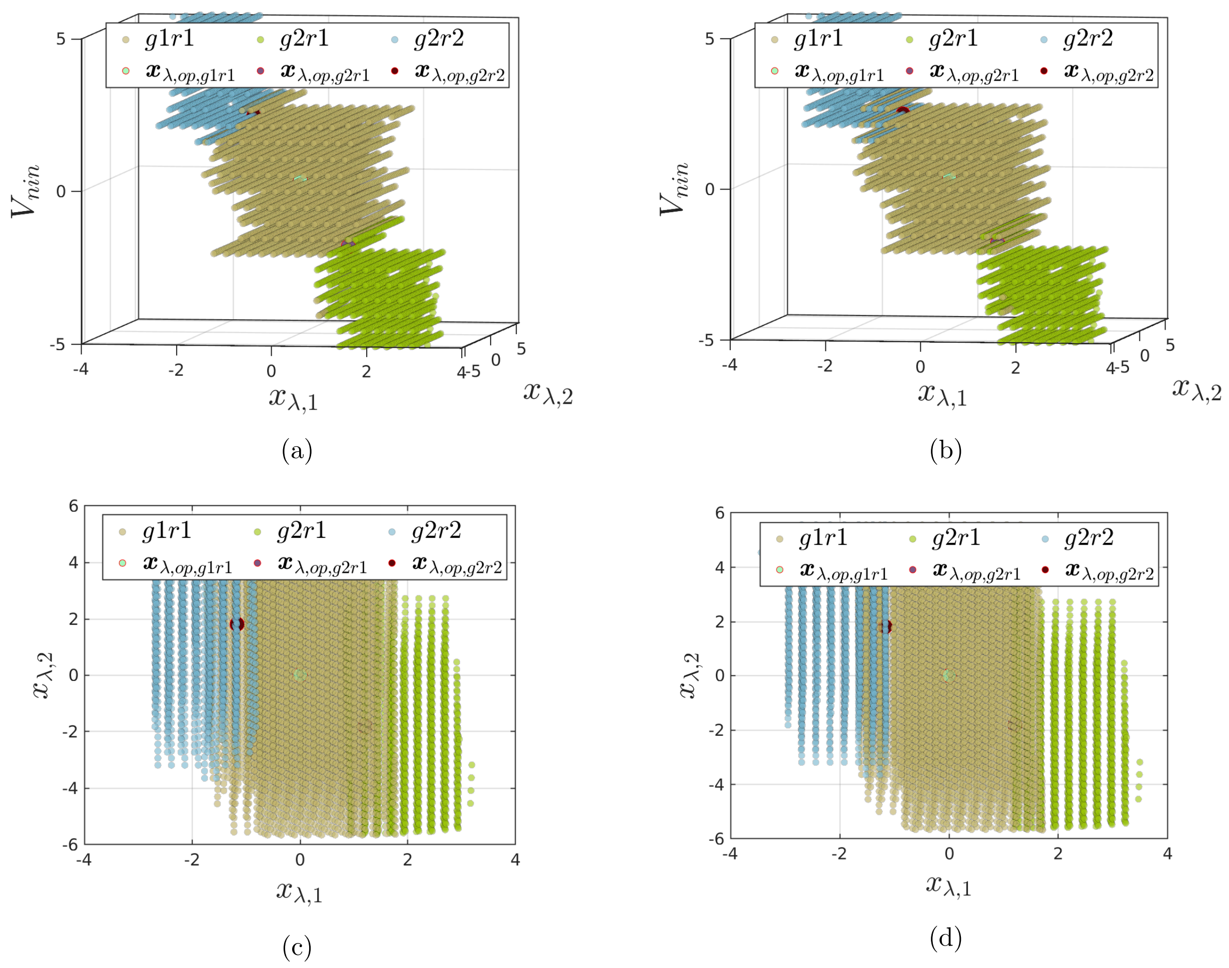


Fig. 4.21. $\mathcal{S}_\lambda$ space after applying the *lsq filter* on the $\mathcal{S}_\lambda$ space from *Vera* ((a) and (c)) and on the recalculated $\mathcal{S}_\lambda$ space ((b) and (d)) using the *transLoc method*. In the first row, the input V_{nin} is taken as an additional dimension.

methods, and indeed, the combination of these two methods yields the best results in case the $\mathcal{S}_\lambda$ space is strongly overlapping.

Summing up all previously mentioned manipulations and adjustments from Chapter 3 till the current chapter, there are four options that can be used to filter and reform the $\mathcal{S}_\lambda$ space:

unchanged	the $\mathcal{S}_\lambda$ space is used as calculated by *Vera* with all sampled points
reach filter	the $\mathcal{S}_\lambda$ space is defined only through the points marked as reachable by *Vera* as stated in Section 3.2.4
lsq filter	removes points from the $\mathcal{S}_\lambda$ space which cannot be calculated from the DC points under a specified error, even though their transformation matrices are used and not those of the DC points (*transPt method*). For consistency, the points removed from the $\mathcal{S}_\lambda$ space are also removed from the $\mathcal{S}_{virt}$ space (Section 4.5.2)
$\mathcal{S}_\lambda$ *recalculation*	discards the $\mathcal{S}_\lambda$ space calculated by *Vera* and recalculates it either by the *transLoc method* which uses the computed transformation matrices $\boldsymbol{F}_{loc}$ and

$\boldsymbol{L}_{loc}$ with Eq. (4.26) and the identified operating points for each location, or by the *transPt method* which uses the transformation matrices $\boldsymbol{F}_\lambda$ and $\boldsymbol{F}_\infty \boldsymbol{H}_\infty \boldsymbol{B}_{\infty,red}$ of the current inspected point with Eq. (4.20) and the DC points with the same input voltage. In both cases the solution is computed with the least-square method. By default, the *transLoc method* is used

Combinations between the filters and recalculations methods are also possible and occur in the ordered stated above. For example, one might apply first the *reach* filter followed by the *lsq filter*. An overview of these manipulations is shown in Fig. 4.22.

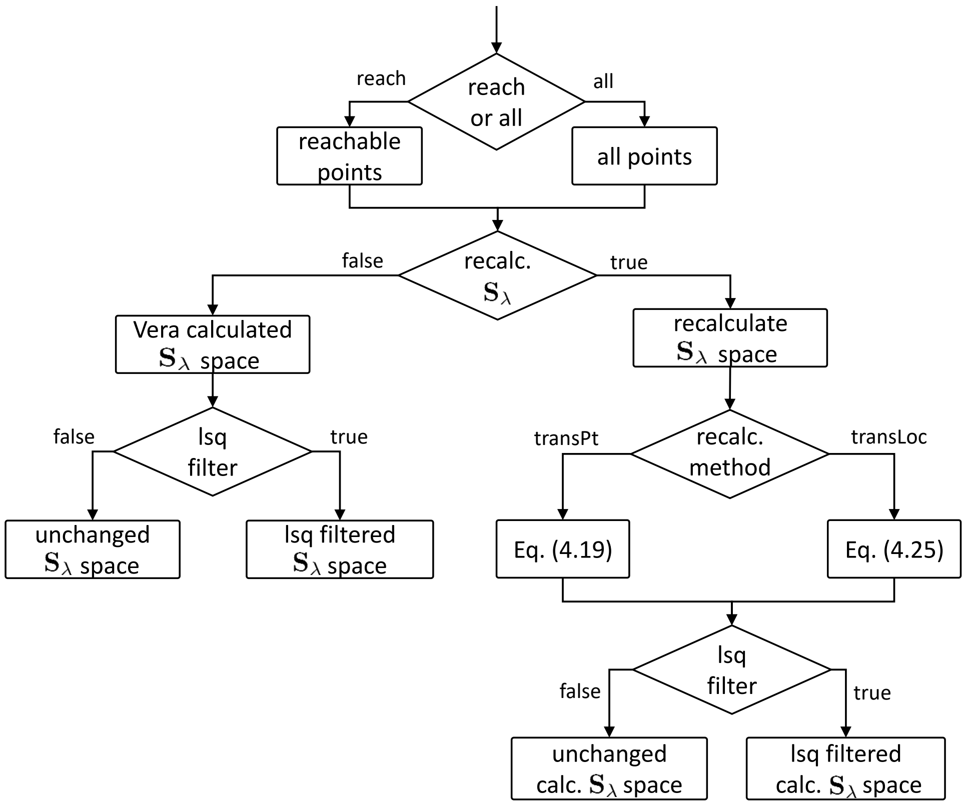

reach
reach or all
all
reachable points
all points
false
recalc. $\mathbf{S}_\lambda$
true
Vera calculated $\mathbf{S}_\lambda$ space
recalculate $\mathbf{S}_\lambda$ space
false
lsq filter
true
transPt
recalc. method
transLoc
unchanged $\mathbf{S}_\lambda$ space
lsq filtered $\mathbf{S}_\lambda$ space
Eq. (4.19)
Eq. (4.25)
lsq filter
false
true
unchanged calc. $\mathbf{S}_\lambda$ space
lsq filtered calc. $\mathbf{S}_\lambda$ space

Fig. 4.22. Overview of the manipulations and adjustments in the $\mathcal{S}_\lambda$ space.

The first decision is to choose either all sampled points or only the reachable marked ones. In the later case, the size of the sampled data set decreases, as points which are marked as not reachable are completely neglected in the further analysis. The second decision involves choosing the $\mathcal{S}_\lambda$ space calculated by *Vera* or recalculating this space using the stated methods. In both cases, the points used can be filtered priory using the *lsq filter* as discussed in Section 4.5.2. Taking into account the two different $\mathcal{S}_\lambda$ space recalculation methods that can be used, there are in total 12 possible scenarios to find the $\mathcal{S}_\lambda$ space. Note that, using the reachable sampled points with the *transPt method* and the *lsq filter* yields the best results if the $\mathcal{S}_\lambda$ space has strongly overlapping locations.

4.5.6 Finding the Invariants

With the $\mathcal{S}_{virt}$ and $\mathcal{S}_\lambda$ spaces at hand, either calculated or obtained from *Vera*, along with the locations of the HA, the guards and invariants are next identified in the 6th block of Fig. 4.12.

As the HA is later simulated in the $\mathcal{S}_\lambda$ space, it is necessary to find the guards and invariants in this space. According to Section 4.5, the identification of the guards and invariants can occur either in the $\mathcal{S}_\lambda$ or $\mathcal{S}_{virt}$ space. The latter case is illustrated in Fig. 4.12 and used in this chapter.

Finding the guards and invariants in the $\mathcal{S}_{virt}$ space brings the advantage of an easier identification as the locations often do not overlap at all. On the other hand, the found results in the $\mathcal{S}_{virt}$ space must be transformed back into the $\mathcal{S}_\lambda$ space, which is not an easy task and error-prone. As previously seen in Section 4.5.5, this lies in the fact that the locations in the $\mathcal{S}_\lambda$ can sometimes overlap, which can be traced back to the options provided to *Vera*, influencing the sampling of the netlist. Specifically, these options can shift the individual calculated DC points by specific values, for example, in accordance with the DC points in the $\mathcal{S}_{virt}$ space, thereby minimizing the overlapping of the locations. As will be clear at the end of this section, the overlapping of the locations does not affect the identification of the invariants, but only the calculation of the guards.

Based on the sampled points belonging to a location, the algorithm first finds the invariant of each location. With the found invariants, the guards between the locations are identified. In the following all calculations are performed in the $\mathcal{S}_{virt}$ space. Note that as stated in Section 4.5.5, the *lsq filter* applied to the $\mathcal{S}_\lambda$ space also removes points from the $\mathcal{S}_{virt}$ space. Thus, the filtered $\mathcal{S}_{virt}$ space shown in Fig. 4.16 is used here.

An invariant of a location is found by hulling the points belonging to this location by one of the following geometric shapes:

- polytope (Section 2.1.1)
- zonotope (Section 2.1.2)
- interval hull (Section 2.1.3)

Regardless of the geometric shape used, throughout this dissertation, an invariant of a location $loc \in Loc$ is denoted as inv_{loc} according to Definition 1 from Section 2.4.

Several steps are required to find the invariant of a location. First, the borders of the locations are found using convex hulls. The convex hulls are formed using the algorithm described in [BDH96]. This yields for each location the vertices $\boldsymbol{v}_{loc}$, as shown in Fig. 4.23 for the running example.

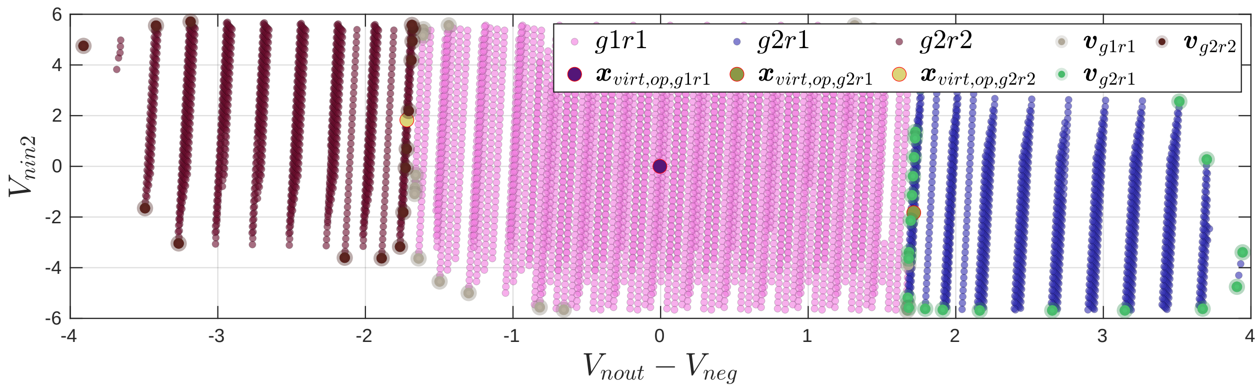
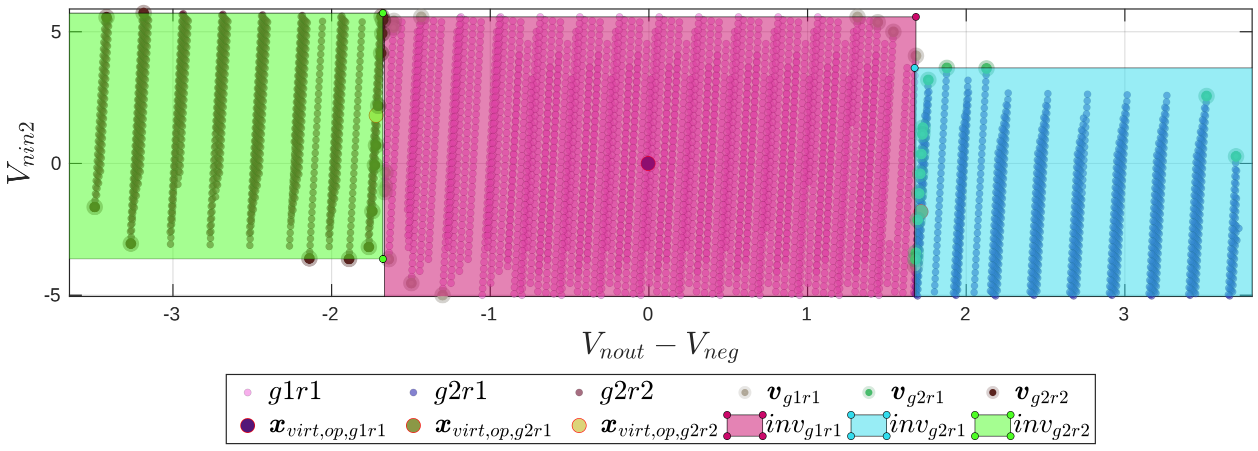
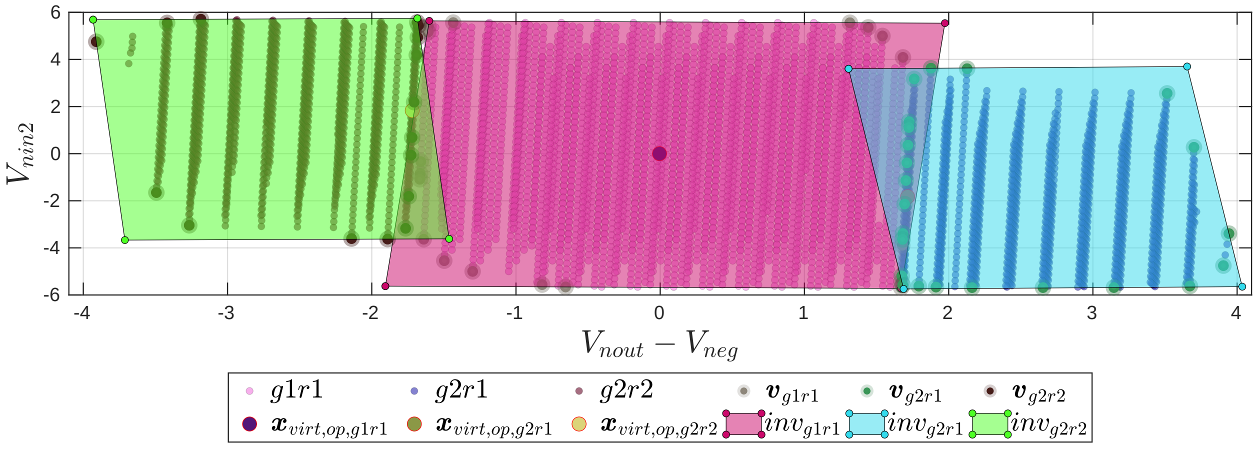


Fig. 4.23. Identifying the borders of the locations in the $\mathcal{S}_{virt}$ space using convex hulls. The vertices ($\boldsymbol{v}_{loc}$) of the convex hulls are subscripted according to the corresponding location.

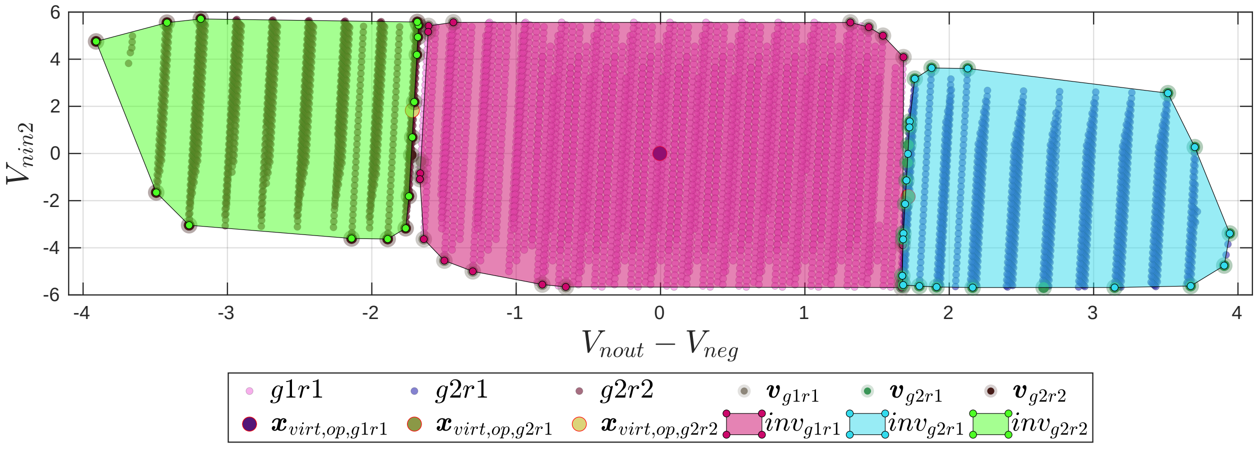
The found vertices are used to model the invariants of the locations with the desired geometric representation according to Section 2.1. For the running example, the possible invariant representations are illustrated in Fig. 4.24.



(a) Invariants modeled as interval hulls.



(b) Invariants modeled as zonotopes.



(c) Invariants modeled as polytopes.

Fig. 4.24. Modeling the invariants of the locations with different geometric shapes.

Representing an invariant as an interval hull basically corresponds to finding the bounding box of the vertices as shown in Fig. 4.24a. This modeling results in large over approximations as observed. When zonotopes are used to model the invariants as shown in Fig. 4.24b, the invariants are over approximated as well. More precisely, in some cases this representation can yield tighter invariants compared to interval hulls. Nonetheless, this strongly depends on the order of the zonotope (number of generators), as well as on the distribution of the points belonging to a location in the $\mathcal{S}_{virt}$ space. Moreover, when comparing invariants modeled by zonotopes to the same invariants modeled with either polytopes or interval hulls, the resulting invariants are no longer bounded by the vertices i.e. the zonotopes can be larger than the *max/min* of all vertices. Depending on the scenario, this representation can be less or more over approximative than using interval hulls, but is usually over approximative when compared to polytopes. Fig. 4.24c shows the most accurate representation of the invariants, which is obtained by enclosing the vertices of the convex hulls by polytopes. Even this representation can yield over approximation of the space spanned by the sampled points, for the case this portion of the space is not convex.

Since this identification was performed in the $\mathcal{S}_{virt}$ space, the found invariants must be transformed into the $\mathcal{S}_{\lambda}$ space (Section 4.5.8). Note that, if the $\mathcal{S}_{\lambda}$ space was used for the identification of the invariants, the approach would be similar to the stated identification in the $\mathcal{S}_{virt}$, with the difference that the results could be directly used.

In general, as observed in the figures of this section, a well partitioned state space (in this case $\mathcal{S}_{virt}$), yields good invariants. Since the sampled points are distributed among the locations based on the location identification performed in Section 4.4, and the invariants are found by hulling these points, the result of the invariants identification strongly depends on the location identification, which labeled the points according to the linearized system behavior. Moreover, this identification also depends on the distribution of the points in a location, which influence the calculation of the convex hulls, which in terms affect the calculations of the geometrics shape used to enclose the vertices of the convex hulls. On the other hand, the identification of the invariants is not affected by the overlapping of the locations, as during this identification each location is handled independently from the others, in contrast to the identification of the guards which is handled next.

4.5.7 Finding the Guards

According to Definition 1, guards are denoted throughout this dissertation as stated in Eq. (2.40). For example, if the current location is $g1r1$ and the target location is $g2r2$, then the h^{th} guard of location $g1r1$ that allows for a transition to the target location $g2r2$ is denoted as:

$$grd_h : \underbrace{g1r1}_{\text{current } loc} \rightarrow \underbrace{g2r2}_{\text{target } loc} \tag{4.27}$$

According to Section 2.4, guards define when a location transition occurs. There are in general four types of guard representations:

- polytope (Section 2.1.1)
- zonotope (Section 2.1.2)
- interval hull (Section 2.1.3)
- halfspace (Eq. (2.1))

For the SystemC-AMS as well as for the Verilog-A models, the definition for the guards is slightly changed compared to Section 2.4. For these two models, a transition always and immediately occurs when a guard is hit. Hence, it makes sense to define the corresponding guards as halfspaces. On the other hand, for the Matlab models, *Cora* can handle guards in various representation. In case a reachable set intersects a guard, *Cora* projects the contained reachable set into the next location if the polytope method is used for the guard intersections[ASB10a]. This method will be used throughout this dissertation for guards of types polytopes, zonotopes, or interval hulls. *Cora* is also able to handle halfspace guards, however, a different guard intersection methodology is used which considers only a single reachable set (hyperplaneMap) [AK12b]. Note that, the remaining guard intersection methods and guard representations of *Cora* will not be considered here.

In any case, the guards are determined by the points spanning the corresponding shapes. Hence, first the guard points are determined, followed by representing these point with the specific guard representation. For guards modeled as polytopes, zonotopes, or interval hulls, the guard points are hulled by the specified geometric shape analogously to the approach from Section 4.5.6. For halfspace guards, the guard points are used to determined the equation of the halfspace (see Eq. (2.1)).

Based on the invariants the guard points are first identified. To identify the points of the guards two methods were implemented:

distance method	uses distance metrics to identify the guards (Section 4.5.7)
intersection method	intersects the invariants to find the guards (Section 4.5.7)

Guards Identification: Distance Method

The *distance method* first identifies the facets of the invariant of the current location. A facet is an $(m–1)$-dimensional face of an m-dimensional polytope. Note that, a 0-dimensional face consists of a single point and is called a vertex, while a 1-dimensional face is called an edge and represents a line segment, and so on. Since the running example has an order of $m = 2$, the methodology identifies the edges of the invariant. In general, identify the facets of the invariants is straight forward. For each location, a convex hull is generated using the vertices of an invariant. The results are the points that make up the facets of the invariants. Note that, each facet in the m-dimensional space is described by m points. The task now becomes finding the facets that qualify as guards.

Based on the facets found, the points of the guards are identified in the $\mathcal{S}_{virt}$ space as described in Algorithm 6. Note that $\boldsymbol{e}_{step}$ represents the minimum step vector containing the minimum state step for each dimension in the $\mathcal{S}_{virt}$ domain, while $\boldsymbol{tol}$ is a specified tolerance vector.

Algorithm 6 is executed for each location loc_i resulting in a structure called *guards*. This structure contains all guards of the current location along with the information of the target locations (see line 11). At this point, each guard is defined as a set of m points in an m-dimensional space.

In order to examine the m points that can span a guard two tests are performed. First, the algorithm examines if the current facet and the line containing the operation points of the current location loc_i and the next location (loc_j) intersect (see line 6 of Algorithm 6). If this condition is true, the current facet is favored by increasing $\boldsymbol{tol}$. Second as line 9 specifies, the distances between each point in the next location to the mean of the facet points $\bar{\boldsymbol{m}}_{facet}$ are computed for each dimension separately and saved in the distance matrix $\boldsymbol{D}_g$. For each point, all corresponding column entries in $\boldsymbol{D}_g$ are compared to the state step $\boldsymbol{e}_{step}$ scaled by the tolerance vector $\boldsymbol{tol}$ through an element-wise multiplication (see line 10). This can be thought of inspecting if the points of the

Algorithm 6 determining the points of the guards using the *distance method*

```
1:  procedure $guards = findGuards(loc_i, Loc, inv, \boldsymbol{tol}, \boldsymbol{e}_{step}, \mathcal{S}_{virt})$
2:      identify the facets of the invariant $inv$ of the current location $loc_i$
3:      for each facet do
4:          for next location $loc_j \neq loc_i$ in $Loc$ do
5:              calculate the mean ($\bar{\boldsymbol{m}}_{facet}$) from the points of the facet
6:              if facet intersects operation line then
7:                  favor this facet by increasing the $\boldsymbol{tol}$
8:              end if
9:              compute distance $\boldsymbol{D}_g$ between $loc_j$ points and $\bar{\boldsymbol{m}}_{facet}$
10:             if $\boldsymbol{D}_g < (\boldsymbol{tol} \odot \boldsymbol{e}_{step})$ then
11:                 add $loc_j$ and facet points to $guards$ of $loc_i$
12:                 count points with $\boldsymbol{D}_g < (\boldsymbol{tol} \odot \boldsymbol{e}_{step})$
13:                 break
14:             end if
15:             set $\boldsymbol{tol}$ to default value
16:         end for
17:     end for
18: end procedure
```

next location loc_j are in an m-dimensional hyperrectangle with a center at $\bar{\boldsymbol{m}}_{facet}$ and bounded by the points $\bar{\boldsymbol{m}}_{facet} - (\boldsymbol{tol} \odot \boldsymbol{e}_{step})$ and $\bar{\boldsymbol{m}}_{facet} + (\boldsymbol{tol} \odot \boldsymbol{e}_{step})$. If at least the distance from one point from loc_j to $\bar{\boldsymbol{m}}_{facet}$ is less than $(\boldsymbol{tol} \odot \boldsymbol{e}_{step})$ (line 10), the points of the facet are considered guard points and saved along with the target location in the data set *guards*. Moreover, the points of loc_j that pass the condition at line 10, and are thus inside the hyperrectangle, are counted (line 12) and used to specify the importance of the identified guard.

Fig. 4.25 shows the guard identification for the location $g1r1$ of the running example. The invariant was modeled as an interval hull in a 2-dimensional space. Therefore, there are four edges to consider. Fig. 4.25a shows the examination of the first edge colored in green for the possibility of being a guard from $g1r1$ to $g2r2$. As observed, this edge does not intersect the red line containing the operating points of $g1r1$ and $g2r2$. Thus, this edge is not favored according to line 6 of Algorithm 6. Fig. 4.25b on the other hand, shows that the fourth edge does intersect the line containing the operating point of $g1r1$ and $g2r2$. Thus, this edge is favored by increasing $\boldsymbol{tol}$, thereby increasing the size of the test rectangle which visualizes the condition at line 10. Note that, initially all test rectangles have the same size.

The test rectangle in Fig. 4.25a does not contain any points from $g2r2$. Hence, this edge does not qualify as a guard. In contrast, the test rectangle in Fig. 4.25b contains several points from the target location. Thus, this edge ($edge_{g1r1,4}$) qualifies as a guard. Hence, the points of this edge (surrounded by purple circles in Fig. 4.25b), along with the target location $g2r2$ and the number of points from $g2r2$ inside the test rectangle are saved in the data structure *guards* of $g1r1$.

As the guard points are in general the points of the facets of the invariants, the complexity of the guard identification is directly linked to the specified invariant representation. Hence, the more complex the geometric shape of the invariant is, the more exhaustive the calculation becomes. For example, considering location $g1r1$ of the running example with an invariant modeled as an interval hull, the location has 4 edges. Additionally to $g1r1$, there are two locations: $g2r1$ and $g2r2$. Hence,

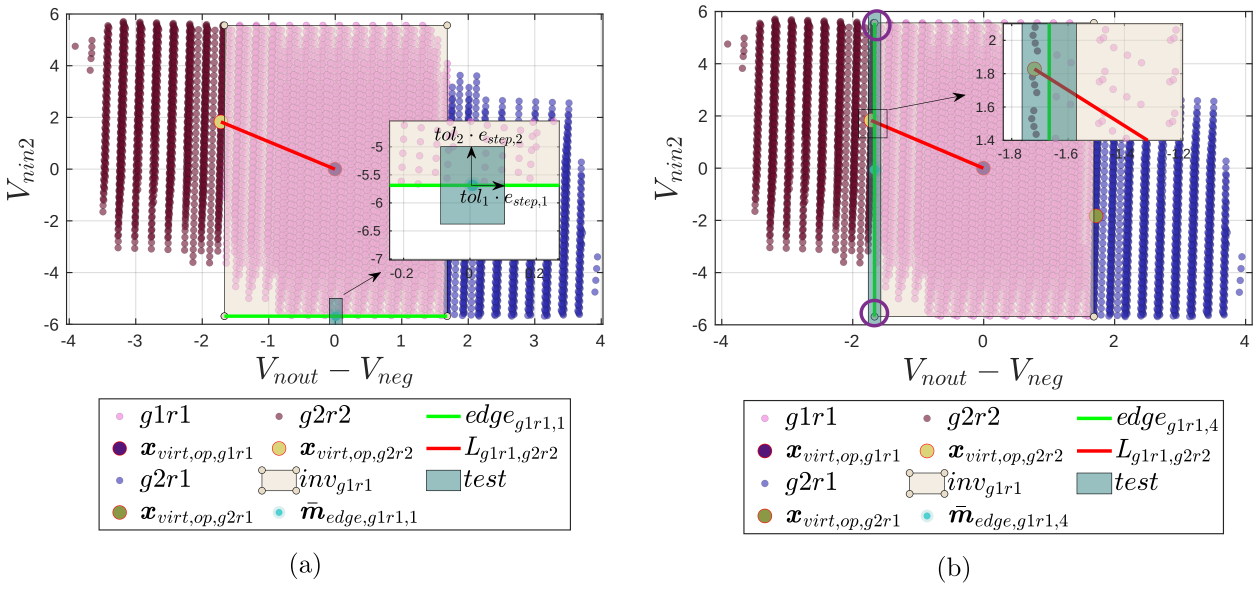


Fig. 4.25. Part of the results of the guard identification for the location $g1r1$ with the invariant modeled as an interval hull. The test rectangle is denoted as $test$ with the center $\bar{\boldsymbol{m}}_{edge,g1r1,i}$, where i denotes the number of the test case.

for each location there are 8 cases to check for possible guards. If the invariant of $g1r1$ is modeled as a polytope like shown Fig. 4.26, the invariant has 17 edges, with two possible target locations, there are thus 34 cases for this location that need to be checked.

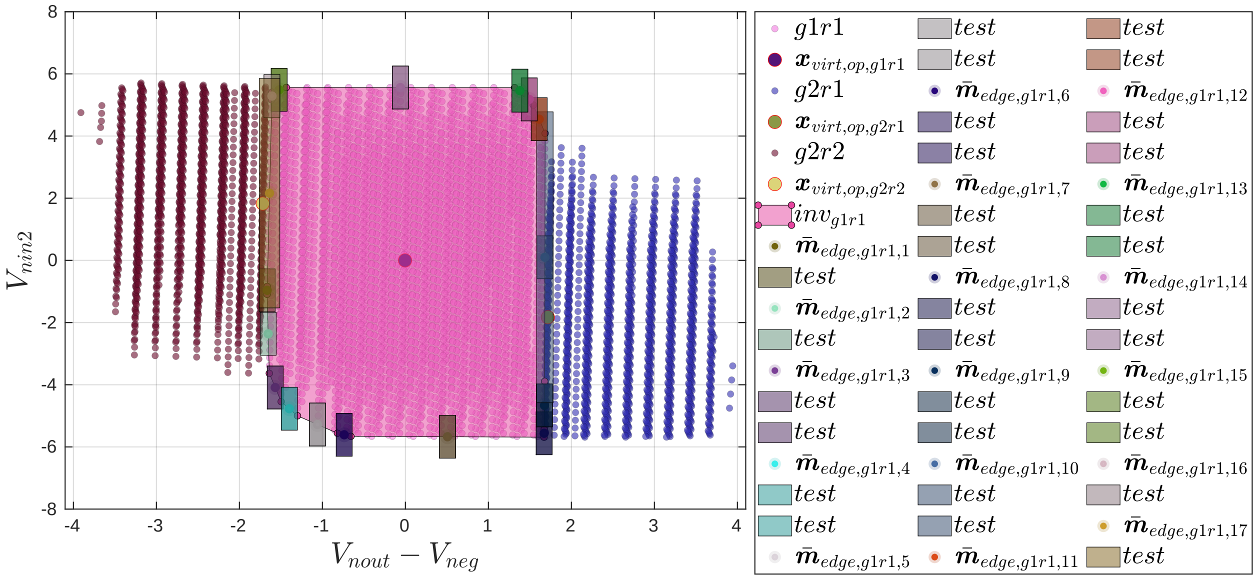


Fig. 4.26. Part of the result of the guard identification for the location $g1r1$ with the invariant modeled as a polytope with 17 edges. A test rectangle is denoted as $test$ with the center $\bar{\boldsymbol{m}}_{edge,g1r1,i}$, where i denotes the number of the test case.

To speed-up the guard identification, tests can be skipped as line 13 in Algorithm 6 indicates. This is possible as we demand that guards have only one target location according to Definition 1. Therefore, if a facet qualifies as a guard from one location to the other, the next facet is handled, as line 13 exits the for loop between lines 4-16. For the current example, this reduces the test cases of the guards identification for $g1r1$ with the invariant modeled as interval from 8 to 7 cases, of which only 2 cases yield guards points. On the other hand, for the second scenario where the invariant was modeled as polytopes, the test cases are reduced from 34 to 30 tests for the guards identification of $g1r1$, of which only 7 cases yield guard points to the next locations as observed in Fig. 4.26. The first 4 of these 7 edges: $[1, 2, 16, 17, 8, 9, 10]$ yield guard points to $g2r2$, while the remaining ones yield guard points to $g2r1$. Note that, every edge is tested in general twice. In the first test, the edge is tested to be a guard to $g2r2$. If this test fails, a second test is launched to examine if the edge is a guard to $g2r1$. In Fig. 4.26 only the performed tests are illustrated.

Moreover, as the location are ordered (Section 4.5.3), the runtime of the algorithm can be additionally reduced the set of target locations. For example, during the guard identification of $g2r2$, only the intermediate neighbor location $g1r1$ needs to be examined as a possible target location (line 4 of Algorithm 6), as no points from $g2r1$ surround $g2r2$.

From the previously identified guard points, the desired geometric shapes of the guards are calculated. For a guard modeled as a polytope, zonotope, or interval hull the guard points are simply hulled by the geometric shapes analogously to the approach from Section 4.5.6. However, in some cases an additional modification is necessary. Since the previous mentioned geometric shapes are always considered to have a volume, points that have the same value across one dimension, for example points in a 3d space that are coplanar, are modified by adding to the defective dimension a lower and an upper bound. This small tolerance value (tol_G) assures that the guards have a thickness, however, results also in portions of the guards that are outside the invariants.

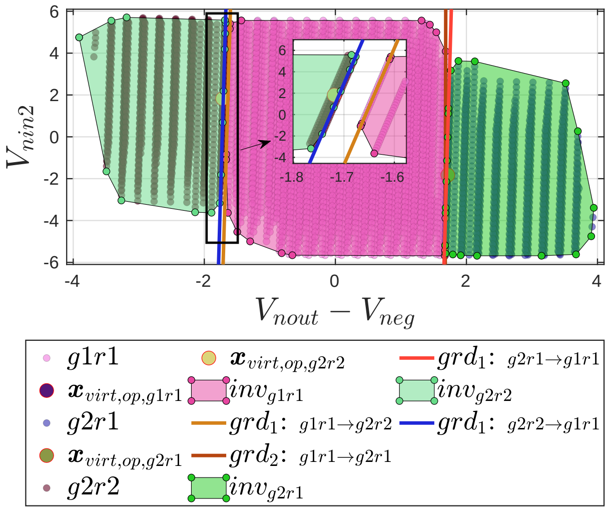
For halfspace guards, a different approach is used. To determine a halfspace in an m-dimensional space, in general m points are needed. For the case that $m > 2$, the guard points should be non-colinear. Since these points belong to the halfspace, they obey Eq. (2.1). For example, in a 2D space ($m = 2$), with $\boldsymbol{c}$ and d as defined in Section 2.1.1 and two points $\boldsymbol{x}_{virt,1}$ and $\boldsymbol{x}_{virt,2}$, following equation can be specified:

$$\begin{aligned}\boldsymbol{c}^T \boldsymbol{x}_{virt,1} &= d\\ \boldsymbol{c}^T \boldsymbol{x}_{virt,2} &= d\end{aligned}$$

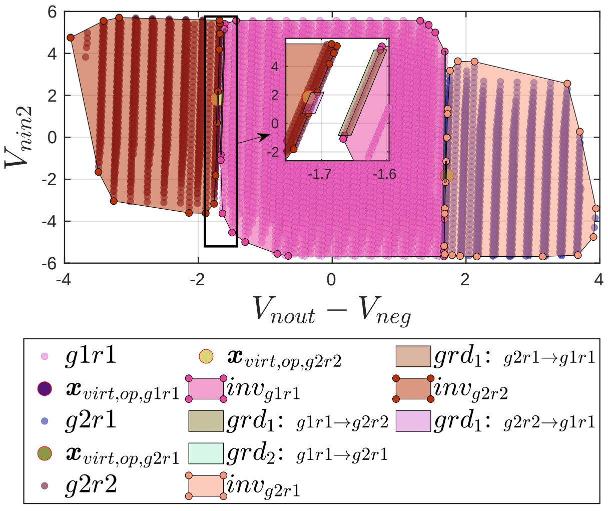
This equation represents an underdetermined system of equation. Solving this equation yields in this case the equation of a line. Next, the intersection of the halfspace for the found values of $\boldsymbol{c}$ and d with the operating point of the current location is tested according to Eq. (2.1) i.e. it is examined if the operating point lies on the halfspace. If this intersection returns false, the sign of both, the normal vector $\boldsymbol{c}$ and the signed scalar d, is inverted.

For each set of guard points, a guard in the desired geometric shape is calculated. However, based on the enclosed points counted at line 12 of Algorithm 6, for all transition between two locations, the guards found can be reduced to a dominant one. This is done by selecting the guard that encloses the most points to the target location. This is referred to as dominant guard reduction.

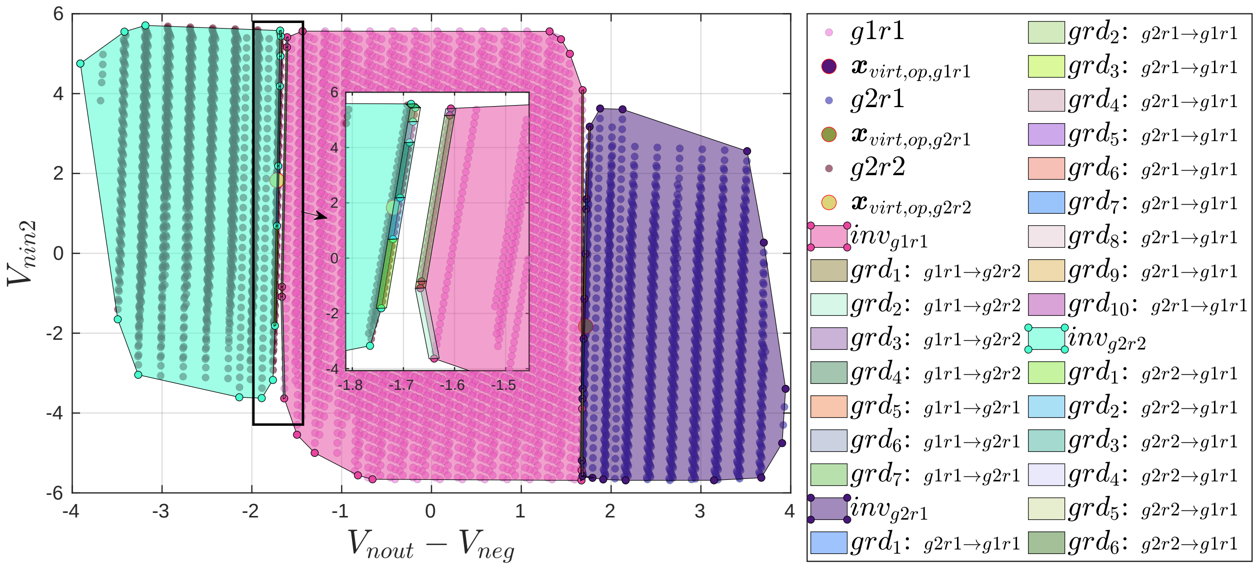
Fig. 4.27 shows the complete results of the guard identification for the running example with the *distance method*. While Fig. 4.27a shows the result of the guards identification performed with halfspace guards, the guards in Fig. 4.27b and Fig. 4.27c are of polytopic nature. Moreover, in Fig. 4.27a and Fig. 4.27b the found guards between two locations are reduced to a single dominant



(a)



(b)



(c)

Fig. 4.27. Results of the guards identifications using the *distance method*. In (a) and (b) a dominant guard reduction is performed, while in (c) all guards are used. The guards are represented in (a) as halfspaces, while in (b) and (c) as polytopes.

one. All calculations were performed with invariants modeled by polytopes. These invariants were not enlarged as shown. However, polytopic guards were enlarged by a tolerance value set for this example to $tol_G = 0.01$. This enlargement is performed by the Minkowski addition of the tolerance value to the polytopic guards, resulting in parts of the guards being outside the invariants. These parts can be neglected, as guards are only valid inside and on the invariants (Definition 1). As observed in Fig. 4.27b, the found guards are not acceptable as they are too small and thereby limit the transition regions. This is especially true for grd_1 of location $g2r2$. On the other hand, modeling the same guards with halfspace, as in Fig. 4.27b, yields more acceptable results. Fig. 4.27c shows the best results, as all edges to the neighbor locations are modeled as guards. However, finding these guards consumed more computation time compared to the previous two guards identifications.

Generally speaking, the best results are obtained when all guards are always considered, as all facets are considered. Reducing the guards to a dominant one yields the best results if only few guards are available. This is the case when the invariants are modeled by interval hulls. If this is not desired and the invariants are modeled as polytopes or as zonotopes, the dominant guard should be modeled as halfspace, even though this does not necessary ensure good results, it solves the problem of too short guards that limit the possible transitions. Moreover, better results are expected with guards of type halfspaces, as the modeling process did not inject any unnecessary tolerance into the guard representations, thereby modeling accurately the borders of the locations.

Good and fast results can be obtained by the guard identification with invariants of type intervals. On the other hand, while the results might be still acceptable with more complex geometric shapes, the run time of the abstraction process suffers due to the computation of the distance matrix $\boldsymbol{D}_g$ at line 9 of Algorithm 6 and the high number of possible test cases. This becomes an even greater problem if the number of the sampled points or the number of locations increases. Solid extensions were deployed that can handled this problem, by reducing the number of points considered in the distance matrix $\boldsymbol{D}_g$, or by performing a dominant guard reduction.

Guards Identification: Intersection Method

To overcome the flaws of the previous method, especially those concerned with the number of sampled points, the geometric representation of the invariants, and the poor controllability of the results, the *intersection method* was created. This method finds the points of the guards between two locations based on the intersection of their invariants. As often the invariants do not intersect after construction, one of the two invariants of adjacent locations is enlarged until it intersects the other one. For the two location loc_i and loc_j, where the current location is loc_i with an invariant inv_{loc_i} for which the guards are being determined with respect to the target location loc_j with an invariant inv_{loc_j}, the process of finding the points of the guards is stated in Algorithm 7. Both invariants are modeled temporary as polytopes. Based on the $flag$, Algorithm 7 decides which polytope to increase ($\mathcal{P}_{temp}$) at line 14. Hence, the *intersection method* is divided into two methods:

intersectionI method increases the polytope modeling the invariant of the current location at each iteration ($flag = 1$). After the guards are found the invariant of the current location is enlarged to cover the identified guards

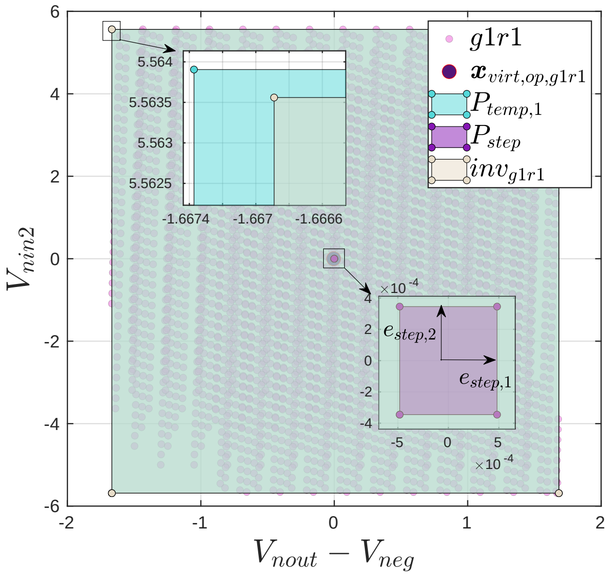
intersectionII method increases the polytope modeling the invariant of the target location at each iteration ($flag = 0$). This enlargement is only temporarily and neglected at the end of the guard identification

The polytope modeling the invariant ($\mathcal{P}_{temp}$) of either the target or the current location is increased at every iteration at line 14 of Algorithm 7, by computing the Minkowski sum of this polytope and an interval formed from the minimum state step vector $\boldsymbol{e}_{step}$, which is in fact an interval hull but can be as well consider a polytope ($\mathcal{P}_{step}$). The result of this enlargement is illustrated in Fig. 4.28a for an invariant modeled as an interval hull. For consistency with the previous stated algorithm, all geometric shapes will be labeled as polytopes for the rest of this section.

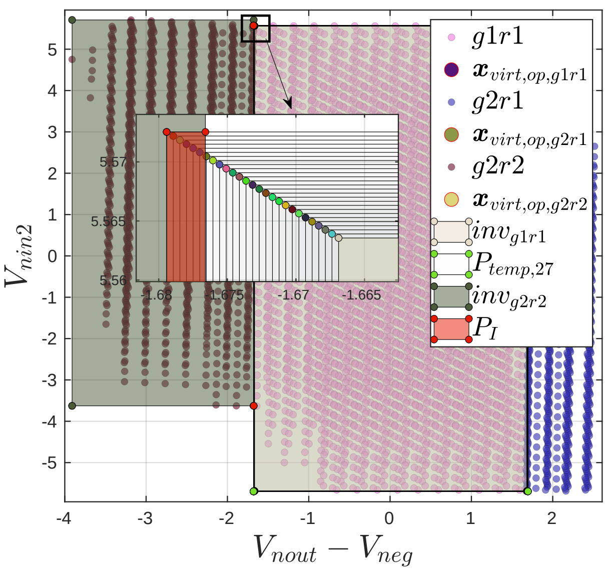
From this point on, the algorithm increases the polytope $\mathcal{P}_{temp}$ until either an intersection with the next location's invariant occurs, or the limit is reached, as specified at line 12. In the case

Algorithm 7 Determining the point of the guards using the *intersection method*

```
1:  procedure $guards = findGuards(inv_{loc_i}, inv_{loc_j}, Loc, limit, \boldsymbol{e}_{step}, flag)$
2:      if $flag == 1$ then
3:          $\mathcal{P}_{temp,0} = inv_{loc_i}$
4:          $\mathcal{P}_{const} = inv_{loc_j}$
5:      else
6:          $\mathcal{P}_{temp,0} = inv_{loc_j}$
7:          $\mathcal{P}_{const} = inv_{loc_i}$
8:      end if
9:      $\mathcal{P}_I = intersect(\mathcal{P}_{temp}, \mathcal{P}_{const})$
10:     $\mathcal{P}_{step} = intervalHull(-\boldsymbol{e}_{step}, \boldsymbol{e}_{step})$
11:     $i = 0$
12:     while $limit$ not reached && isempty($\mathcal{P}_I$) do
13:         $i = i + 1$
14:         $\mathcal{P}_{temp,i} = \mathcal{P}_{temp,i-1} + \mathcal{P}_{step}$
15:         $\mathcal{P}_I = intersect(\mathcal{P}_{temp,i}, \mathcal{P}_{const})$
16:     end while
17:     if not isempty($\mathcal{P}_I$) then
18:         add $loc_j$ and verttices of $\mathcal{P}_I$ to $guards$ of $loc_i$
19:     end if
20: end procedure
```



(a)



(b)

Fig. 4.28. The enlargement of the current invariant $g1r1$ by one step size is presented in (a). The result of the *intersectionI method* between location $g1r1$ and $g2r2$ is shown in (b), with invariants represented as interval hulls before the identification of the guards.

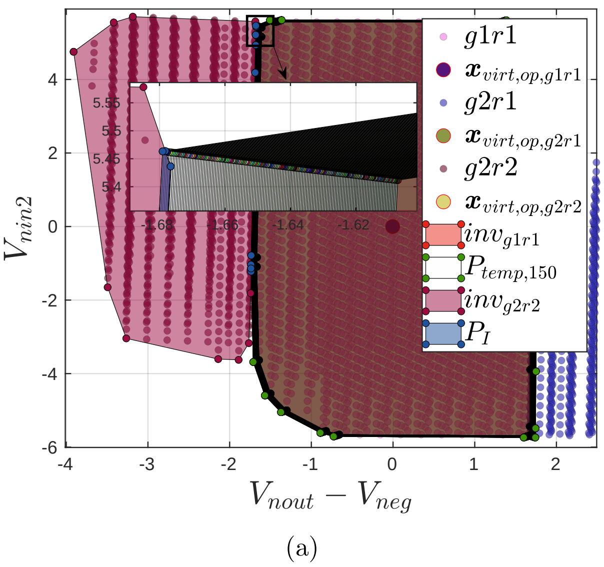
an intersection occurs, the intersection is represented as a polytope $\mathcal{P}_I$. The limit can be either defined as the maximum allowed volume increase of the polytope $\mathcal{P}_{temp}$, or as a maximum number of iterations. Moreover, additional conditions which are concerned with the volume of the polytope $\mathcal{P}_I$ can be included as well at line 12 of this algorithm, but will be skipped here for simplicity.

For the running example the guard identification with the *intersectionI method* was performed for the current location $g1r1$, considering only the target location $g2r2$. The result is shown in Fig. 4.28b. The identification has been carried out with the *limit* set to:

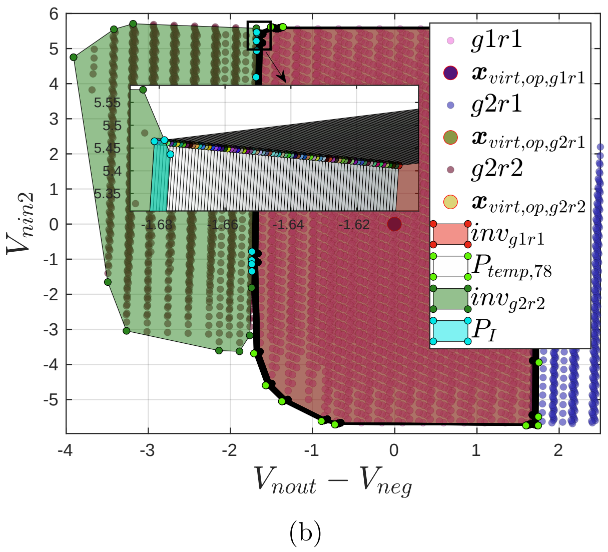
$$volume(\mathcal{P}_{temp,i}) < k(volume(\mathcal{P}_{temp,0}) + volume(\mathcal{P}_{const})),$$

where k is set to $\frac{1}{2}$, the subscript i denotes the current iteration, and $\mathcal{P}_{temp,0}$ denotes the initialization to the invariant inv_{g1r1}. On the other hand, $\mathcal{P}_{cont}$ has been initialized to the invariant inv_{g2r2}. Additional to the *limit*, a condition concerning $\mathcal{P}_I$ has been set. This condition states that 5 additional iterations are performed after the first intersection between $\mathcal{P}_{temp,i}$ and $\mathcal{P}_{const}$ occurs. This assures in case the invariants are overlapping, that at least few iterations are performed controlling thereby the size of the guards. Usually this value is set to 1, but was set to 5 for the running example for illustration purposes. At iteration 22, the intersection between $\mathcal{P}_{temp,i}$ and $\mathcal{P}_{const}$ occurs for the first time. Thus, the algorithm terminates after 27 iterations as shown in Fig. 4.28b. For simplicity, only the last polytope $\mathcal{P}_{temp,27}$ was added to the legend. The red polytope $\mathcal{P}_I$ represents the guard found. Note that, the guard type was set to polytope.

This method depends on the geometric representation of the invariants used (Section 4.5.6) and the minimum state space step vector $\boldsymbol{e}_{step}$. As observed in Fig. 4.28, several iterations are needed until an intersection of the invariants occurs, even though the simplest invariant type is chosen. On the other hand, performing this identification with the same settings, but with invariants modeled as polytopes results in much more computational effort as shown in Fig. 4.29a. This time 150 iterations are needed till the algorithm terminates. This lies in the fact that the distance between the invariants increased, as a more precise invariant representation is used. In Fig. 4.29b, the step vector $\boldsymbol{e}_{step}$ has been doubled compared to its previous value. As observed, nearly half the number of iterations is needed compared to the previous run. However, the identified guard yields a greater volume compared to the previous case. For simplicity, in both figures only the last computed



(a)



(b)

Fig. 4.29. Result of the *intersectionI method* for the current location $g1r1$ to the target location $g2r2$ with polytopic invariants. In (a) state step vector $\boldsymbol{e}_{step}$ is used, while (b) uses twice this value per iteration.

polytope $\mathcal{P}_{temp,150/78}$ was added to the legend.

After finding the intersection polytope $\mathcal{P}_I$, the polytope is changed into the specified geometric shape. This is performed by first extracting the vertices of the polytope, followed by generating the desired shape from these vertices according to Section 2.1. In case the guard type is set to halfspace, the *intersection method* finds the outer facets of the intersection polytope $\mathcal{P}_I$ with the help of the operation points in each location. In Fig. 4.30, the intersection polytope $\mathcal{P}_I$ from Fig. 4.29b is used to determine the halfspace guards from $g1r1$ to $g2r2$. In the left of Fig. 4.30 the identified intersection polytope is illustrated. This polytope is decomposed into its halfspaces. The halfspaces that contain the operation point $\boldsymbol{x}_{virt,op}$ of $g1r1$ are then selected as guards (green lines), while the halfspaces that do not contain this point are neglected (red lines).

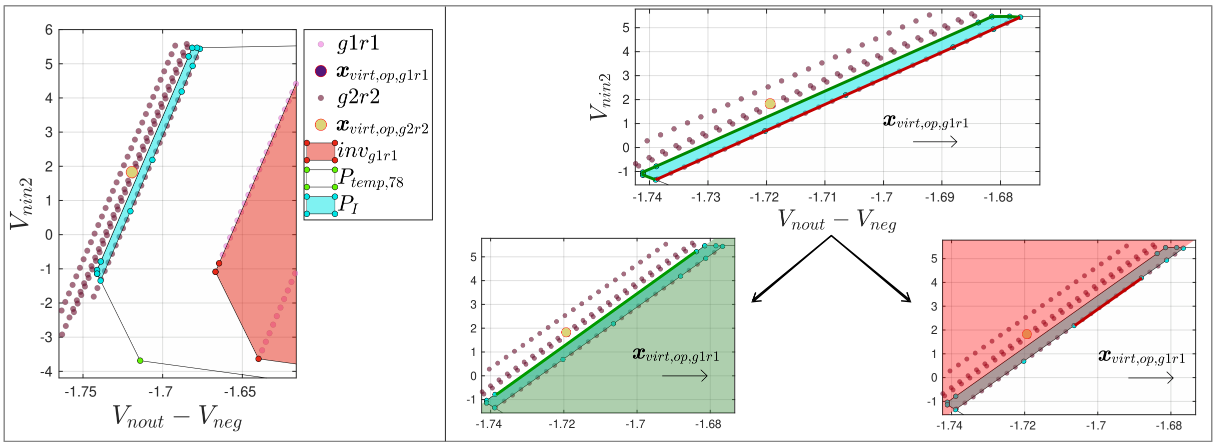


Fig. 4.30. Determining the halfspaces of the polytope $\mathcal{P}_I$ that are qualified as guards from $g1r1$ to $g2r2$ after using the *intersectionI method*.

Unlike the *distance method*, reducing the guards to a dominant one is only possible if halfspace guards are used. If the guards are of any other type, all intersections found form the guards. Moreover, guards found with the *intersection method* are not modifications by any tolerances.

In the last step of the guard determination, the invariants are adjusted to enclose the found guards in the case the *intersectionI method* was used, as it makes little sense to define guards outside the invariants (see Definition 1). This is performed by adding the points of the found guards to the points of the invariants. By that, the invariants are recalculated with the new set of points, resulting in enlarged invariants. Hence, in this mode the invariants of the locations overlap in the final model. In case the *intersectionII method* is used, the invariants of the target locations are only temporarily enlarged. Thus, the invariants of the final model are not modified. Consequently, the *intersectionI method* can inject errors into the system behavior, as the invariants exceed the sampled points of the locations. Note that, in the *distance method* stated at the beginning of Section 4.5.7, the invariants were not modified.

Fig. 4.31 shows the complete results of the guard identification using both modes of the *intersection method*. This time the condition for the intersection polytope $\mathcal{P}_I$ has been set to:

$$volume(\mathcal{P}_I) \geq \frac{1}{80}(volume(\mathcal{P}_{temp,0}) + volume(\mathcal{P}_{const})),$$

instead of limiting this method as previously done with the number of iterations (Fig. 4.29), the volume of the intersection polytope $\mathcal{P}_I$ was used. While in Fig. 4.31b the *intersectionII method* was used, Fig. 4.31a used the *intersectionI method*. As observed, both options yield good results. How-

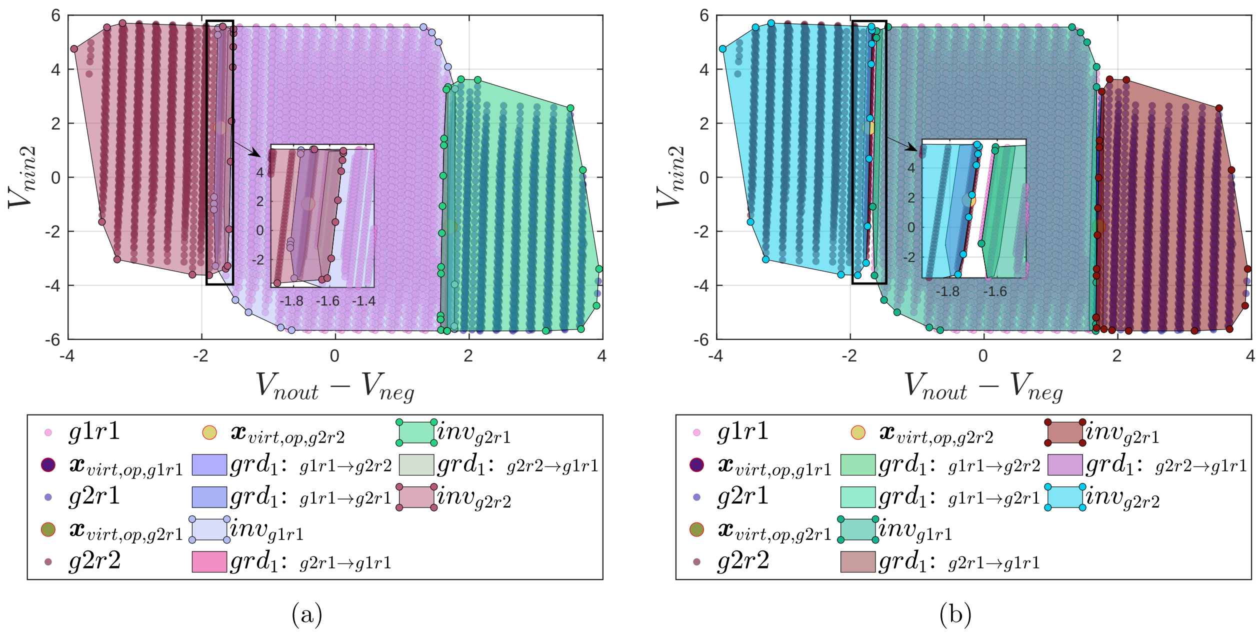


Fig. 4.31. Result of the (a) *intersectionI method* and (b) the *intersectionII method*. In (a) the invariants are enlarged to cover the identified guards.

ever, the *intersectionI method* resulted in an enlargement of the invariants as shown in Fig. 4.31a. This enlargement happened post to the guards identification of each location, and thus assured that the invariants contain the guards.

In general, this method yields good results if the condition for the intersection polytope $\mathcal{P}_I$ are well chosen. Moreover, depending on the state step vector $\boldsymbol{e}_{step}$, the runtime can be improved in potentially loss for accuracy. Regarding the guards, halfspace guards yield preciser descriptions than the remaining shapes. This is due to the fact that the geometric shapes used i.e. polytopes, zonotopes, and interval hulls are convex, hence, this causes over approximation especially as the dimensions increase compared to halfspace guards. However, identifying the halfspace guards from the intersection polytopes is still not optimal, and thus error prone.

In case the guard and invariant identifications were performed in the $\mathcal{S}_\lambda$ domain, and thereby the previous calculations were performed in this domain, the invariants can be optionally enlarged after the identification. However, if the previous calculations were performed in the $\mathcal{S}_{virt}$ space, as this is the case here, only the *intersectionI method* can enlarge the invariants. All remaining methods do not modify the invariants. This is on purpose disabled as modifying the invariants, for the case that the guards and invariants are identified in the $\mathcal{S}_{virt}$ domain, does not yield good results as no accurate transformation to the $\mathcal{S}_\lambda$ is possible as described in the next section.

4.5.8 State Space Transformation of the Guards and Invariants

The 7th block of Fig. 4.12 transforms the found guards and invariants into the $\mathcal{S}_\lambda$ space. This is necessary if the previous calculations were carried out in the $\mathcal{S}_{virt}$ space. Specifically, this section targets the results from Section 4.5.6 and Section 4.5.7, where the invariants and guards were identified in the $\mathcal{S}_{virt}$ domain. In all remaining calculations of Section 4.5, the results were calculated in both state spaces. In case the guards and invariants were identified in the $\mathcal{S}_{\lambda}$ space, no transformations are necessary, and thus the transformation block from Fig. 4.12 is skipped.

The invariants are taken as calculated in Section 4.5.6, disregarding the method chosen for the guard identification and the possible modification applied to the invariants. Moreover, it is necessary to take all sampled points in a location, rather than just to consider the vertices of the invariants in the $\mathcal{S}_{virt}$ space, to determine the invariants in the $\mathcal{S}_{\lambda}$ space. The reason for this lies in the fact that transforming only the vertices of an invariant into the $\mathcal{S}_{\lambda}$ space and enclosing the result by a convex hull, does not necessary assure that all sampled points belonging to this location are covered by the found hull. Fact is, each sampled point has in general different transformation matrices calculated by *Vera* (see Eq. (4.9)). Thus, when transforming points with different transformations, a change of the position of the transformed points between the two spaces can occur. This becomes an even more challenging task when the invariants were modeled with interval hulls or zonotopes, as the vertices of these shapes were not calculated by *Vera* in general, and therefore no information regarding the transformations are available. This becomes clear when examining the possible invariant representation from Fig. 4.24 for the location $g1r1$ as shown in Fig. 4.32. As observed, only in the case the invariant is modeled as a polytope, the vertices of the polytope are from the sampled points. Note that, in this case the vertices of the polytope are from the vertices of the convex hull ($\boldsymbol{v}_{g1r1}$), which are in terms from the sampled points.

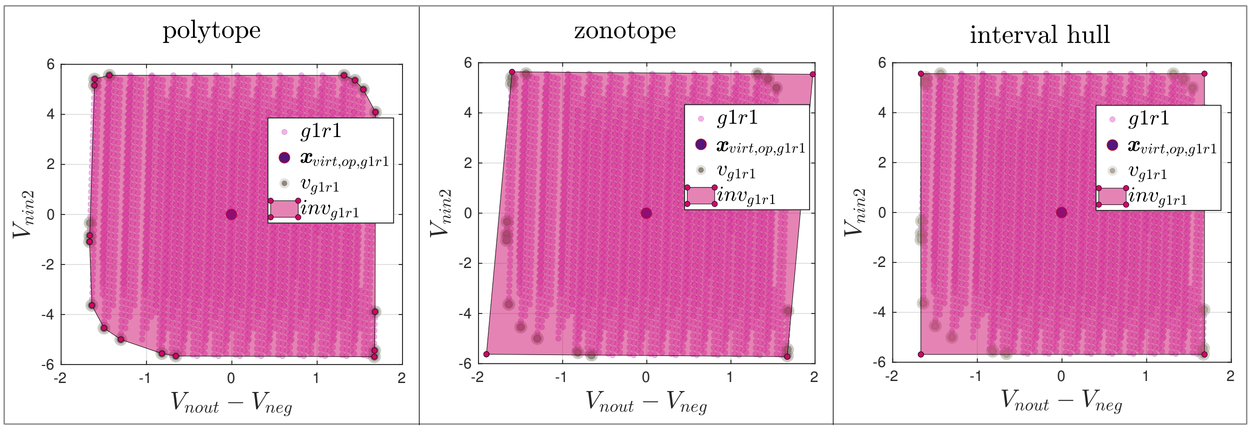


Fig. 4.32. Invariant representation of $g1r1$ using different geometric shapes. Only if the invariant is modeled as a polytope, the vertices of the invariant are from the sampled data.

However, even if the transformation matrices are available, as is the case for invariants modeled as polytopes, it is not enough to consider only the vertices of this shape. This is due to the fact that each sampled point is coupled with different transformation matrices (see Section 3.2), yielding different projections as illustrated in Fig. 4.33.

This becomes clear when examining Fig. 4.34. Fig. 4.34a shows the result of the guard and invariant identifications performed previously (Fig. 4.31b) with invariants modeled as polytopes and guards found with the *intersectionII method* and modeled also as polytopes. The numbers in the figure indicate the indices of the points that form the convex hull of the first location $g1r1$. Note that the invariant inv_{g1r1}, which is a polytope formed from these points, dropped few of these

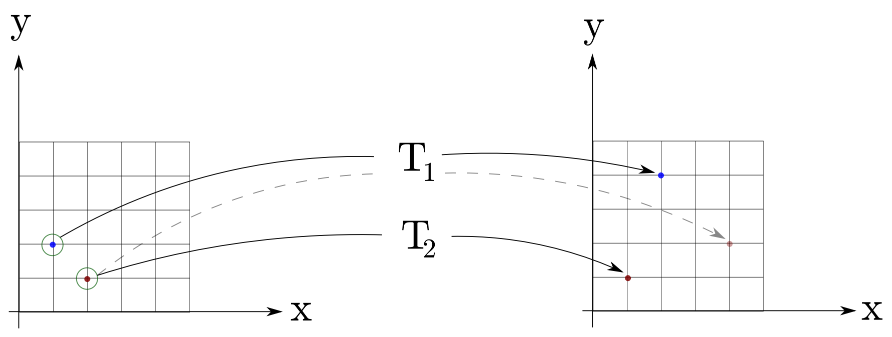


Fig. 4.33. Transforming sampled points with different transformation matrices

points. The points, which define the convex hull in the $\mathcal{S}_{virt}$ space, are transformed into the $\mathcal{S}_{\lambda}$ space, and are denoted by $\boldsymbol{v}_{est,g1r1}$ in Fig. 4.34b. However, generating a polytope from these points in the $\mathcal{S}_{\lambda}$ space does not hull all sampled points of the location. Fig. 4.34c shows the desired result.

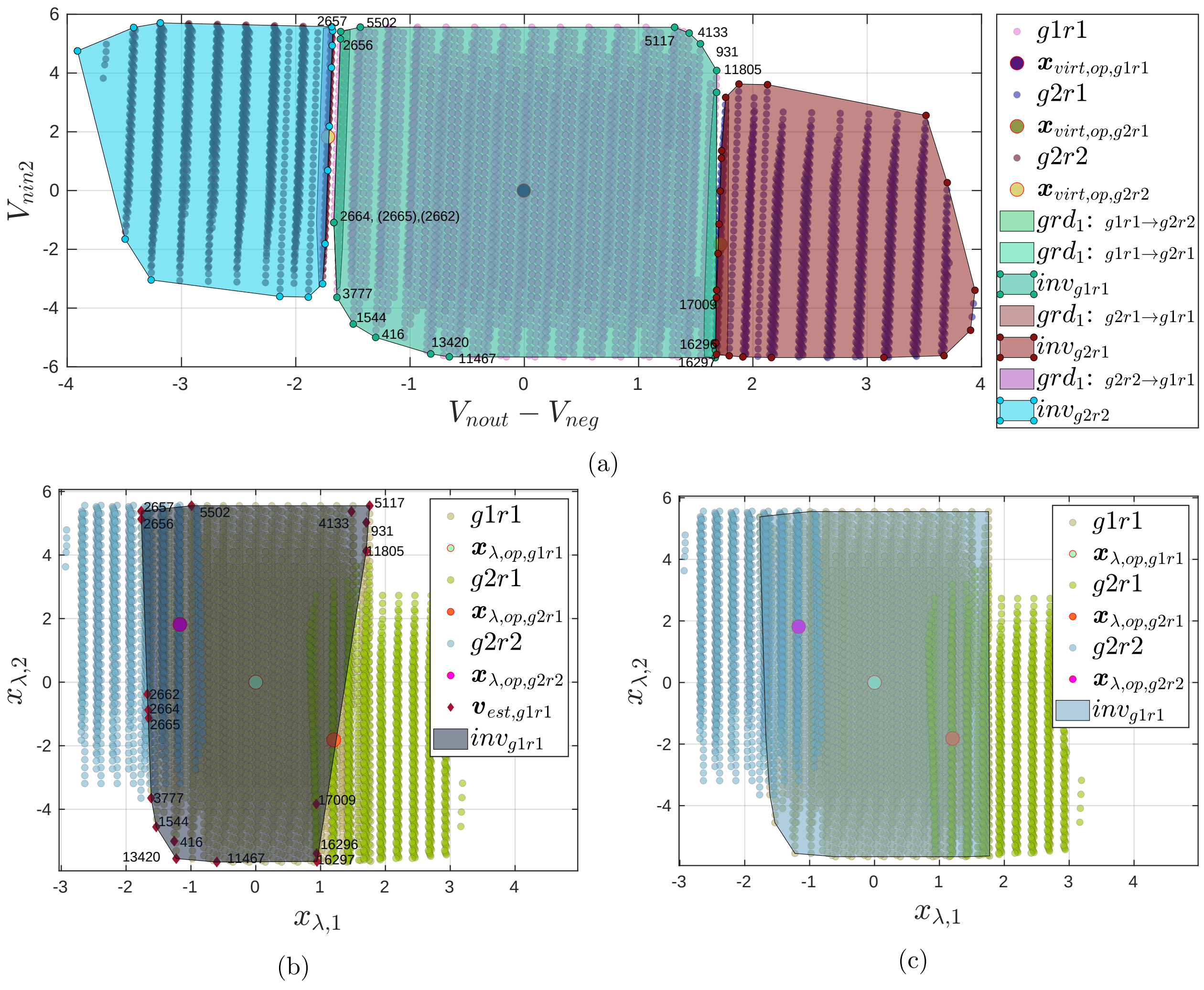


Fig. 4.34. Examining the vertices of the invariant of location $g1r1$ in (a) the $\mathcal{S}_{virt}$ space and in (b) and (c) in the $\mathcal{S}_{\lambda}$ space.

Regardless of the method used to obtain the $\mathcal{S}_\lambda$ space, using all points the invariants in the $\mathcal{S}_\lambda$ space are identified. Similarly to the process of finding the invariants in the $\mathcal{S}_{virt}$ space, the invariants in the $\mathcal{S}_\lambda$ space are found by hulling the points of a location with a convex hull, followed by transforming the hull into the desired representation (see Section 4.5.6).

Now it becomes clear why the invariants should not be modified. Modifying the invariants by enlarging them yields points describing the convex hull that were not sampled. Consequently, these points neither have transformation matrices nor corresponding points in the $\mathcal{S}_\lambda$ domain. Modification applied on the invariants are neglected during the transformation of these shapes from the $\mathcal{S}_\lambda$ space to the $\mathcal{S}_{virt}$ space. These modifications can be applied to the invariants after they have been found in the $\mathcal{S}_\lambda$ space, for example by applying a state step enlargement corresponding to the minimum state step in the $\mathcal{S}_\lambda$ space. However, finding the guards in the $\mathcal{S}_\lambda$ space becomes then an even more challenging task. For this reason, invariant modifications should not be performed if the $\mathcal{S}_{virt}$ space is used to find the invariants and guards. Specifically, the *intersectionI method* should not be used in the $\mathcal{S}_{virt}$ domain, as the enlargements of the invariants yield vertices that do not belong to the sampled data set.

The fact that a point does not belong to the sampled data set is quite often. The simplest approach involves finding surrounding sampled points and use them instead. As long as this is performed inside a location's invariant, acceptable results can be expected as these points share similar transformation matrices and system behaviors, which is especially true if various filters and modification from Section 4.5.5 were applied. However, the results are not optimal. Moreover, in the case that the points are outside the invariants, this is not feasible. The best option in this case is to replace the points by the borders points of the invariants. Obviously, in the case the *intersectionI method* is used, this corresponds to reversing the enlargement of the invariants, thereby using the unmodified invariants. If the guards lie outside the unmodified invariants, bad results can be expected. Therefore, the *intersectionII method* is always used if the guards and invariants were calculated in the $\mathcal{S}_{virt}$ space, while the *intersectionI method* is dropped out in this case due to the problems this methodology faces during transformation of the results.

Due to the stated limitation to unmodified invariants, guard points are always very close to the sampled data, even though the guard points are often not directly from sampled data. Only the *distance method* with invariants modeled as polytopes and guards modeled as halfspaces usually forces the guard points to be from the sampled data set. While the *distance method* with guards not modeled as halfspaces usually results in guards that are partially outside the invariants (due to tol_G), the *intersectionII method* always results in guard inside and on the borders of the invariants.

Generally speaking, guard points may not belong to the data set calculated by *Vera*, but lie in the surrounding of points which do. Still, the guards found need to be transformed into the $\mathcal{S}_\lambda$ domain from the guards identified in the $\mathcal{S}_{virt}$ space. For this, four methods have been implemented:

estT method estimates a transformation matrix $\boldsymbol{T}_{loc}$ based on all sample points belonging to a location. This transformation matrix is used to transform the guard points

sampT method finds the sampled points inside a guard and transforms them to the $\mathcal{S}_\lambda$ domain. In case the guards do not contain any sampled data point, the closest points to the guards are found. These points are transformed to the $\mathcal{S}_\lambda$ space *disT method* extends the first method. First the guards are found using the *estT method*. In the $\mathcal{S}_\lambda$ space, for each guard the closest facet of the invariant is identified by finding the closest facet points. The found edges represent the new guards

halfT method computes first the guards as in the *estT method*. In the $\mathcal{S}_\lambda$ domain, the invariants are modeled as polytopes in the halfspace representation. The distances from the guard points to the hyperplanes of halfspaces are calculated and the closest halfspace is found. Based on the halfspace, the closest points of the invariants are identified and used as guard points

The *estT*, the *disT*, as well as the *halfT* methods estimate all a transformation matrix $\boldsymbol{T}_{loc} \in \mathbb{R}^{m\times m}$ for each location $loc \in Loc$, such that:

$$\{\boldsymbol{x}_{virt,loc,i}\boldsymbol{T}_{loc} = \boldsymbol{x}_{\lambda,loc,i} \mid \boldsymbol{x}_{virt,loc,i} \in \boldsymbol{x}_{virt,loc},\, \boldsymbol{x}_{\lambda,loc,i} \in \boldsymbol{x}_{\lambda,loc}\} \tag{4.28}$$

This is done by finding the least square solution of the this linear system. Thus, $\boldsymbol{T}_{loc}$ is estimated using all sampled points $\boldsymbol{x}_{virt,loc}$ and $\boldsymbol{x}_{\lambda,loc}$ from both state spaces for a given location. For all three methods, all points of the guards at a given location are transformed from the $\mathcal{S}_{virt}$ to the $\mathcal{S}_\lambda$ space by using this estimated transformation. The *estT method* ends here, by transforming the guard points to the desired representation, that is by either hulling the guard points with the specified geometric shape, or by finding a halfspace description.

The *disT method* takes an additional step after finding the estimated guard points in the $\mathcal{S}_\lambda$ space. The problem with the *estT method* is that the estimated transformation does not yield good guards. It is desired to obtain guards that are as close as possible to the facets of the invariants. Therefore, the *disT method* was implemented which directly uses the invariants and replaces the estimated guard points with the closest invariant points for each location. This is done by computing the distance between the points of the invariant and the previous estimated guard points for each location. The closest invariant points are then taken as the new guard points. In the final step, these points are transformed to the desired guard representation. Note that, the points of the guards were first called estimated, as this method corrects them after finding them previously with the estimated transformation $\boldsymbol{T}_{loc}$.

The *halfT method* takes a different approach after finding the estimated guard points in the $\mathcal{S}_\lambda$ space. The problem of the *disT method* is that the points of the invariants can be close to each other. Calculating the guards based on these points can join different facet points of the invariant resulting in guards that instead of being on the facets pass through the invariant. To solve this problem, the *halfT method* was implemented, which yields the best results compared to the previous three. This method transforms the invariant of a location into a polytope $\mathcal{P}_{loc}$ in the halfspace representation (Theorem 2.1.1), that is for the k_h halfspaces of $\mathcal{P}_{loc}$:

$$\mathcal{P}_{loc} = \left\{\boldsymbol{x}_\lambda \in \mathbb{R}^m \mid \boldsymbol{C}_h\boldsymbol{x}_\lambda \leq \boldsymbol{d}_h,\, \boldsymbol{C}_h \in \mathbb{R}^{k_h\times m},\, \boldsymbol{d}_h \in \mathbb{R}^{k_h}\right\} \tag{4.29}$$

Each guard is then identified as one of the halfspaces of $\mathcal{P}_{loc}$. For that, consider a estimated single guard consisting of k_{grd} points. From each $\boldsymbol{x}_{\lambda,grd}$ of the k_{grd} points, the distance $\boldsymbol{d}_i$ to each of the k_h halfspaces of the invariant polytope ($\mathcal{P}_{loc}$) is calculated via:

$$\boldsymbol{d}_i = \frac{\mid \boldsymbol{C}_h \cdot \boldsymbol{x}_{\lambda,grd} - \boldsymbol{d}_h \mid}{\sqrt{\boldsymbol{C}_h \cdot \boldsymbol{C}_h}} \tag{4.30}$$

For the k_{grd} points, the total distance is then computed:

$$\boldsymbol{d}_{Total} = \sum_{i=1}^{k_{grd}} \boldsymbol{d}_i \tag{4.31}$$

This allows for the selection of a single halfspace which is closest to all k_{grd} points. This halfspace replaces then the estimated guard. Next the polytope $\mathcal{P}_{loc}$ is brought into the vertex representation as stated in Theorem 2.1.2. Using the previous found halfspace:

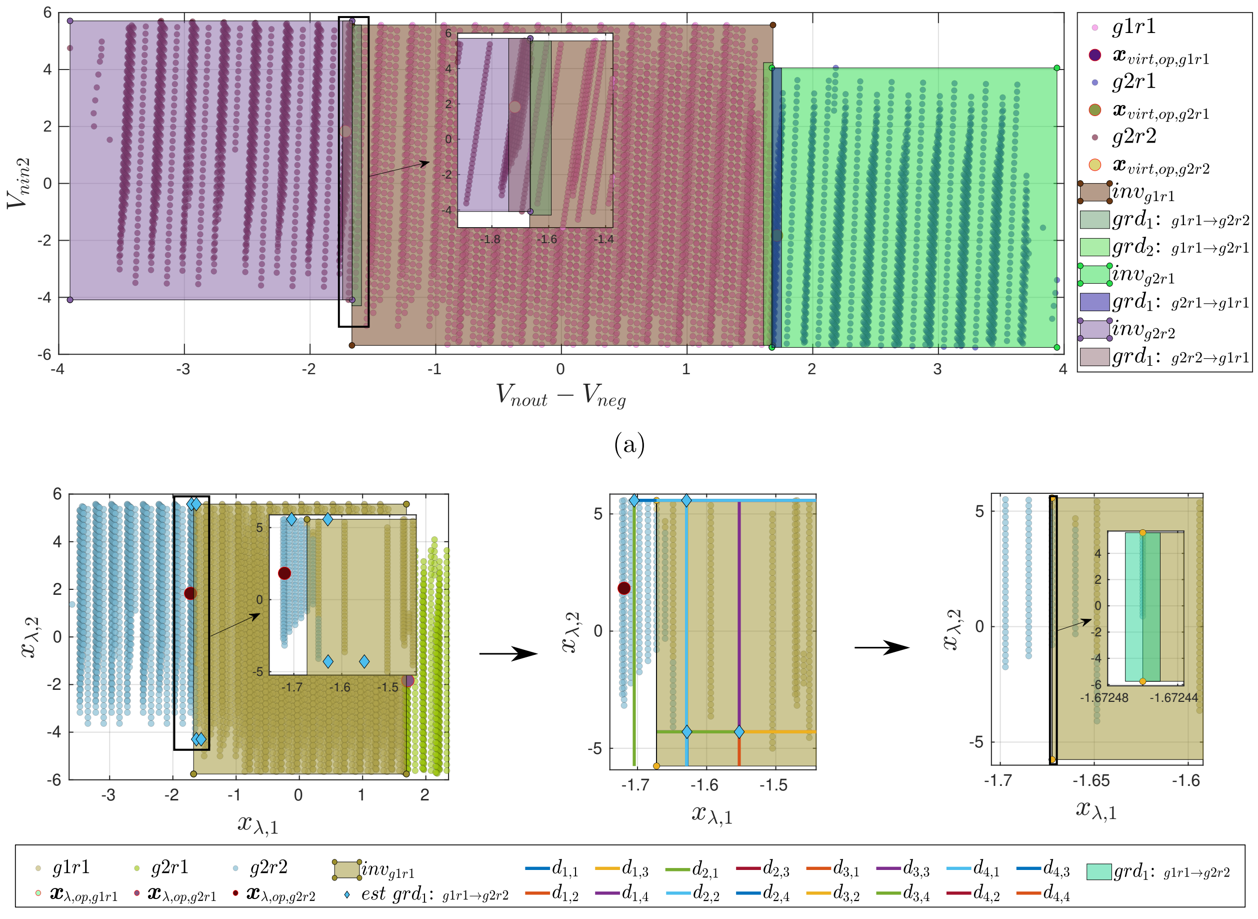
$$\mathcal{H} = \{\boldsymbol{x}_\lambda \mid \boldsymbol{c}_h^T \boldsymbol{x}_\lambda \leq d_h\},$$

the corresponding vertices of the invariants can be identified, as these points belong to the hyperplane describing the halfspace, thereby satisfying:

$$\boldsymbol{c}_h^T \boldsymbol{x}_\lambda = d_h$$

The vertices identified are finally transformed into the desired guard representation.

To keep things simple, for an invariant modeled as an interval hull, the results of the transformation of the first guard grd_1 of $g1r1$ to the $\mathcal{S}_\lambda$ space using the *halfT method* is presented in Fig. 4.35.



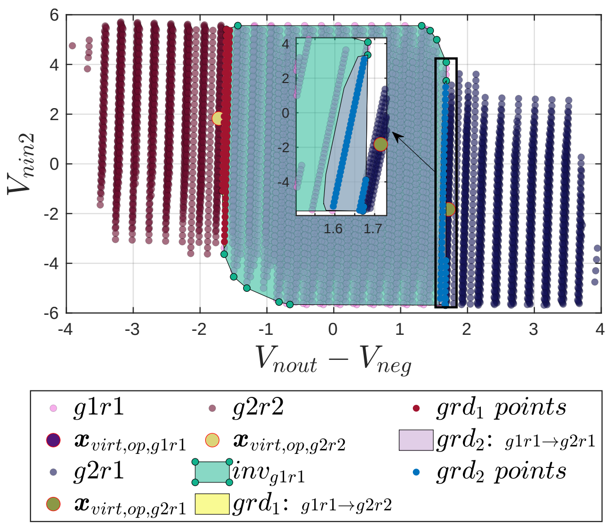
(b)

Fig. 4.35. Finding guard grd_1 of location $g1r1$ in the $\mathcal{S}_\lambda$ space using the *halfT method*.

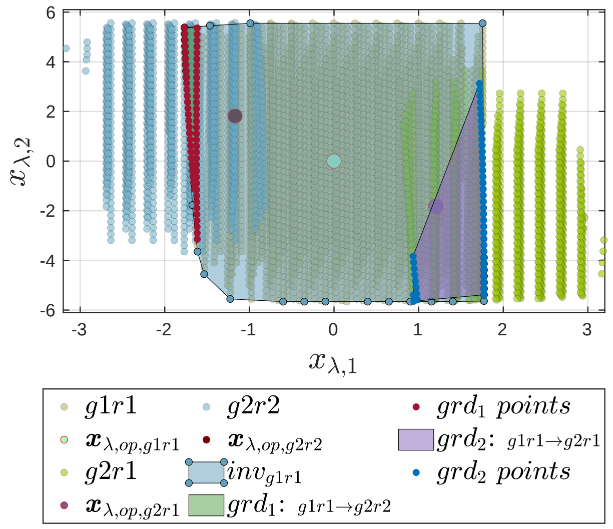
The *intersectionII method* was used to identify the guards in the $\mathcal{S}_{virt}$ space as illustrated in Fig. 4.35a. The guard grd_1 of location $g1r1$ was thereby described using $k_{grd_1} = 4$ points. These points where transformed into the $\mathcal{S}_\lambda$ domain using an estimated transformation matrix $\boldsymbol{T}_{g1r1}$. The estimated points are marked by diamonds colored in cyan in Fig. 4.35b. For each estimated point, the distances to all edges of the invariant inv_{g1r1} are computed, thereby denoting the distances by two subindcies, such that the first represents the estimated point, while the second represents the examined edge. For the four 4 points of the estimated guards, Eq. (4.30) is executed 4 times. The closest edge is then identified by computing first the total distance $\boldsymbol{d}_{Total}$ (Eq. (4.31)) along the k_{grd} points, followed by finding the minimum across the rows of $\boldsymbol{d}_{Total}$. The found halfspace can be directly used if the guard type is set to halfspace. If this is not the case, the vertices of the invariant are identified, and changed into the desired representation. For example, in Fig. 4.35b the vertices are used to model the identified guard points as an interval hull (Section 4.5.7). In this case, by computing the Minkowski sum of this guard with a tolerance value (tol_G), the guard attains a volume and can be represented in the desired shape. Usually the *halfT method* identifies several facets as guards, especially if the invariants are modeled as polytopes, thus several vertices of the invariants can be used for each guard (see Fig. 4.36c).

The reader might ask now why an approach similar to the invariant is not applied. In fact the *sampT method* takes this approach. This method does not estimate the guards and corrects them, rather it uses an approach similar to the invariants. First the points inside the guards are identified in the $\mathcal{S}_{virt}$ space. This identification is straight forward for guards modeled as polytopes, zonotopes, or interval hulls. On the other hand, this is also applicable for a guard of type halfspace. This lies in the fact that the points used to generate the halfspace can be used here. Using either of the $\mathcal{S}_\lambda$ space recalculation method (see Section 4.5.5), these points are then transformed into the $\mathcal{S}_\lambda$ space. Finally, the guards are modeled in the desired representation using the identified points. This method has a fallback in case no points were found inside or on the guards. Under such circumstances, the closest points are selected and the process continues as normal. As this method resembles the approach of transforming only the vertices of the invariants during the invariant identification, obviously since the sampled points inside a location have different transformation matrices, the results are not always optimal.

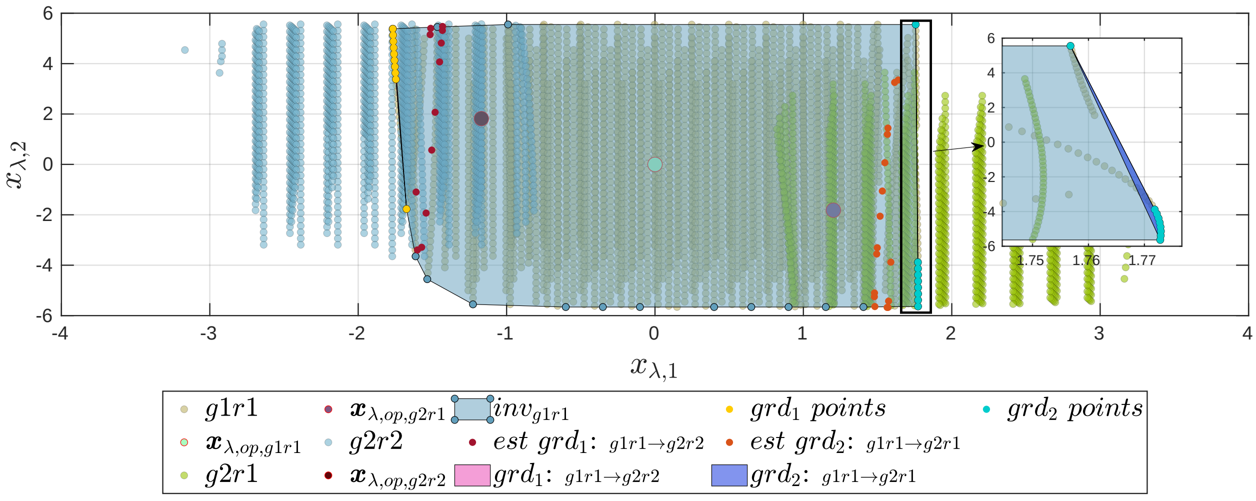
For the running example, the result of the guard transformation is presented in Fig. 4.36, for simplicity, by regarding only location $g1r1$. Using the *intersectionII method* in the $\mathcal{S}_{virt}$ space, the guards of the center location $g1r1$ were found. The invariant as well as the guards were model as polytopes, as illustrated for $g1r1$ in Fig. 4.36a. The red and blue points in this figure indicate the points hulled by the two polytopic guards grd_1 and grd_2 of $g1r1$, respectively. If these points were transferred to the $\mathcal{S}_\lambda$ domain using the *sampT method*, the resulted points are shown in Fig. 4.36b. The *sampT method* first transforms these points into the $\mathcal{S}_\lambda$ domain, then hulls them with the specified guard type. In this case, the guards are specified as polytopes. As observed, the obtained guards in the $\mathcal{S}_\lambda$ domain (Fig. 4.36b), especially grd_2, cover a large portion of the invariants, even though concise guards were found in the $\mathcal{S}_{virt}$ space. Transforming the same guards identified in the $\mathcal{S}_{virt}$ space with the *halfT method* yields much better results as presented in Fig. 4.36c. This method first estimates the guard points by using the transformation matrix $\boldsymbol{T}_{g1r1}$ and the guard point from the $\mathcal{S}_{virt}$ domain. Then the estimated points, red for grd_1 and orange for grd_2, are mapped to the corresponding halfspaces of the invariant inv_{g1r1}, identifying thereby the halfspaces that qualify as



(a)



(b)



(c)

Fig. 4.36. In (a) the result of the guard identification using the *intersectionII method* in the $\mathcal{S}_{virt}$ space for only the first location $g1r1$ is presented for an invariant and guards specified as polytopes. The result of transforming the two guards of location $g1r1$ is illustrated in (b) by using the *sampT method*, and in (c) by using the *halfT method*.

guards. Moreover, since the guards are specified as polytopes, the vertices of invariant lying on the halfspaces are used to form the new guards, colored yellow for grd_1 and cyan for grd_2. These vertices are then used to form the new guards in the $\mathcal{S}_\lambda$ domain. As observed in Fig. 4.36c, the results are by far superior and preferable to the results of the *sampT method* (see Fig. 4.36a). Moreover, the guards obtained using the *halfT method*, always lie on the borders of the invariants. Therefore, the *halfT method* is used as the default method unless specified otherwise.

4.5.9 Jump Functions

The last block in the system modeling process presented in Fig. 4.12, is the jump function block. This block can be skipped, but usually modeling the abstracted systems with jump functions results in superior behavioral models. This is especially true for the case the locations do strongly overlap in the $\mathcal{S}_\lambda$ space compared to the $\mathcal{S}_{virt}$ space. Note that the overlapping of the $\mathcal{S}_\lambda$ space can be controlled to some extent by the sampling method of *Vera*, as later examined in Section 7.1.2.

The DC points are a good indicator if the jump functions are necessary. As observed in Fig. 4.13a, the DC points in the $\mathcal{S}_{virt}$ space inside a location can be mapped to a single line as illustrated in Fig. 4.37. However, for the current sampled $\mathcal{S}_\lambda$ space illustrated in Fig. 4.13b, this is not possible.

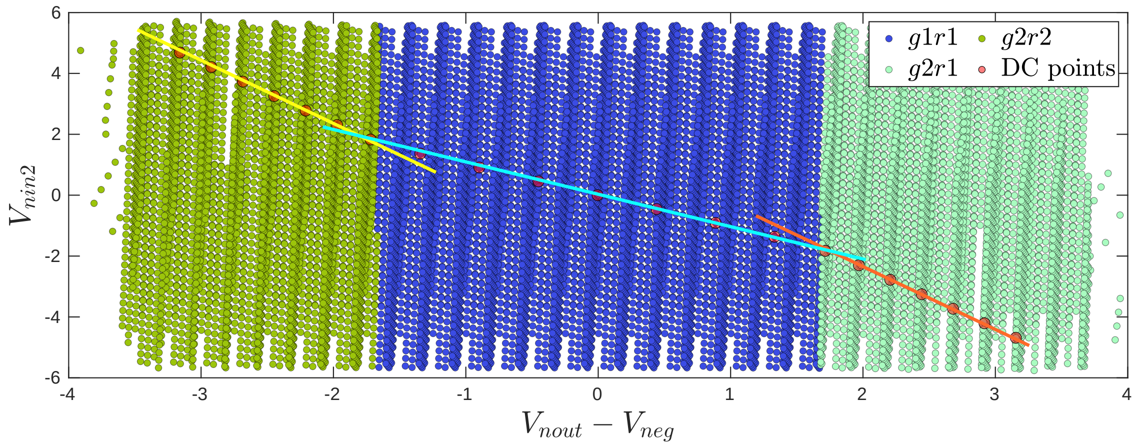


Fig. 4.37. Analyzing the $\mathcal{S}_{virt}$ space. Three lines are required to cover all DC points.

The number of lines with different slopes needed to join the DC points is a good indicator for the number of locations at least needed, as for a linear SISO system the operating points obtained at different inputs usually lie on one line. However, even though this can be extended to MIMO systems to some extent using planes for example, this approach cannot be generalized, especially for system with defective system descriptions, as systems exist that can have several operating points for the same input stimuli.

In general, the overlapping of the locations in the $\mathcal{S}_\lambda$ space proposes a problem for the simulation as well as for the location distinction. This problem can be solved by using jump functions and adjusting the guards and invariants in $\mathcal{S}_\lambda$ space. In the instance the abstract model is described with jump functions, several aspects need to be modified. The jump functions are linked to guards (Definition 1). Once a guard is taken and thus a state transition occurs, the corresponding jump function is applied. In Section 2.4, the jump function of a hybrid automaton was described as:

$$\boldsymbol{x}_{\lambda,new} = \boldsymbol{Q}_r \boldsymbol{x}_{\lambda,old} + \boldsymbol{v}_r \, , \tag{4.32}$$

where $\boldsymbol{Q}_r \in \mathbb{R}^{m \times m}$ is the mapping matrix, $\boldsymbol{v}_r \in \mathbb{R}^m$ is the reset vector, and $\boldsymbol{x}_{\lambda,new} \in \mathbb{R}^m$ and $\boldsymbol{x}_{\lambda,old} \in \mathbb{R}^m$ are the new state vector after and before the transition, respectively. For the current application, Eq. (4.32) is changed to:

$$\boldsymbol{x}_{\lambda,new} = \boldsymbol{I}_r \boldsymbol{x}_{\lambda,old} - \boldsymbol{x}_{\lambda,op,loc_j} \, , \tag{4.33}$$

with $\boldsymbol{I}_r \in \mathbb{R}^{m \times m}$ and $\boldsymbol{x}_{\lambda,op,loc_j} \in \mathbb{R}^m$ representing an identity matrix and the operating point of a target location loc_j in the $\mathcal{S}_\lambda$ space, respectively. Thus, whenever a guard is taken from the current location to the target location, Eq. (4.33) is applied by subtracting the operating point of the target location from $\boldsymbol{x}_\lambda$.

In addition to the previous mentioned adjustment of the state vector $\boldsymbol{x}_\lambda$, the guards as well as the invariants are shifted. Both objects are shifted by the current operating point. This corresponds to subtracting the operating point of the current location in the $\mathcal{S}_\lambda$ space from these objects, thereby moving these objects to the global reference point (Section 4.5.1) of the $\mathcal{S}_\lambda$ space, as shown in Fig. 4.38.

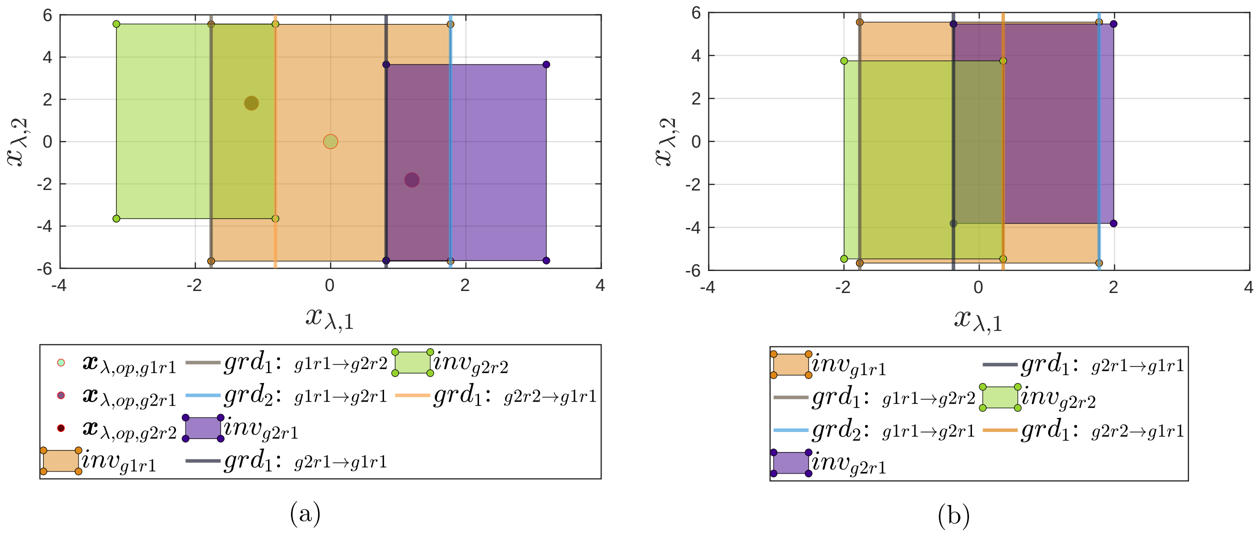


Fig. 4.38. Adjusting the invariants and guards in the $\mathcal{S}_\lambda$ space for the use with the jump functions. The found results (a) are shifted by the corresponding operating to obtain (b).

Fig. 4.38a shows the previous calculated guards and invariants in the $\mathcal{S}_\lambda$ domain. These objects were shifted by subtracting the corresponding operating points as observed in Fig. 4.38b. For example, considering $g2r1$, the invariant inv_{g2r1} and the guard grd_1 from $g2r1$ to $g1r1$ are shifted by subtracting $\boldsymbol{x}_{\lambda,op,g2r1}$ from both objects. Note that this is only applied for the Matlab (*Cora*) models as explained in Section 4.6.1.

By these modifications, the overlapping of the locations can be controlled. Even though the obtained results in Fig. 4.38b seem to be overlapping, they are in fact separated by the location variable $loc \in Loc = \{g1r1, g2r1, \ldots\}$. For the running example, three locations have been identified, thereby $Loc = \{g1r1, g2r1, g2r2\}$. Fig. 4.39 shows how the location variable allows for a clear distinction of the invariants and guards.

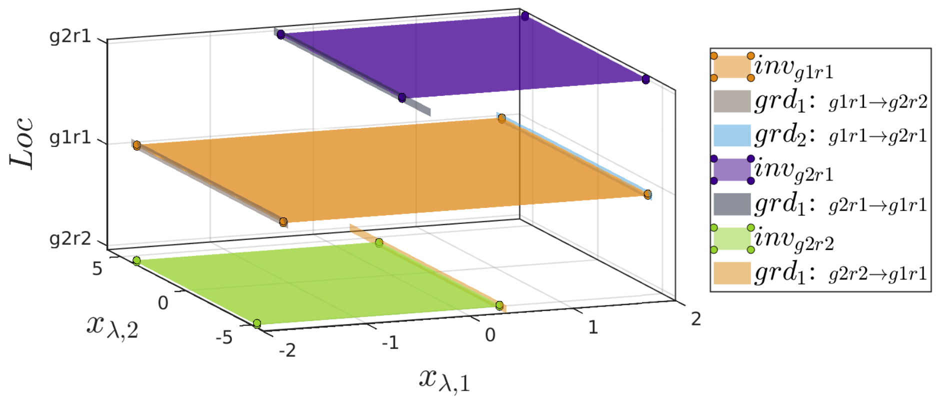


Fig. 4.39. Result of the guards and invariants after adjustment illustrated against the locations.

For the current example, the system starts at the center location $g1r1$. If the system does not leave inv_{g1r1}, the system behavior is characterized by Eq. (4.10) and the operating points and input voltages of the current location. In case the simulation terminates while the system does not leave this location, the back transformation Eq. (4.11) uses only the operating values of the current location as well. However, for a simulation in which the system intersects either on of the guards grd_1 or grd_2 of location $g1r1$, the system performs a transition to either location $g2r2$ or $g2r1$, respectively. In both case the jump function is activated, and the system behavior is shifted by subtracting the value of the target operating point as specified by Eq. (4.33). In the new location, the simulation is continued until either a guard is intersected, the invariant is left, or the simulation time is elapsed. In the two latter cases the simulation is terminated, while in the first case a location transition again occurs. The results of the performed simulation or reachability analysis are transformed from the $\mathcal{S}_\lambda$ space back to the original $\mathcal{S}_o$ space for each visited location separately by using the corresponding operating values in Eq. (4.11).

To analyze the result, a phase diagram illustrated in Fig. 4.40 has been created for a constant input voltage $V_{nin} = 4$ V. At the specified input voltage, the operating amplifier goes into saturation thus changing the behavior of the system from a linear to a limiting nonlinear one. For the abstract HA, this can be understood as changing the location from $g1r1$ to $g2r2$. As observed in Fig. 4.40, for the system starting at location $g1r1$, the steady state point $\boldsymbol{x}_{\lambda,ss,g1r1}$ lies outside the invariant.

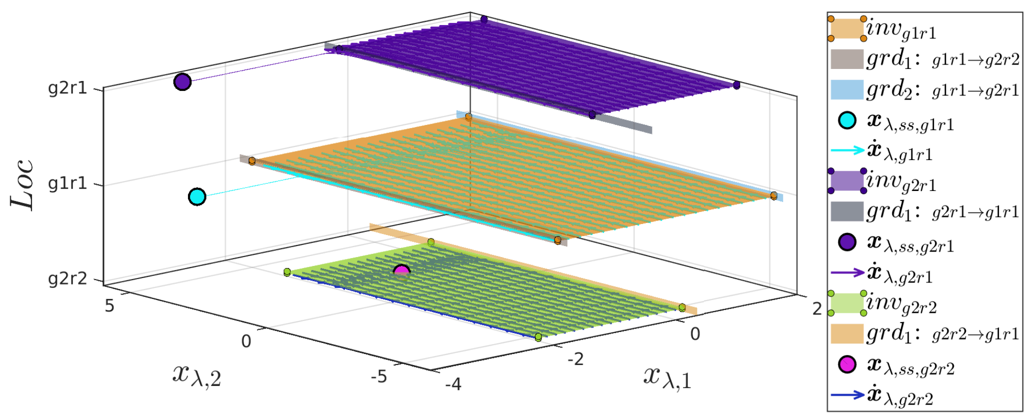


Fig. 4.40. Phase diagram of the created HA in the $\mathcal{S}_\lambda$ space with adjusted locations for a constant input $V_{nin} = 4$ V illustrated along the location variable Loc.

Specifically, reaching this point from $g1r1$ intersects the guard grd_1 before reaching $\boldsymbol{x}_{\lambda,ss,g1r1}$. In this case, the HA performs a jump to $g2r2$ at the instance the intersection occurs. In $g2r2$ the system tries to reach $\boldsymbol{x}_{\lambda,ss,g2r2}$. As this point lies inside the invariant inv_{g2r2}, the system reaches this point and stays there. Note that, in this case $g2r1$ is not visited.

To gain more insight on the created HA, a reachability analysis was performed with *Cora* for the same input voltage. The result in the $\mathcal{S}_\lambda$ space is illustrated in Fig. 4.41. As observed, the reachability analysis starts at the center location $g1r1$. The reachable set $\mathcal{R}_{g1r1}$ intersects grd_1 at $x_{\lambda,1} = -1.767$. The system performs then a jump to $g2r2$ continuing the reachability analysis with $\mathcal{R}_{g2r2}$ until the steady state is reached. The result of this analysis is finally transformed back into the $\mathcal{S}_o$ space using the corresponding back-transformation (Eq. (4.11)) for each location. Fig. 4.42 shows one of the $\boldsymbol{x}$ variables which corresponds to the output voltage V_{nout}. As observed, the jump

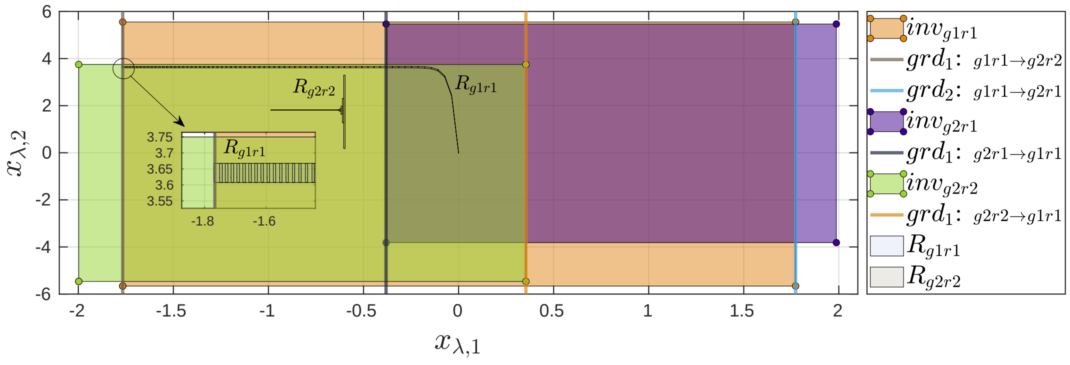


Fig. 4.41. Result of the reachability analysis in the $\mathcal{S}_\lambda$ space for a constant input at $V_{nin} = 4$ V.

occurred at the time $t = 0.067$ s. Note that the reachable sets are in fact illustrated as intervals which bound the upper and lower bounds of the computed zonotopes (stored in $\boldsymbol{x}$) along with a specified time step of 0.001 s. As observed, the HA exhibits a limiting behavior. A comparison of

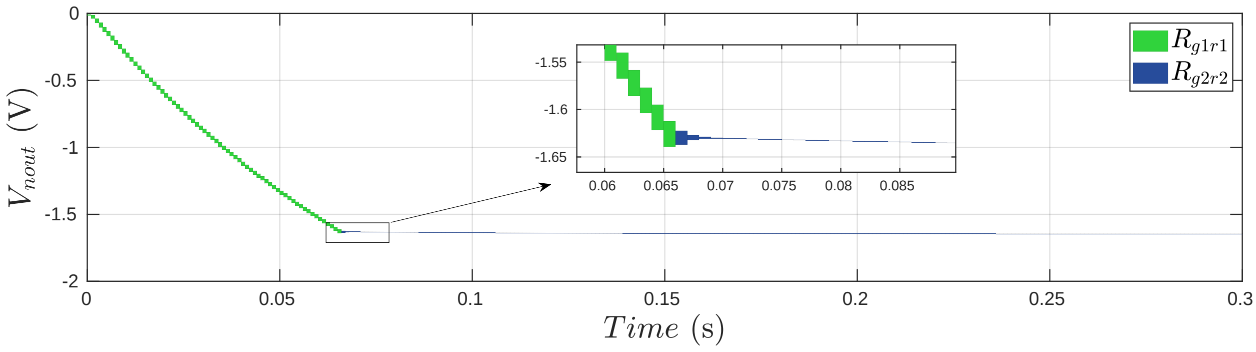


Fig. 4.42. Result of the back-transformation of the reachability results in the $\mathcal{S}_o$ domain. The element of $\boldsymbol{x}$ corresponding to the output voltage V_{nout} is illustrated versus time.

the generated HA and the netlist is presented in Section 7.1

If the system is modeled without jump functions, skipping thereby the 8th block in Fig. 4.12, the adjustments performed in this section on the guards and invariants in the $\mathcal{S}_\lambda$ space are skipped. Moreover, the simulations are performed directly on the variables instead of their delta values, thereby removing the operating points $\boldsymbol{x}_{op}$, $\boldsymbol{x}_{\lambda,op}$, and the operating input voltage $\boldsymbol{u}_{op}$ from Eqs. (4.10, 4.11) for each location. However, in this case the invariants found previously must overlap at least slightly in the $\mathcal{S}_\lambda$ space. If this is not the case, spaces between the invariants can lead the system during transitions to leave all locations, thereby terminating the simulation. Generally speaking, a HA modeled with jump functions yields significantly better results than a HA modeled without jump functions. A detailed comparison of the running example modeled with and without jump functions is presented in Section 7.1.5. Note that this section described the application of the jump functions on the Matlab models. For the Verilog-A and SystemC-AMS models, the jump functions applies a shift and not a reset on the system as explained in Section 4.6.

4.6 Model Creation

The last block in Fig. 4.1, and thereby the final step in the model abstraction process, is the model creation. According to the specified output language, three different types of models can be created: Matlab, Verilog-A, or SystemC-AMS models. For the Matlab models a reachability analysis can be performed using the reachability tool *Cora*, while simulations can be performed with the remaining two types using standard Verilog-A and SystemC-AMS simulator, respectively. Additionally, the SystemC-AMS models can be extended for performing affine arithmetic simulations with the standard simulator, yielding bounded results similar to a reachability analysis.

In the following, the model generation in the specified output language and corresponding modeling methodology will be handled. Six properties need to be specified for the generated models:

(a) the system behavior according to Eq. (4.10), specified by the system matrix $\boldsymbol{A}_{loc} \in \mathbb{R}^{m\times m}$, the input matrix $\boldsymbol{B}_{loc} \in \mathbb{R}^{m\times k}$, the operating points $\boldsymbol{x}_{op} \in \mathbb{R}^{n}$ and $\boldsymbol{x}_{\lambda,op} \in \mathbb{R}^{m}$, and the operating input $\boldsymbol{u}_{op} \in \mathbb{R}^{k}$

(b) the invariant of location

(c) the guards that allow for transitions to the neighbor locations

(d) the location id $loc \in Loc = \{g1r1, g2r1, \dots\}$, specifying the current location of the HA

(e) the jump functions corresponding to the guards if necessary

(f) the back-transformation according to Eq. (4.11), specified by the transformation matrices $\boldsymbol{F}_{loc} \in \mathbb{R}^{n\times m}$ and $\boldsymbol{L}_{loc} \in \mathbb{R}^{n\times k}$

Depending on the model type, the properties are embedded differently into the model description.

4.6.1 Matlab Models

The detailed generation of a Matlab model has been handled thought this chapter in detail. Each location of the HA is described by a location class. The location class hosts:

(a) the system description: $\boldsymbol{A}_{loc}$ and $\boldsymbol{B}_{loc}$ specified in general as matrices. The system description can be also specified using matrix zonotopes or interval matrices to model process parameter variations (Section 5.2) or compensate for the abstraction process (Section 5.1.1)

(b) the invariant: in any of the in Section 4.5.6 specified geometric shapes

(c) the guards: in any of the in Section 4.5.7 specified geometric objects

(d) the location id: loc, which specifies the current location of the HA

(e) the jump functions: specified by the *reset* variable hosting the matrix $\boldsymbol{Q}_r$ and the reset vector $\boldsymbol{v}_r$, and the target location. Note that the guards are saved, along with their corresponding jump functions and target locations, in the transition variable of each location.

(f) the back-transformation: $\boldsymbol{F}_{loc}$ and $\boldsymbol{L}_{loc}$ specified in general as matrices

For each location, a location class element is initialized and passed to a cell array named *loc*. Finally, the hybrid automaton is initialized by passing this cell array to the hybrid automaton class (*hybridAutomaton*). Unlike the SystemC-AMS and Verilog-A models, the result of the reachability analysis of the Matlab (*Cora*) models is first completely computed, followed by transforming (post analysis) the found results from the $\mathcal{S}_\lambda$ space to the $\mathcal{S}_o$ space via the back-transformation. For the SystemC-AMS and Verilog-A models, the results in the $\mathcal{S}_o$ space are computed during the simulations. In Listing D.1, an example of a Matlab model is stated. In case the system is modeled with jump functions, the invariants and guards are adjusted according to Section 4.5.9.

4.6.2 Verilog-A Models

Unlike *Cora*, the Verilog-A simulator performs a simulation and not a reachability analysis. Moreover, a clear condition must be stated when the transition occurs for the Verilog-A models, thereby forcing the generated HAs to be deterministic. Additionally, due in to the behavior of the Verilog-A simulator and the absence of the digital states, a different approach compared to the methodology of the Matlab models must be used to identify the current location of the system during simulation. For this purpose, two methods arise that can be used to model a HA in Verilog-A:

grdV method discard the invariants and use only the guards to define the current location and the transitions to the next locations

invV method discard the guards and use only the invariants to define the current location and the transitions to the next locations

The *grdV method* uses only the guards to define the transitions between the locations. Moreover, using the $\boldsymbol{x}_\lambda$ variables along with the guards and optionally the jump functions, the current location of the HA is determined.

The guards are modeled as if conditions. For halfspace guards, this is straight forward. For the reaming guard types, the geometric shapes are transformed to their halfspace representation, followed by modeling selected halfspaces as if conditions. The concept of selecting the appropriate halfspaces is similarly to the approach of extracting the relevant halfspaces from $\mathcal{P}_I$ in the *intersection method* from Section 4.5.7 (see Fig. 4.30).

To illustrate the model behavior of a HA in Verilog-A, consider Fig. 4.43. A HA was generated for the running example with invariants modeled as interval hulls and guards specified as halfspaces using the *distance method*. Moreover, the HA is modeled with jump functions, and described in

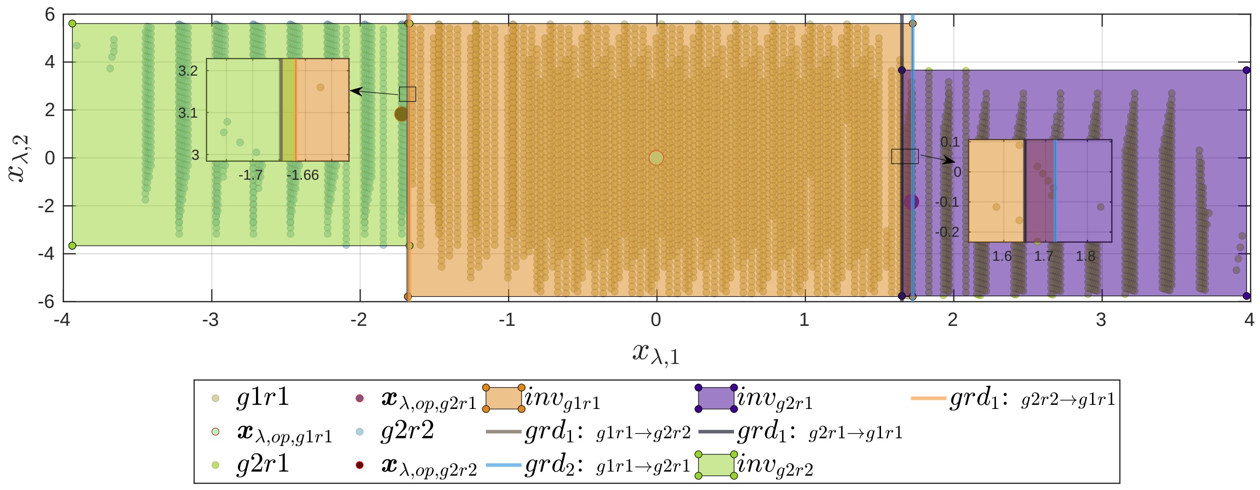


Fig. 4.43. The $\mathcal{S}_\lambda$ space with the $\boldsymbol{x}_\lambda$ stated variable as handled by the Verilog-A simulator.

Verilog-A using the *grdV method*. Even though the HA was modeled with jump functions, the locations are not adjusted in Fig. 4.43, in contrast to the Matlab model shown in Fig. 4.39. This is due to the fact that the simulator uses the $\mathcal{S}_\lambda$ space as illustrated in Fig. 4.43, with the $\boldsymbol{x}_\lambda$ *s*hifted during simulation and not *r*eset as for the Matlab models. More precisely, the $\boldsymbol{x}_\lambda$ variables are constantly shifted by a shift vector:

$$\boldsymbol{v}_s = \boldsymbol{x}_{\lambda,op,loc}\,, \tag{4.34}$$

that is set to the operating point of the current location. Hence, this vector is subtracted from the $\boldsymbol{x}_\lambda$ variables in all equations, as will be clarified through this section.

Fig. 4.44 shows how the current location can be identified from the $\boldsymbol{x}_\lambda$ variables, guards, and jump functions. Even though the invariants are ignored and irrelevant for the system behavior due to the modeling method, their are illustrated in Fig. 4.44 for consistency. During a simulation, Verilog-A simulator performs for every time step several iterations. At each time step, the system starts from the center location as observed in Fig. 4.44. The HA stays in the center location as long

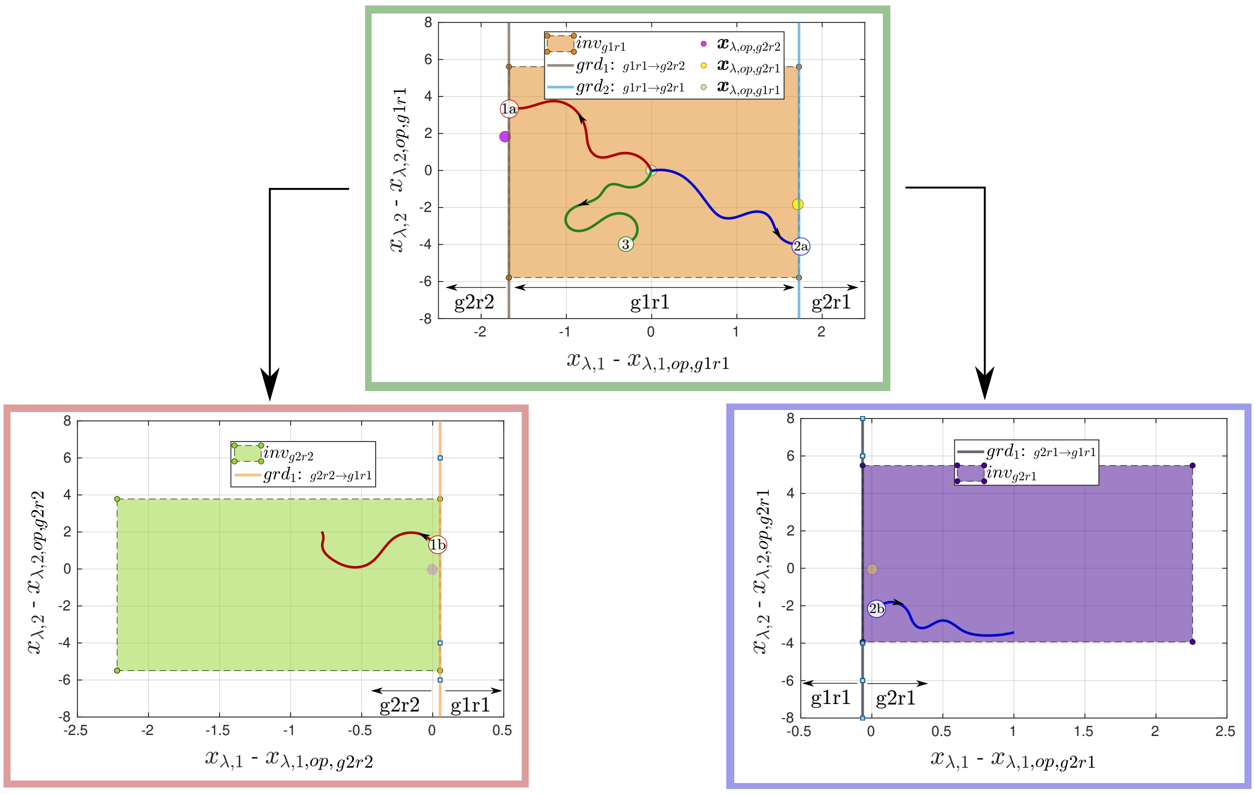


Fig. 4.44. Determining the current location using the $\boldsymbol{x}_\lambda$ variables, guards, and jump functions. From location $g1r1$, the system can either stay in this location, or transition to the next locations if a halfspace guard is intersected.

as the guards to the neighbor locations are not intersected. An example of a trajectory is colored in green with the final destination point labeled 3. Once the trajectory in the $\mathcal{S}_\lambda$ space intersects a guard of $g1r1$, a location transition occurs. For example, if the red trajectory is considered, the HA intersects grd_1 of $g1r1$ at the point labeled $1a$. At this instance, the HA switches the location to $g2r2$. Since the HA is modeled with jump functions, the operating point $x_{\lambda,op,g2r2}$ of $g2r2$ is subtracted from the intersection point $1a$ (see Eq. (4.33)). With the obtained value $1b$, the system continuous the simulation in the new location $g2r2$. As long as grd_1 from $g2r2$ is not intersected, the HA stays in $g2r2$. Specially, as long as the HA stays to the left or grd_1 of $g2r2$, the system stays in $g2r2$. Similarly, the blue trajectory shows a possible transition to $g2r1$ from $g1r1$. Note that the shift values are as well subtracted from the guards and invariants.

During the iterations in each time step, the HA is faced with the decision of the current location. Starting from the center location, the possible intersections of the guards are analyzed. In case an intersection occurs, the location is switched and the guards of the new location are analyzed. In case no intersections occur, the current location is obtained. Algorithm 8 states the basic description of the HA in Verilog-A for the running example using the *grdV method*.

Algorithm 8 Verilog-A model generated with the *grdV method*

```
1: module hybridAutomaton($V_{nin}, V_{nout}$)
2:     initialize the variables
3:     set or unset the debug flag
4:     identify the current location loc
5:     issue a warning if debug is set and the current invaraint is left
6:     assign all variables according to loc
7:     solve nodal equations to find all $V(x_{\lambda,i})$
8:     perform the back-transformation using the previous found voltages
9: end module
```

As stated in Algorithm 8, the variables are first initialized. Since Verilog-A does not support matrices, all elements of the matrices are initialized as real variables, except the input and output voltages, V_{nin} and V_{nout} along with the $\mathcal{S}_\lambda$ state space variables, $x_{\lambda,1}$ and $x_{\lambda,2}$, which are initialized as electrical variables. At line 4, the HA finds the current location as described in Algorithm 9. At line 3, a *debug* flag can be set, that issues a warning at line 5 after the current invariant is left. Note that the invariants are described using their halfspace representation. Thus, only the guard conditions decide when a location is left, while the invariants can be used to issue warnings. Based on the current location identified at line 4, the elements of all matrices and vectors are assigned according to their corresponding values at this location. This includes the system and input matrices, the transformation matrices, and the operating values including the shift vector of the jump function. With the new assigned values, the simulation continues at line 7 by solving two nodal equations for the states $V(x_{\lambda,1})$ and $V(x_{\lambda,2})$. For the first state, the nodal equation is:

```
I(x_lam1) <+ -1*scale*ddt(V(x_lam1))   ;
I(x_lam1) <+ scale*(A11*(V(x_lam1) - xShift1) + A12*(V(x_lam2) - xShift2)
               + B11*(V(nin) - uDc));
```

With xShift1 and xShift2 representing the elements of the shift vector $\boldsymbol{v}_s$ for the $m = 2$ dimensional system, set according to the operating point of the current valid location loc.

$$\underbrace{\begin{bmatrix} xShift1 \\ xShift2 \end{bmatrix}}_{\boldsymbol{v}_s} = \boldsymbol{x}_{\lambda,op,loc} \,,$$

and uDc representing the operating input voltage at $\boldsymbol{x}_{\lambda,op,loc}$ used accordingly with the shift vector. Note that the shift vector $\boldsymbol{v}_s$ describes the jump function in each location. Specifically, instead of describing the jump function with Eq. (4.32) along with the reset vector $\boldsymbol{v}_r$ only once a transition occurs, the jump function is described by a shift vector $\boldsymbol{v}_s$ which is constantly subtracted from the $\boldsymbol{x}_\lambda$ variables in the equations of the guards, invariants, back-transformation, and as previously seen from the nodal equations. This vector does not change the values of the $\boldsymbol{x}_\lambda$ variables as the reset vector does, but constantly shifts them by a fixed vector ($\boldsymbol{x}_{\lambda,op,loc}$) for each location. Hence, the geometric objects in the $\mathcal{S}_\lambda$ space in Fig. 4.43 are not adjusted.

According to Eq. (4.11), the found voltages are transformed back into the $\mathcal{S}_o$ domain at line 8 for all $\boldsymbol{x}$ variables. For example, the voltage at the negative terminal of the operation amplifier (see Fig. 4.2) is calculated via:

```
X_neg = xDc22 + F221*(V(x_lam1) - xShift1) + F222*(V(x_lam2) - xShift2)
            + FooEoob221*(V(nin) - uDc);
```

This process is repeated for every iteration during each time step. As observed in Algorithm 8, the values in the $\mathcal{S}_o$ domain are calculated during the simulation in the $\mathcal{S}_\lambda$ space, in contrast to the Matlab (*Cora*) models.

Based on the $\boldsymbol{x}_\lambda$ variables along with the guards and optionally the jump functions, the current location *loc* is identified as stated at line 4 of Algorithm 8. In Algorithm 9, this process is described for the running example.

Algorithm 9 Identifying the current location *loc* using $\boldsymbol{x}_\lambda$, the guards, and the jump functions

```
 1: module $loc = currentLoc(x_\lambda, x_{\lambda,op}, guards)$
 2:     $loc = 0$
 3:     set shift vector $\boldsymbol{v}_s$ to $\boldsymbol{x}_{\lambda,op,g1r1}$
 4:     if shifted guard from $g1r1$ to $g2r2$ is valid then
 5:         $loc = 22$
 6:         set shift vector $\boldsymbol{v}_s$ to $\boldsymbol{x}_{\lambda,op,g2r2}$
 7:         if shifted guard from $g2r2$ to $g1r1$ is valid then
 8:             $loc = 11$
 9:         end if
10:     else if shifted guard from $g1r1$ to $g2r1$ is valid then
11:         $loc = 21$
12:         set shift vector $\boldsymbol{v}_s$ to $\boldsymbol{x}_{\lambda,op,g2r1}$
13:         if shifted guard from $g2r1$ to $g1r1$ is valid then
14:             $loc = 11$
15:         end if
16:     else
17:         $loc = 11$
18:     end if
19: end module
```

As observed in Algorithm 9, based on the current location (*loc*), the guards are shifted by the current operating point $\boldsymbol{x}_{\lambda,op,loc}$. For example, the guard of $g1r1$ to $g2r2$ at line 4 is adjusted:

```
if 1*(V(x_lam1) - xShift1) + 0*(V(x_lam2) - xShift2) < -1.673
```

The if conditions at lines 7 and 13 of Algorithm 9 can be skipped, which is similar to removing the guards from $g2r1$ and $g2r2$ to $g1r1$ in Fig. 4.44. On one hand, this brings the disadvantage of not examining the correctness of the behavior after a transition. In this case, the guards of the current location are favored over the guards of the target locations. However, this facilitates the convergence of the simulator during an iterations, especially in the case the guards of different locations are overlapping. On the other hand, in the presence of these conditions, the guards of the target locations are favored over the guards of the current location. This is due to the fact that in each iteration, the HA's final decision concerning the current location is specified by the validity of the guards in the target location in case a transition occurred.

The *grdV method* is prone to errors if the guards were not modeled directly as halfspaces, as not the optimal facets might be selected, especially when the *intersection method* was used for the guards identification. Moreover, as the behavior inside a location is unbounded due to the absence of the invariants of the system, the HA might reach unsampled portions in the state space.

Therefore, to solve the first problem, the guards should always be modeled as halfspaces if this method is used. Moreover, to solve the second problem, the *debug* flag can be set to issue a warning once the system leaves an invariant, as stated at line 5 of Algorithm 8. On the other hand, an even greater problem occurs with the iterations Verilog performs. Due to the structure of the code in Algorithm 8, the simulator can fall into the problem of switching constantly between two locations during a single simulation run, which leads to convergence issues terminating the simulation. This is often the case when the guards of the different locations are not well chosen i.e. they lie over each other. This problem can be solved for example by enlarging the invariants till they intersect slightly, in case they did not intersect prior to the identification of the guards, or by skipping the if conditions at lines 7 13 of Algorithm 9. Note that the *grdV method* yields the best models in case the guards were identified by the *distance method* and modeled as halfspaces. Moreover, as the guards are found from the invariants, polytopic invariants should be used.

The *grdV method* works with both variants, in the presents as well as in the absence of the jump functions. In the case the system was modeled without jump functions, the invariants must overlap, which can be easily achieved by finding the guards and invariants directly in a well sampled $\mathcal{S}_\lambda$ space, and slightly enlarging the invariants if necessary. In this case, the guards, nodal equations, and the back-transformation are not shifted. That is, the model is not adjusted by the jump functions. Note that better results are usually obtained in the presence of the jump functions. An example of the Verilog-A model is stated in Listing D.2 in Appendix D.

As mentioned, the selection of the guard facet, in the case the guards are not modeled as halfspaces, as well as the absence of the invariant might inject errors into the obtained model. Therefore, the *invV method* was implemented. This method neglects the guards and uses the invariants to define the transitions. Moreover, as the invariants can be in general represented as polytopes, and as the polytopes can be represented using the halfspace representation (Theorem 2.1.1), invariants can be precisely defined by logical linked if conditions. For that, regardless of the specified invariant type, the invariants are brought into their halfspace representation. Each halfspace represents an if condition. In general, the polytope is then described by liking the if conditions with logical ANDs. The complete model description is listed in Algorithm 10 for the running example.

After the initialization of the variables, the current location *loc* is found based on the invariants. The invariants are realized using if conditions on lines 3, 5, and 7. Hence, each condition can only become true if the voltages $V(x_{\lambda,1})$ and $V(x_{\lambda,2})$ lie inside the corresponding invariant. Note that these conditions are formulated similarly to line 5 of Algorithm 8 (see line 236 in Listing D.2).

Algorithm 10 Verilog-A model generated with the *invV method*

```
1: module $hybridAutomaton(V_{nin}, V_{nout})$
2:     initialize the variables
3:     if $(loc == 11 \,||\, loc == 0)$ $\&\&(V(x_{\lambda,1}) - x_{\lambda,op,g1r1,1}), (V(x_{\lambda,2}) - x_{\lambda,op,g1r1,2}) \in inv_{g1r1}$ then
4:         $loc = 11$
5:     else if $(loc == 21 \,||\, loc == 0)$ $\&\&(V(x_{\lambda,1}) - x_{\lambda,op,g2r1,1}), (V(x_{\lambda,2}) - x_{\lambda,op,g2r1,2}) \in inv_{g2r1}$ then
6:         $loc = 21$
7:     else if $(loc == 22 \,||\, loc == 0)$ $\&\&(V(x_{\lambda,1}) - x_{\lambda,op,g2r2,1}), (V(x_{\lambda,2}) - x_{\lambda,op,g2r2,2}) \in inv_{g2r2}$ then
8:         $loc = 22$
9:     else
10:        find closest location by computing min. of $\sqrt{\sum_{i=1}^{m}(V(x_{\lambda,i}) - x_{\lambda,op,loc,i})}$
11:        assign $loc$
12:    end if
13:    Assign all variables based on the current location $loc$
14:    solve nodal equations to find all $V(x_{\lambda,i})$
15:    perform back-transformation using the previous found voltages
16: end module
```

These conditions are extended by a query regarding the *loc* variable. This forces the simulator to stay in a location during the iteration performed in a single time step. During the first iteration of each simulation run, *loc* is set to zero and is thereby only assigned if the conditions concerning the invariants become valid. Thus, after the first run, the system stays in the found location as long as the invariant condition is true. Once this condition becomes false, the system tries to find the closest operating point in the $\mathcal{S}_\lambda$ space, by finding the minimum distance from the current point to the operating points of all locations. The location variable *loc* is then assigned according to the found operating point. With *loc* at hand at line 13, the elements of all matrices and vectors are assigned to their corresponding values at the location. At this point, the two nodal equation are formulated, to find the values of the state variables $V(x_{\lambda,1})$ and $V(x_{\lambda,2})$. Finally, the found voltages are transformed back into the $\mathcal{S}_o$ domain.

As observed in Algorithm 8 and Algorithm 10, unlike the Matlab models, the back-transformation in the Verilog-A models happens at each simulation step. Moreover, comparing the two methods of this section, it is better to use *grdV method* as this method yields better results (Section 7.1.5), even though it comes with the disadvantage of ignoring the invariants in the system description. However, with the ability to issue a warning once an invariant is left, the approach is able state the validity of the models during simulation and warn the user once the model reaches unsampled regions. On top of that, in Section 6.1, an approach is handled that defines an error margin for the generated Verlog-A models against the original netlist.

Summing up this section, a HA is described in Verilog-A by:

(a) the system description: specified element-wise for the matrices $\boldsymbol{A}_{loc}$ and $\boldsymbol{B}_{loc}$, and modeled via nodal equations

(b) the invariants: defined as if conditions (*invV* and optional for debugging with *grdV*)

(c) the guards: in case the *grdV method* is used. The guards are specified as if conditions

(d) the location id: *loc* which is assigned using the $\mathcal{S}_\lambda$ space and optionally the jump functions in addition to the guards (*grdV method*) or the invariants (*invV method*)

(e) the jump functions: which are directly embedded into the model. If the system is modeled with jump functions, a shift value is constantly subtracted from the simulation voltages $V(x_{\lambda,i})$. Note that the shift value is assigned according to the current valid location. As this value is constantly subtracted from the variables of the system, it is referred to as shift ($\boldsymbol{v}_s$) and not reset ($\boldsymbol{v}_r$)

(f) the back-transformation: specified element-wise for the matrices $\boldsymbol{F}_{loc}$ and $\boldsymbol{L}_{loc}$

4.6.3 SystemC-AMS Models

The modeled HA can be also generated in SystemC-AMS syntax. Similarly to the Verilog-A models, the *grdV method* can be used for the modeling process. On the other hand, the system descriptions of SystemC-AMS models need special attention. On top of generating models in the standard modules of SystemC-AMS, the further aim is to generate models which use affine arithmetic decision diagrams (AADD) from [RGJR17] to perform range computations using affine arithmetic. For this, an extensive study has been performed in [Pip19] concerning the model description. The main results are adapted in this part.

Starting with the simplest case, the models can be deployed in SystemC-AMS using the timed data flow (TDF) computation model. Note that during computation, the TDF model processes the data at discrete time steps. However, the system equations are solved by considering the input samples as well as the dynamic behavior of the system as continuous time signals [BCE+10]. The result is then sampled into a signal with the corresponding time step of the port. By that, the continuous dynamic behavior modeled by *Elsa* can be directly used with this computation model. The generated HA for the running example is presented in Algorithm 11.

Algorithm 11 SystemC-AMS model generated as a TDF computation model

```
 1: SCA_TDF_MODULE (HA_tdf)
 2:     declare input and output ports and constructor of HA_tdf
 3:     initialize variables for first
 4:     set module and port attributes
 5:     if $loc == 11 || loc == 0$ then
 6:         set shift vector $\boldsymbol{v}_s$ to $\boldsymbol{x}_{\lambda,op,g1r1}$
 7:         check shifted guards and reassign $loc$ if necessary
 8:     end if
 9:     if $loc == 21$ then
10:         set shift vector $\boldsymbol{v}_s$ to $\boldsymbol{x}_{\lambda,op,g2r1}$
11:         check shifted guards and reassign $loc$ if necessary
12:     end if
13:     if $loc == 22$ then
14:         set shift vector $\boldsymbol{v}_s$ to $\boldsymbol{x}_{\lambda,op,g2r2}$
15:         check shifted guards and reassign $loc$ if necessary
16:     end if
17:     assign all variables according to $loc$
18:     solve dynamic equations
19:     perform back-transformation using the previous found states
20:     variables declarations
21: end SCA_TDF_MODULE
```

Instead of modeling the guards into the if conditions as the *grdV method* did with the Verilog-A models, the location variable *loc* can be used inside the if conditions. This is due to the fact that the value of *loc* is passed down between the simulated time steps. Moreover, the SystemC-AMS models can be also modeled according to Algorithm 8, but this will not be handled here. Hence, whenever the *grdV method* is used for the SystemC-AMS model generation, a modeled is generated similarly to Algorithm 11.

As stated in Algorithm 11, the HA is declared as a TDF computation model. At first, the input and output ports of the HA are declared as sca_out and sca_in from the namespace sca_tdf. Than, the constructor is defined. All remaining variables, of different types including sca_matrix, sca_vector, and sca_trace_variable, are defined at the end of the module at line 20. The variables are then initialized at line 3 inside the *initialize* method of this module. Attributes, such as the time step, are specified at line 4. The lines 5 till 19 are performed inside the *processing* method of this module. Initial, the integer location variable *loc* is set to zero at line 4. During the first run, the HA enters the location $g1r1$ at line 5. The shift vector $\boldsymbol{v}_s$ is then set to the operating point $\boldsymbol{x}_{\lambda,op,g1r1}$, and the guards shifted by this vector are then evaluated. Note that the SystemC-AMS models can either use a reset vector as the Matlab models, or a shift vector analogous to the Verilog-A models. For simplicity, only SystemC-AMS models with shift vectors will be handled. In case a guard condition is true, a location transition occurs and loc is reassign to the target location. If all guard conditions of a location are false, the value of *loc* represents the current location. According to the current location, the remaining variables and matrices are assigned at line 17. At line 18, the dynamic equations of the system are solved in the $\mathcal{S}_\lambda$ space. Four modes have been implemented that described the system behavior in the $\mathcal{S}_\lambda$ space:

ssC method solves the state space equation (sca_ss) provided by the TDF module

eulC method uses the backward Euler method with a fixed time step

rukC method uses the Runge-Kutta method with a fixed time step

disC method discretizes the linear state space with a fixed time step

All methods describe both, the system and input matrices, $\boldsymbol{A}_{loc}$ and $\boldsymbol{B}_{loc}$, as sca_matrix variables of type double. The operating points $\boldsymbol{x}_{op}$, $\boldsymbol{x}_{\lambda,op}$, as well as the operating input $\boldsymbol{u}_{op}$ are variables of type double sca_vector.

The *ssC method* uses the TDF implementation of the state space (sca_tdf::sca_ss). The sca_ss object is provided with the system and input matrix, an identity matrix as the output matrix, a zero matrix as the feedthrough matrix, the state space vector $\boldsymbol{x}_\lambda$, and the input of the system. The solution returned is directly assigned (by reference) into the state space vector $\boldsymbol{x}_\lambda$. Note that the $\boldsymbol{x}_\lambda$ variables are shifted prior to passing the variables to the sca_ss object:

```
// apply shift
  u(0) = in_nin.read() - uOp(0);
  xLamTemp(0) = xLam(0) - xShift(0);
  xLamTemp(1) = xLam(1) - xShift(1);
// statespace is a sca_ss object
  y = statespace(a,b,c,d,xLamTemp,u);
//remove shift
  xLam(0) = xLamTemp(0) + xShift(0);
  xLam(1) = xLamTemp(1) + xShift(1);
```

The remaining methods make use of a constant predefined time step. The *eulC method* uses the backward Euler method. For this purpose, consider the discrete version of Eq. (4.10) with the sample time step T at the i^{th} sample:

$$\dot{\boldsymbol{x}}_\lambda(iT) = \boldsymbol{A}_{loc}(\boldsymbol{x}_\lambda(iT) - \boldsymbol{x}_{\lambda,op}) + \boldsymbol{B}_{loc}(\boldsymbol{u}(iT) - \boldsymbol{u}_{op}) \tag{4.35}$$

The derivative of the state vector $\dot{\boldsymbol{x}}_\lambda(iT)$ can then be replaced with:

$$\dot{\boldsymbol{x}}_\lambda(iT) = \frac{\boldsymbol{x}_\lambda(iT) - \boldsymbol{x}_\lambda((i-1)T)}{T} \tag{4.36}$$

By substituting Eq. (4.36) in Eq. (4.35), and solving for $\boldsymbol{x}_\lambda(iT)$, the following equation can be obtained:

$$\boldsymbol{x}_\lambda(iT) = (\boldsymbol{I} - T\boldsymbol{A}_{loc})^{-1}[\boldsymbol{x}_\lambda((i-1)T) - T\boldsymbol{A}_{loc}\boldsymbol{x}_{\lambda,op} + T\boldsymbol{B}_{loc}(\boldsymbol{u}(iT) - \boldsymbol{u}_{op})] \tag{4.37}$$

Thus, solving Eq. (4.37) with a given sample time step T for each state separately, yields the state space vector $\boldsymbol{x}_\lambda$. Note that, for this method the sample time step T is specified in the model creation options provided to *Elsa*. This is necessary as $(\boldsymbol{I} - T\boldsymbol{A}_{loc})^{-1}$ is calculated during the abstraction process in Matlab and the obtained result is assigned to a matrix in the SystemC-AMS model.

As the *rukC method*, which uses the Runge-Kutta method, is described similarly, the explanation is skipped.

The *disC method* follows a different approach by discretizing the linear state space. As mentioned in Section 2.3.1, the time domain solution of a linear system is given by Eq. (2.26). Performing this calculation in the $\mathcal{S}_\lambda$ space in general with a system matrix $\boldsymbol{A}$ and input matrix $\boldsymbol{B}$ for an initial value $\boldsymbol{x}(iT)$ with a sample time step T yields:

$$\boldsymbol{x}_\lambda((i+1)T) = e^{\boldsymbol{A}_{loc}T}\boldsymbol{x}_\lambda(iT) + \int_0^T e^{\boldsymbol{A}_{loc}(T-\tau)}\boldsymbol{B}_{loc}\boldsymbol{u}(iT)d\tau \tag{4.38}$$

As $\boldsymbol{u}(iT)$ is constant during the sample time step T, it can be moved outside the integral in Eq. (4.38). Moreover, instead of considering the current value of the state vector to compute the next one, the previous value is considered to calculate the current one. Thus, solving this integral for a nonsingular system matrix $\boldsymbol{A}$ yields:

$$\boldsymbol{x}_\lambda(iT) = e^{\boldsymbol{A}_{loc}T}\boldsymbol{x}_\lambda((i-1)T) + \boldsymbol{A}_{loc}^{-1}(e^{\boldsymbol{A}_{loc}T} - \boldsymbol{I})\boldsymbol{B}_{loc}\boldsymbol{u}(iT) \tag{4.39}$$

Similarly to the *eulC* and *rukC* methods, the operations in Eq. (4.39) are performed in Matlab and the resulting two matrices, one multiplied with the state vector and one multiplied with the input vector, are specified in the SystemC-AMS model. Therefore, the sample time step T must be provided to *Elsa* as well.

In the case the jump functions are considered, the state and the input vectors are adjusted as previously seen. Thus, the system behavior is modeled using the *disC method* by:

$$\boldsymbol{x}_\lambda(iT) = e^{\boldsymbol{A}_{loc}T}(\boldsymbol{x}_\lambda((i-1)T) - \boldsymbol{x}_{\lambda,op}) + \boldsymbol{A}_{loc}^{-1}(e^{\boldsymbol{A}_{loc}T} - \boldsymbol{I})\boldsymbol{B}_{loc}(\boldsymbol{u}(iT) - \boldsymbol{u}_{op}) \tag{4.40}$$

A detailed model description of the running example generated using the *disC method* is provided in Listing D.3 in Appendix D.

Regardless of the method used to calculate the current state space vector $\boldsymbol{x}_\lambda(iT)$, the results are transformed back into the $\mathcal{S}_o$ space at line 19 of Algorithm 11 with Eq. (4.11). For the running example, with $x\_lam(0)$, $x\_lam(1)$, and $fooeoob$ representing $\boldsymbol{x}_{\lambda,1}$, $\boldsymbol{x}_{\lambda,2}$, and $-\boldsymbol{L}_{loc}$, respectively, the voltage at the negative terminal of the operation amplifier (Fig. 4.2) is calculated via:

```
neg.write(xOp(21) + f(21,0)*(x_lam(0) - xShift(0))
          + f(21,1)*(x_lam(1) - xShift(1))
          + fooeoob(21,0)*(in_nin.read() - uOp(0)));
```

Note that $fooeoob$ is assigned to $-\boldsymbol{L}_{loc}$.

Optionally, the models can be extended for symbolic simulations using the AADD library from [RGJR17] (available in Github [RG]). The AADD Library combines ordered binary decision diagrams (OBDD) with affine arithmetic forms. This library allows to perform a symbolic simulation in combination with the SystemC-AMS data structure. For that, the standard data types of the variables must be replaced with data types that allow symbolic executions: doubleS, floatS, intAS and boolS instead of the standard data types: double, float, int and bool, respectively. Moreover, the control statements need to be adjusted with ifS, elseS and endS. Thus, during simulation, both cases of these conditions are added to the AADD as terminal nodes. For more details see [RGJR17].

At the current time, the AADD library is not compatible with the TDF state space (sca_tdf::sca_ss). Hence, the *ssC method* cannot be used with the AADD library, while the remaining three methods can. If the system is modeled to be compatible with the AADD library, the variables types of $\boldsymbol{x}$ and $\boldsymbol{x}_\lambda$ as well as the guard conditions are adjusted. Moreover, the tracing of the variables is replaced with the opt_sol class to capture the minimum and maximum of each variable. An detailed example of an AADD model is handled in Section 5.1.2. Note that, the structure of Algorithm 11 was optimized for the use with this library. Specifically, the if conditions at lines 5, 9, and 13, were structure in this way to minimize the path explosion problem.

Summing up this section, a HA in System-AMS is defined by:

(a) the system description: each matrix $\boldsymbol{A}_{loc}$ and $\boldsymbol{B}_{loc}$ is specified as sca_matrix of type double (doubleS for AADD). Four methods exist to model the system behavior

(b) the invariant: *invV method* or optional for debugging (*grdV method*)

(c) the guards: guards are specified as if conditions (ifS for AADD) (*grdV method*)

(d) the location id: an integer variable (doubleS for AADD) *loc* assigned using the shifted guards (*grdV method*) or the shifted invariants (*invV method*)

(e) the jump functions: if the system is modeled with jump functions, a shift vector is constantly subtracted from state space vectors in the $\mathcal{S}_\lambda$ space. Moreover, the shift vector is assigned according to the current operating point. Jump functions can be, analogously to the Matlab models, described with reset vectors, however, this is not handled in this dissertation for simplicity

(f) the back-transformation: each matrix $\boldsymbol{F}_{loc}$ and $\boldsymbol{L}_{loc}$ is specified as sca_matrix of type double (doubleS for AADD). The back-transformation is computed during the simulation at each time step

In Fig. 4.45 the three covered model output languages are presented. The Matlab (*Cora*) models

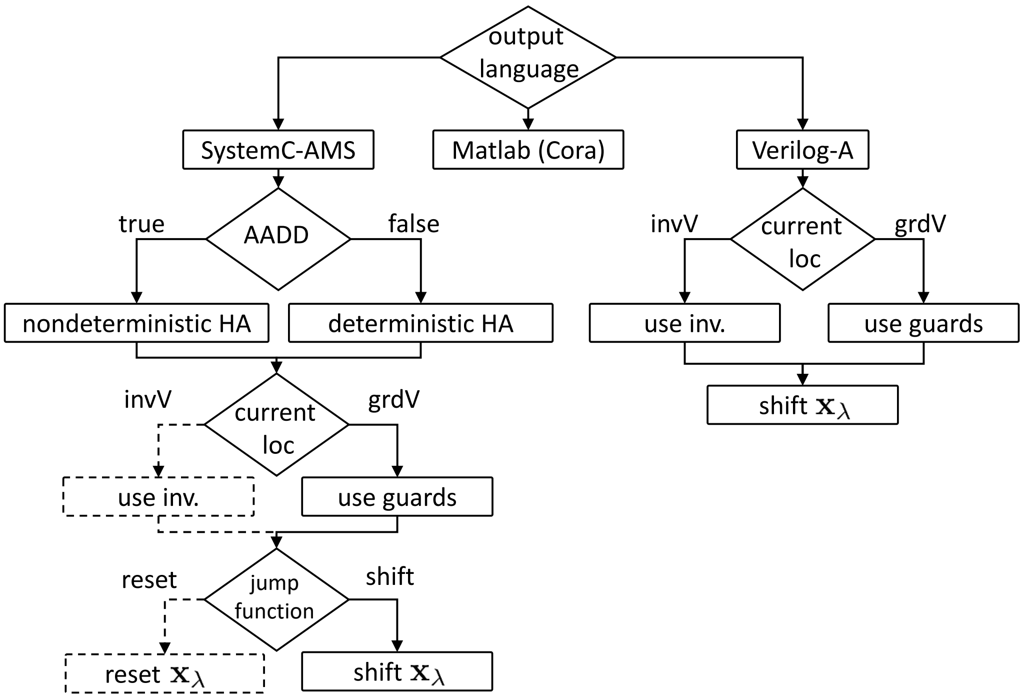


Fig. 4.45. Possible models generated for the three covered output languages. Dashed lines indicate topics not handled in this dissertation.

handled here are usually nondeterministic. Using the AADD library, nondeterministic SystemC-AMS models can be created. In contrast to the previous two output languages, the Verilog-A models are always deterministic. Moreover, in contrast to the Matlab models, the Verilog-A models constantly shift the $\boldsymbol{x}_\lambda$ variables as described in Section 4.6.2. The Matlab models reset the $\boldsymbol{x}_\lambda$ variable once a location transition occurs. Even though both options are available for the SystemC-AMS models, only the shift variant will be handled here for simplicity.

4.7 Summary

In this chapter, the model generation was explained in detail. The approach can be divided into 4 building blocks (see Fig. 4.1). Depending on the space used for calculation, the result as well as the run time of abstraction process vary. Fig. 4.46 shows the section covered depending on the space used for calculation and the desired output language.

Depending on the options specified for the sampling performed by *Vera*, the $\mathcal{S}_\lambda$ space varies. If a $\mathcal{S}_\lambda$ space is obtained with strongly overlapping locations, it is better to use the $\mathcal{S}_{virt}$ space for identifying the guards and model the system with jump functions. If the $\mathcal{S}_\lambda$ has nearly no overlapping regions, the guard and invariant identification can be performed in this space, and the jump functions can be dropped out if desired. In general modeling with the $\mathcal{S}_{virt}$ space yields better guards and invariants. However, the transformation of the result into the $\mathcal{S}_\lambda$ space is a difficult process which does not always yield the best results. Moreover, modeling the system with jump

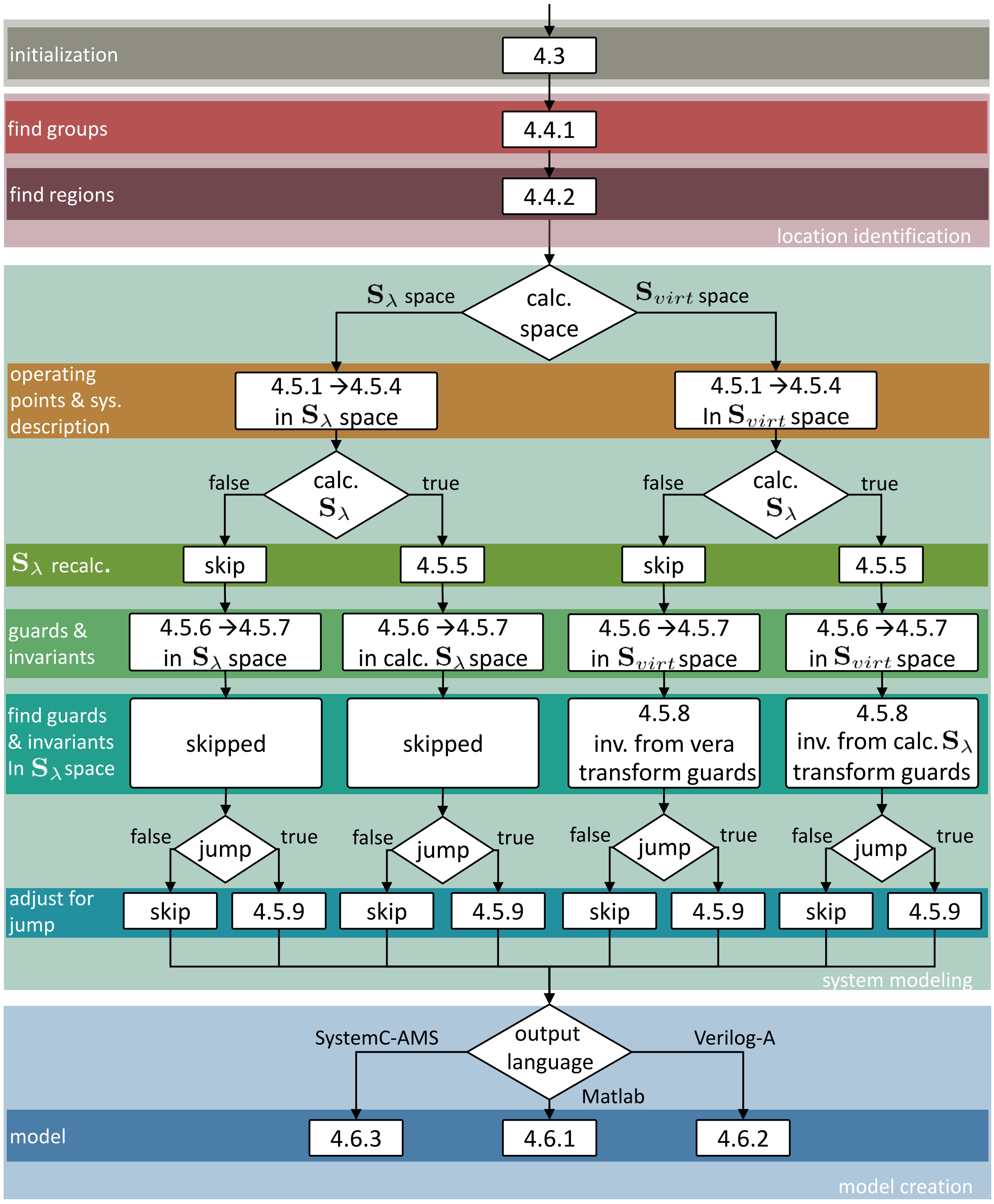

initialization
4.3
find groups
4.4.1
find regions
4.4.2
location identification
$\mathbf{S}_\lambda$ space
calc. space
$\mathbf{S}_{virt}$ space
operating points & sys. description
4.5.1 → 4.5.4 in $\mathbf{S}_\lambda$ space
4.5.1 → 4.5.4 In $\mathbf{S}_{virt}$ space
false
calc. $\mathbf{S}_\lambda$
true
$\mathbf{S}_\lambda$ recalc.
skip
4.5.5
guards & invariants
4.5.6 → 4.5.7 in $\mathbf{S}_\lambda$ space
4.5.6 → 4.5.7 in calc. $\mathbf{S}_\lambda$ space
4.5.6 → 4.5.7 in $\mathbf{S}_{virt}$ space
find guards & invariants In $\mathbf{S}_\lambda$ space
skipped
4.5.8 inv. from vera transform guards
4.5.8 inv. from calc. $\mathbf{S}_\lambda$ transform guards
jump
adjust for jump
4.5.9
system modeling
SystemC-AMS
output language
Verilog-A
Matlab
model
4.6.3
4.6.1
4.6.2
model creation

Fig. 4.46. Sections covered depending on the output language and options for the system modeling.

function usually yields better results. For Verilog-A and SystemC-AMS models it is advised to use *grdV method*, as the models obtained usually attain a fast convergence and simulation time.

In some cases, discontinuities can occur during the transition from one location to another as will be shown in Chapter 7. In general, there are different aspects that can cause discontinuities:

- guards and invariants
- system and input matrices
- back-transformation

When the guards and invariants are ill chosen, discontinuities can occur. In fact, there are many sources that can lead to bad selection of the guards and invariants, for example: when then $\mathcal{S}_\lambda$ space has been strongly modified compared to *Vera*'s calculations, over approximating geometric shapes such as interval hulls are used for the invariant identification (Section 4.5.6), or when the guards are not modeled as halfspaces and enlarged by tolerance factors. Often, these discontinuities also result from the system and input matrices computations performed in Section 4.5.2, especially if there exists a large variation of these matrices among the sampled data. As the matrices of the back-transformation were often as well calculated using mean values, discontinuities can occur, especially if the values of these matrices change strong on the border of the regions. Usually, this is the main reason why discontinuities exist. In this case, increasing the number of locations, or using an enhanced system description as in Section 5.1 can control and decrease the discontinuities.

To sum up, the introduced modeling process has the following properties:

- *Vera* conducts a sampling of the *reachable* state space of the netlist. A nonlinear order reduction is performed up to a given frequency. The sampled points are connected in a graph structure consisting of predecessors, successors, timing information on edges between states, and slew rate limited input connections. Depending on the specified options (mainly $\mathcal{S}_\lambda$ space), different results are obtained during the sampling process.
- at each sample point, information are available about the nonlinear and locally linearized system including:
  - the Spice accurate circuit signal solutions $\boldsymbol{x}$
  - the reduced Kronecker canonical linearized system with the state space vector $\boldsymbol{x}_\lambda$
  - the transformation matrices between the spaces
- *Elsa* models the system by a hybrid automaton with a finite set of locations. In each location, the system behavior is modeled via a linear state space representation
  - various options can be specified, such as the number of locations of the desired HA, the type of the invariants and guards, and the system description resulting in more accurate models.
  - the generated models are available in Matlab (*Cora*), SystemC-AMS, or Verilog-A syntax
- the modeling process is fully automated
- Various extensions can be used to enhance the obtained models (see Chapter 5)
- the models can be verified against their original netlist (see Chapter 6), or used in verification (reachability analysis, symbolic simulations, online monitoring) or simulations routines with huge speed-up factors (see Chapter 7)

5 Extended Model Abstraction

The models obtained from the abstraction approach presented in Chapter 4 can be enhanced by various extensions. These extensions are:

(a) modeling the system with parameter variation to accommodate for the errors performed during the abstraction process

(b) modeling the system with parameter variation to include the process parameter variations

(c) generating a compositional HA

(d) generating a conformant model that harbors the system behavior from the real circuit

(e) optimizing the HA over the available options, thereby comparing the generated HA with the original netlist

The extension (e) optimizes the HA over its generation options. As seen in Chapter 4, various options can be used to generate the HA. This extension iterates over all available options creating various models to find the best one. These options include the number of locations (cluster number), the region identification method, the state space used for the identification of the guards and invariants, the guard identification methods, as well as the graphical representation of the guards and invariants. Based on the deviation of the $\boldsymbol{x}$ values between the netlist and the generated models, the models are judged and the best one is selected. As this is straight forward, this extension will not be explained further here.

On top of generating models that accommodate for the abstraction errors as well as for the process parameters ((a) and (b)), extension (d) generates a conformant model by using measurements from the real circuit. Thus, the generated model harbors the behavior from the real circuit. The detailed process is explained in [KTR+20]. This process includes generating an abstract model with *Elsa*, and then train this model by various performed simulations and measurements. As the training is done after the model abstraction, the generation of a conformant model will not be covered here.

Extensions (a), (b), and (c) will be handled in Section 5.1, Section 5.2, and Section 5.3, respectively. Note that extension (a) is presented in [TH19a], and extension (b) is presented in [TH20].

5.1 Modeling With Parameter Variations

The models from Chapter 4 can be extended to compensate for the errors and deviations obtained during the abstraction process. For this, the model descriptions are extended. In the following, this extension will be explained for the Matlab (*Cora*) and the SystemC-AMS abstract models.

5.1.1 HAs in Cora With Parameter Variations

Instead of modeling the system matrix $\boldsymbol{A}_{loc}$ and the input matrix $\boldsymbol{B}_{loc}$ from Eq. (4.10) by matrices, whose values were calculated by either method from Section 4.5.2, $\boldsymbol{A}_{loc}$ and $\boldsymbol{B}_{loc}$ can be specified as matrix zonotopes or interval matrices as performed in [TH19a].

As stated in [Alt15], a matrix zonotope is defined as:

$$\boldsymbol{A}_{[z]} = \{\boldsymbol{C} + \sum_{i=1}^{k} \beta_i \cdot \boldsymbol{G}_i \mid \beta_i \in [-1, 1]\}, \qquad \boldsymbol{C}, \boldsymbol{G}_i \in \mathbb{R}^{m \times m} \tag{5.1}$$

In this case $\boldsymbol{C}$ is the matrix center, while $\{\boldsymbol{G}_i \mid i = 1, \ldots, k\}$ are the matrix generators of the matrix zonotope $\boldsymbol{A}_{[z]}$. On the other hand, an interval matrix can be seen as a special case of a matrix zonotope, which specifies an interval for each matrix element a_{ij}:

$$\boldsymbol{A}_{[i]} = \{[\boldsymbol{A}_{min}, \boldsymbol{A}_{max}] \mid \forall i, j : a_{min,i,j} \leq a_{max,i,j}\}, \qquad \boldsymbol{A}_{min}, \boldsymbol{A}_{max} \in \mathbb{R}^{m \times m} \tag{5.2}$$

Instead of modeling the System matrix $\boldsymbol{A}_{loc}$, for example with the mean of eigenvalues as shown in Fig. 4.19a, all eigenvalues are considered. This is done by hulling the eigenvalues in the desired shape. These shapes include intervals, zonotopes, or polytopes as shown in Figs.5.1a, 5.1b, and 5.1c, respectively, for the running example from Section 4.1. Note that the HA has 3 locations $g1r1$, $g2r1$ and $g2r2$, but only two groups $g1$ and $g2$ of similar eigenvalues (see Section 4.5).

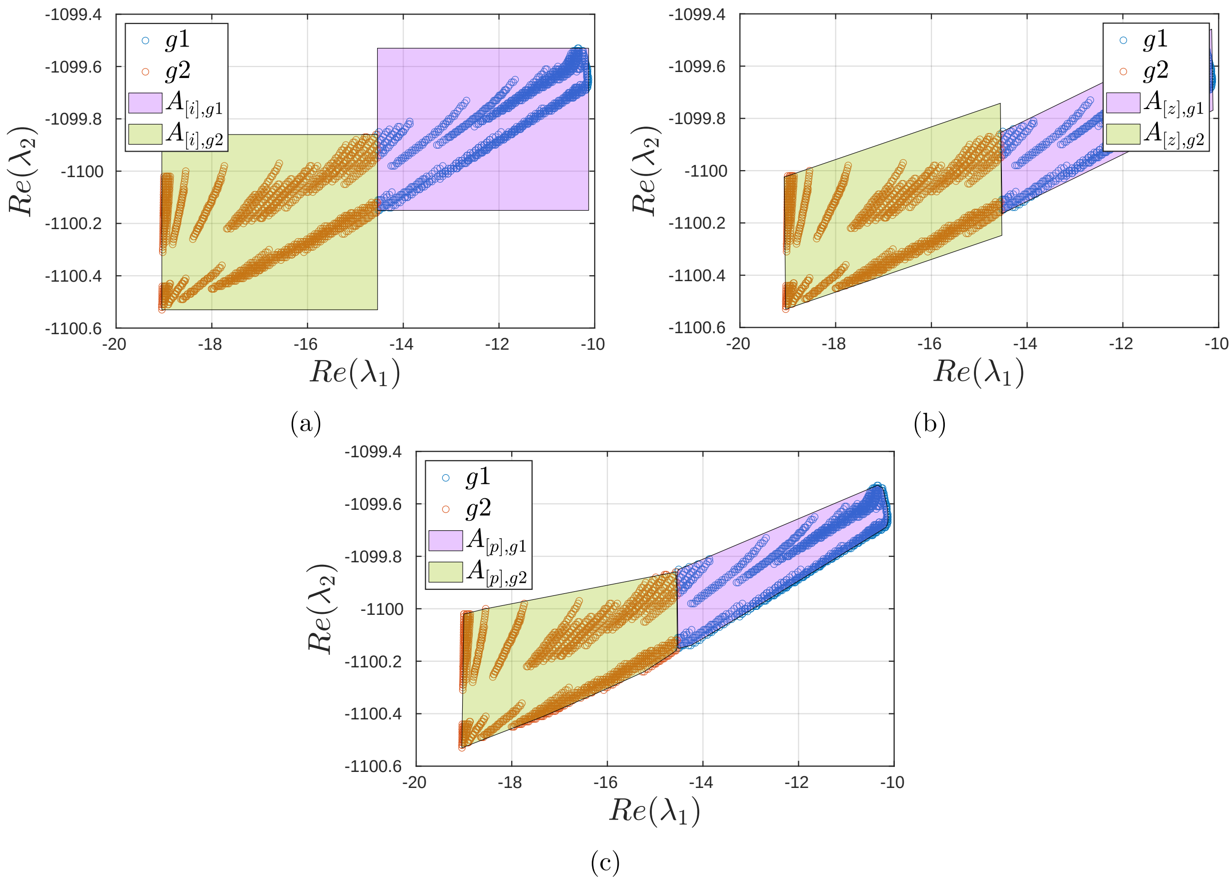


Fig. 5.1. Hulling the eigenvalues by (a) intervals, (b) zonotopes, and (c) polytopes.

Considering the matrix that contains the eigenvalues $\boldsymbol{Eig}$ (see Eq. (4.4)), the columns of this matrix are considered individual dimensions and are thus transposed. The $1, \ldots, l_{loc}$ points belonging to the same location are then hulled for example by a zonotope as follows:

$$\mathcal{Z}\left(\begin{bmatrix} Re(\lambda_{1,1}) & Im(\lambda_{1,1}) & \dots & Re(\lambda_{1,m}) & Im(\lambda_{1,m}) \\ Re(\lambda_{2,1}) & Im(\lambda_{2,1}) & \dots & Re(\lambda_{2,m}) & Im(\lambda_{2,m}) \\ \vdots & \vdots & \dots & \vdots & \vdots \\ Re(\lambda_{l_{loc},1}) & Im(\lambda_{l_{loc},1}) & \dots & Re(\lambda_{l_{loc},m}) & Im(\lambda_{l_{loc},m}) \end{bmatrix}^T\right) \tag{5.3}$$

As observed when comparing Eq. (5.3) with Eq. (2.4), each row of the obtained zonotope $\mathcal{Z} \in \mathbb{R}^{2m}$ represents a dimension. Moreover, each row in Eq. (5.3) (before transposing) represents a point which is contained in the obtained zonotope $\mathcal{Z}$. This is illustrated in Fig. 5.1b for the first and third dimension of $\mathcal{Z}$ with the reduced order $m = 2$.

The obtained results from Fig. 5.1a, Fig. 5.1b, and Fig. 5.1c are transformed to interval hulls, matrix zonotopes or matrix polytopes. For example, in case the eigenvalues are distinct, this can be realized for a matrix zonotope $\boldsymbol{A}_{[z]} \in \mathbb{R}^{m \times m}$, with the zonotope $\mathcal{Z}$ obtained from Eq. (5.3) via:

$$\boldsymbol{C} = \begin{bmatrix} c_1 & c_2 & \dots & 0 & 0 \\ c_4 & c_3 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & c_{2m-3} & c_{2m-2} \\ 0 & 0 & \dots & c_{2m} & c_{2m-1} \end{bmatrix} \qquad \boldsymbol{G}_i = \begin{bmatrix} g_{i,1} & g_{i,2} & \dots & 0 & 0 \\ g_{i,4} & g_{i,3} & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & g_{i,2m-3} & g_{i,2m-2} \\ 0 & 0 & \dots & g_{i,2m} & g_{i,2m-1} \end{bmatrix} \tag{5.4}$$

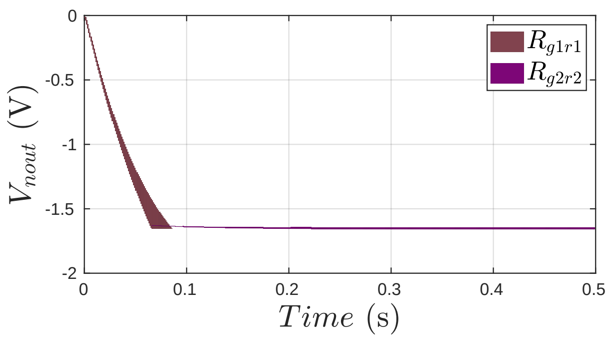
Where $\boldsymbol{c}$ and $\boldsymbol{g}_i$ represent the center and i^{th} generator of the zonotope $\mathcal{Z}$ (see Eq. (2.4)), and $\boldsymbol{C}$ and $\boldsymbol{G}_i$ represent the matrix center and the i^{th} matrix generator of the matrix zonotope $\boldsymbol{A}_{[z]}$ (see Eq. (5.1)). Note that, in general $\boldsymbol{C}$ and $\boldsymbol{G}_i$ are in the Jordan normal form (see Section 2.2.1). For the input matrix $\boldsymbol{B}_{loc}$, the same procedure is performed.

At the current time, the reachability analysis of system description specified using matrix polytopes for HAs is not completely implemented in *Cora* and therefore dropped out in the following.

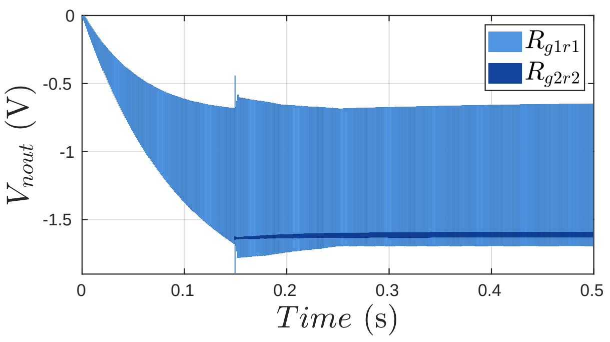
As observed, this approach improves the previous abstractions of the system behavior performed in Section 4.5.2, by over approximating the system and input matrices by matrix zonotopes or interval hulls, thereby covering the whole sampled system behavior.

The over approximation of the behavior varies depending on the sampled point as well as on the desired representation. Considering for example the eigenvalues that characterize the system matrix, for the same sampled data, greater over approximation would be obtained when using interval hulls (see Fig. 5.1a) instead of using matrix zonotopes (see Fig. 5.1b). This is due to the fact that a matrix zonotope considers the correlation between the underlying elements of the matrices (Eq. (5.1)), while an interval matrix does not (Eq. (5.2)). For the running example, both cases still yield over approximations compared to the sampled data or a matrix polytopic description (Fig. 5.1c). On the other hand, the transformation matrices could be modeled similar to the system and input matrices by this extension. However, as this yields even larger over approximations, this is not performed in the following.

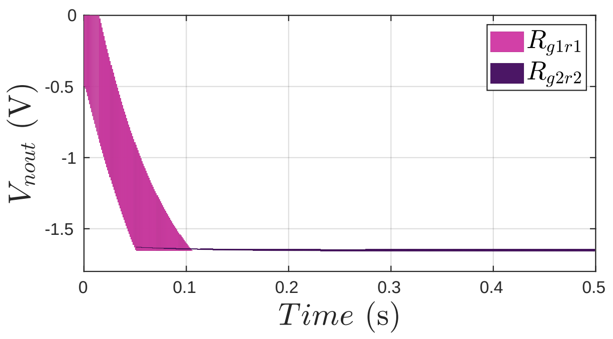
For the running example, an abstract model was generated using this methodology similarly as in Chapter 4. For simplicity, the invariants of the HA were modeled as intervals, while the guards were modeled as polytopes identified using the *distance method*. The system descriptions were generated according to this section, with the in Fig. 5.1b illustrated zonotopes. These zonotopes were then converted into matrix zonotopes to replace the system matrix. Similarly, the input matrix was as well described using matrix zonotopes. For an input voltage $V_{nin} = 4$ V, the result of the reachability analysis performed in *Cora* is presented in Fig. 5.2a. If the initial conditions of the state space variables are specified to be $x_{\lambda,i} \in [-0.5, 0.5]$, Fig. 5.2c is obtained. Hence, this figure illustrates the result of a reachability analysis performed on a HA with parameter variation and an uncertain region for initial conditions specified as a zonotope. Additionally, the input can be specified with uncertainties as well. For an input $V_{nin} = [1.5, 2.5]$ V and $V_{nin} = [2.5, 3.5]$ V, with the initial conditions again set to zero, Fig. 5.2b Fig. 5.2d are obtained, respectively. With a larger input voltage, the HA reaches the saturation faster as observed in Fig. 5.2d compared to Fig. 5.2b.



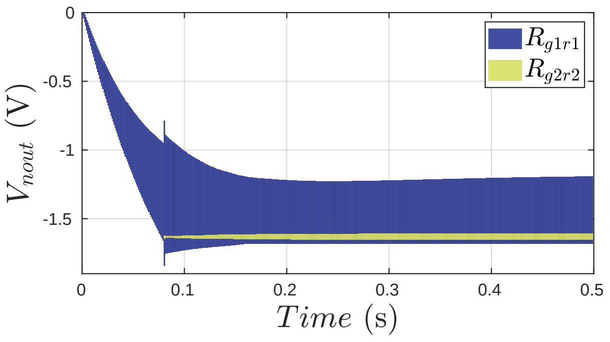
(a)



(b)



(c)



(d)

Fig. 5.2. Results of the reachability analysis performed with *Cora* for the HA modeled according to this section. The HA is simulated (a) with an input voltage $V_{nin} = 4$ V, in (c) with additionally uncertainties in the initial conditions ($\boldsymbol{x}_{\lambda,i} = [-0.5, 0.5]$), while in (b) and (d) with uncertainties in the input $V_{nin} = [1.5, 2.5]$ V and $V_{nin} = [2.5, 3.5]$ V, respectively.

5.1.2 HAs in SystemC-AMS With Parameter Variations

As stated in Section 4.6.3, using the AADD library, the SystemC-AMS models can be extended to model uncertainties. These uncertainties can be either presented in the simulation input, thereby allowing for a symbolic simulation yielding similar results as a reachability analysis, or in the system description. In this section, as previously seen with the Matlab models. Both aspects will be handled in this section.

The AADD library represents continuous propagated uncertainties by affine arithmetic forms

[RGJR17; dS04]:

$$\{x = x_c + \sum_{i=1}^{n} x_i\beta_i \mid \beta_i \in [-1,1]\}, \qquad x, x_i, x_c \in \mathbb{R} \tag{5.5}$$

Each variable x_i models the sensitivity of x to the basic uncertainty β_i. Specifically, the uncertainty symbol β_i, also known as noise symbol, is a symbolic variable whose exact value is unknown, but lies in the interval $[-1, 1]$. The $i \in \{1, \dots, n\}$ symbols are independent of each other. However, different affine forms can share the same uncertainty symbols representing their dependencies [ZGO+19]. Each symbol β_i is scaled by the corresponding partial deviation x_i. Notice the similarity between this equation and the definition of a zonotope in Theorem 2.1.3.

In Fig. 5.3, a graphical representation of the joint range of the two correlated variables $x_1 = 10 - 4\beta_1 + 5\beta_2 + 3\beta_3$ and $x_2 = 5 - 2\beta_1 + 1\beta_2 - 2\beta_3$ in affine and interval arithmetic is presented. As observed, while the joint range obtained with affine arithmetic (zonotope $\mathcal{Z}$) considers the

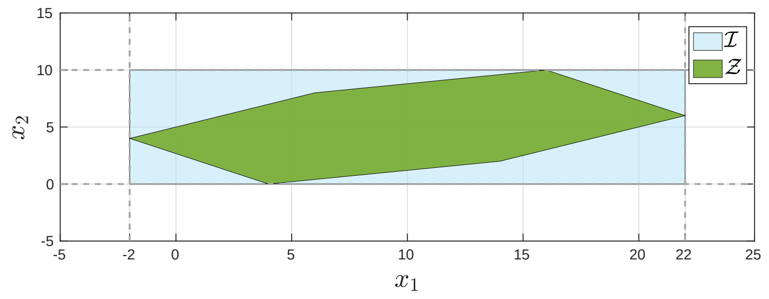


Fig. 5.3. The joint range obtained for the dependent variables x_1 and x_2 using affine arithmetic ($\mathcal{Z}$) and interval arithmetic ($\mathcal{I}$)

correlation between the variables, interval arithmetic (interval hull $\mathcal{I}$) considers the maximum and minimum values of the variables independently.

There are two approaches available for the deployment of the affine forms in the generated models. The first approach is to model the elements of the input vector $\boldsymbol{u}$, the elements of the state variables $\boldsymbol{x}$, $\boldsymbol{x}_\lambda$, and the *loc* variable as affine forms of type doubleS. Moreover, the guard conditions are as well changed (ifS and elseS). Hence, a standard SystemC-AMS simulation becomes a symbolic simulation with AADD for a specific range of the input signals. To demonstrate this option, a SystemC-AMS model was created similarly to Section 7.1.6 with a modeling time of 8.22 s. The underlying $\mathcal{S}_\lambda$ space exhibited minimal overlapping regions (Fig. 7.10). The invariants were modeled as interval hulls, while the guards, identified using the *distance method*, were modeled as halfspaces. Moreover, the HA is modeled with jump functions. As stated in Section 4.6.3, there are four methods available to generate the abstracted HA in SystemC-AMS syntax. For this section, the modeling process is restricted to the *disC method*, which models the behavior of the circuit as a discrete system according to Eq. (4.40).

For this model, a symbolic simulation using the AADD library was performed with an input sine wave at V_{nin} of amplitude 1 V, an uncertain offset of 0.5 V, and a frequency of 1 Hz. In short, $V_{nin} = sin(2\pi \cdot t) + 0.5\beta$ V with $\beta \in [-1, 1]$. Fig. 5.4 shows the result of this simulation. As observed, the circuit does not reach the limiting behavior. Thus, the HA stays in a single location during the entire simulation.

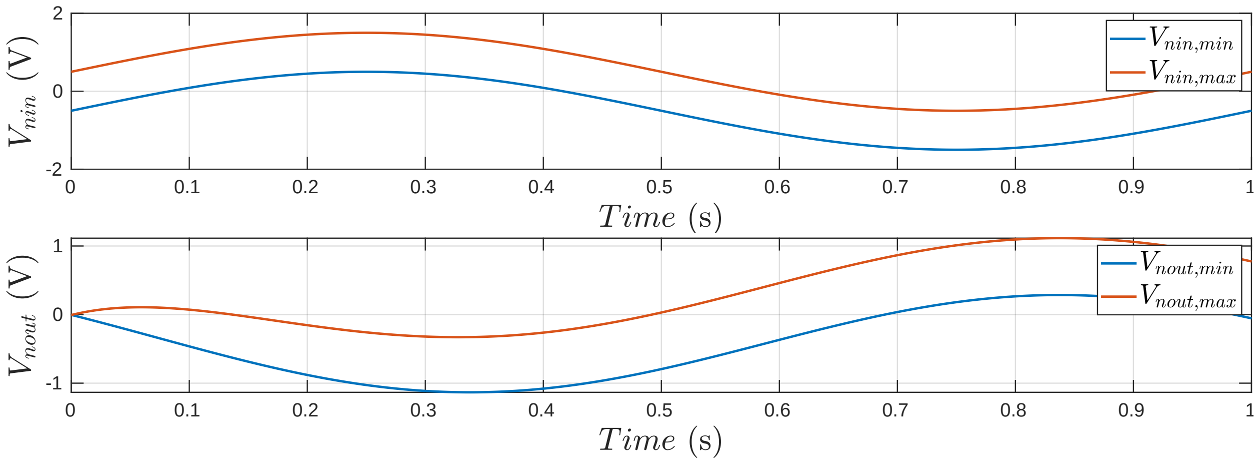


Fig. 5.4. Simulation of the generated HA with an input $V_{nin} = sin(2\pi \cdot t) + 0.5\beta$ V.

Two more simulation were performed on the same model with two different input signals. The results are presented in Fig. 5.5. Both input signals are still sine waves with a frequency of 1 Hz. The first input signal has an amplitude of 4 V and an uncertain offset of 0.1β V. The corresponding results are illustrated in the colors blue (min) and red (max). The second signal has an amplitude of 3 V with the same uncertain offset. The corresponding results are illustrated in yellow (min) and purple (max).

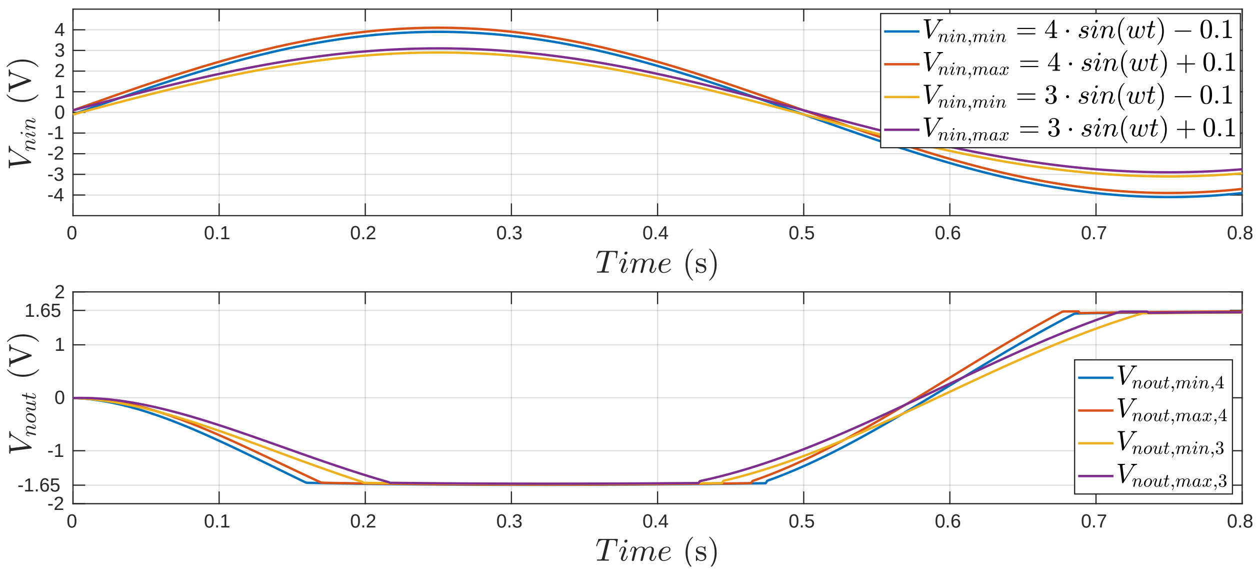


Fig. 5.5. Simulation of the generated HA with different values at the input V_{nin} (first row). The result of the output voltage is illustrated in the second row. For the case the input is $V_{nin} = 4 \cdot sin(wt) + 0.1\beta$ V, $V_{nout,min,4}$ and $V_{nout,max,4}$ are obtained, while for the input $V_{nin} = 3 \cdot sin(wt) + 0.1\beta$ V, $V_{nout,min,3}$ and $V_{nout,max,3}$ are obtained with $w = 2\pi \cdot 1\ s^{-1}$.

As observed in Fig. 5.5, in both cases the system goes into the limiting behavior. For the sine wave at the input of the system with a larger amplitude (red and blue), the HA reaches saturation faster than for a smaller amplitude. In both cases, the HA undergoes several location switches. Specifically, the HA performs four location transitions. Note that the output voltage of both models at V_{nout} was extracted from the $\boldsymbol{x}$ vector after simulating in the $\mathcal{S}_\lambda$ space and performing the back-transformation to the $\mathcal{S}_o$ space during every time step.

The second approach for the deployment of the affine forms in the generated models, is to additionally model the system description with parameter variations. This can be realized by modeling the elements of the system and input matrices, $\boldsymbol{A}_{loc}$ and $\boldsymbol{B}_{loc}$, respectively, using affine forms. For this, the elements can be either specified using a single sensitivity which bounds the range of the element thus corresponding to an interval description, or using several sensitivities x_i and their corresponding symbolic uncertainties β_i (see Eq. (5.5)), thereby consider the correlations between the elements, which is similar to a zonotopic description.

To demonstrate this approach, a model was created using affine forms that additionally to the guards, the input vector $\boldsymbol{u}$, the state vectors $\boldsymbol{x}$ and $\boldsymbol{x}_\lambda$, and the location variable loc, models the elements of the system matrix $\boldsymbol{A}_{loc}$ and the input matrix $\boldsymbol{B}_{loc}$ for each location using affine forms. During this process, each element of these matrices was described by a single uncertainty that bound its maximum and minimum values. This can be interpreted similar to using an interval matrix description from Section 5.1.1 to replace the system and input matrices.

Two simulations were performed on the generated HA. In the first simulation a sine wave was applied at the input V_{nin} with an amplitude of 2.7 V and a frequency of 1 Hz. The corresponding signals in Fig. 5.6 are colored in blue and red for the minimum and maximum values, respectively. Note that the voltage at the input is illustrated in the first row of Fig. 5.6. The voltage at the output, which is selected from the $\boldsymbol{x}$ vector after performing the back-transformation at each iteration step, is illustrated in the second row. In the second simulation, the input signal has been modeled additionally with an uncertain offset, such that the signal at the input is now $V_{nin} = 2.7 \cdot sin(2\pi t) + 0.5\beta$ V. The corresponding results in Fig. 5.6 are colored in yellow and purple for the minimum and maximum values, respectively. As observed in Fig. 5.6, even though the model

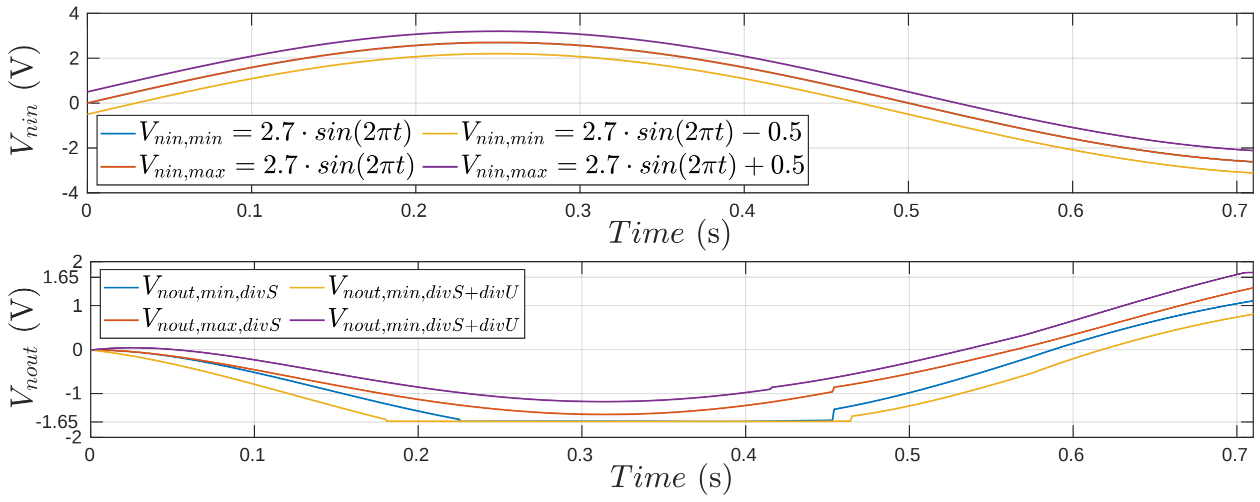


Fig. 5.6. Simulation of the generated HA. If the input $V_{nin} = 2.7 \cdot sin(2\pi t)$ V has no uncertainties, only the uncertainties in the system description are considered and $V_{nout,divS}$ is obtained. In case the input has an uncertain offset $(0.5\beta$ V), $V_{nout,divS+divU}$ is obtained.

harbors deviations in the system descriptions, the results are tightly bounded in both cases by the affine forms. Moreover, the result bounded by the minimum and maximum values (colored in yellow and purple) of the second simulation, is slightly wider and contains the first simulation. Note that the system reached the limiting behavior sooner in the presence of the additional deviation in the input signal.

Two additional symbolic simulations were performed on this model using again the AADD library. Fig. 5.7a illustrates the result at the output of the circuit V_{nout} for a step function of $2 + 0.5\beta$ V at the input V_{nin} at $t = 0$ s. In Fig. 5.7b, the amplitude of the input signal is increased to 3 ± 0.5

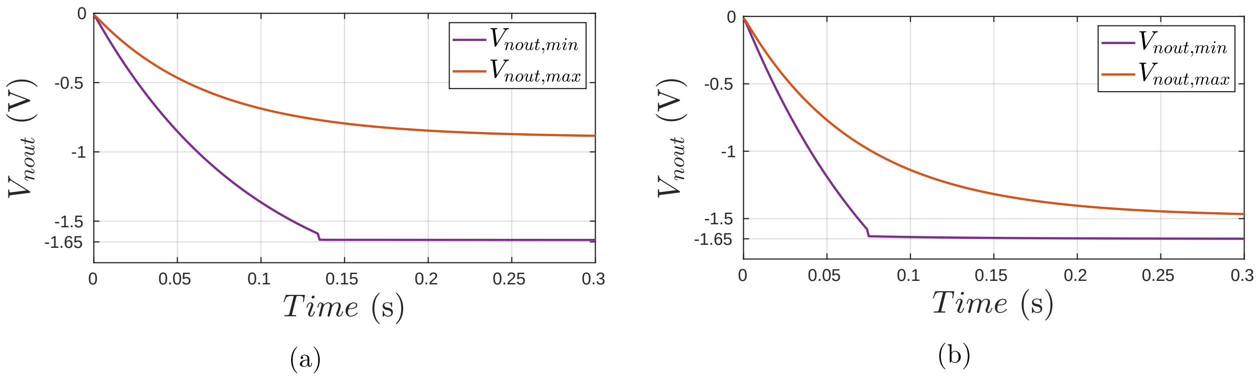


Fig. 5.7. Symbolic simulations of a HA modeled with parameter variations. The output is presented for an input (a) $V_{nin} = 2 + 0.5\beta$ V and (b) $V_{nin} = 3 + 0.5\beta$ V.

V. As illustrated in Fig. 5.7, even for an over-approximative model, the symbolic simulation yield tight bounded results. As the input voltage increases, the system reaches the saturation state at 1.65 V faster, decreasing the uncertain region bounded by the maximum and minimum values.

Fig. 5.8 presents a comparison between the reachability analysis conducted in *Cora* from Section 5.1.1, and the previous performed symbolic simulation for the same input voltages. Both used

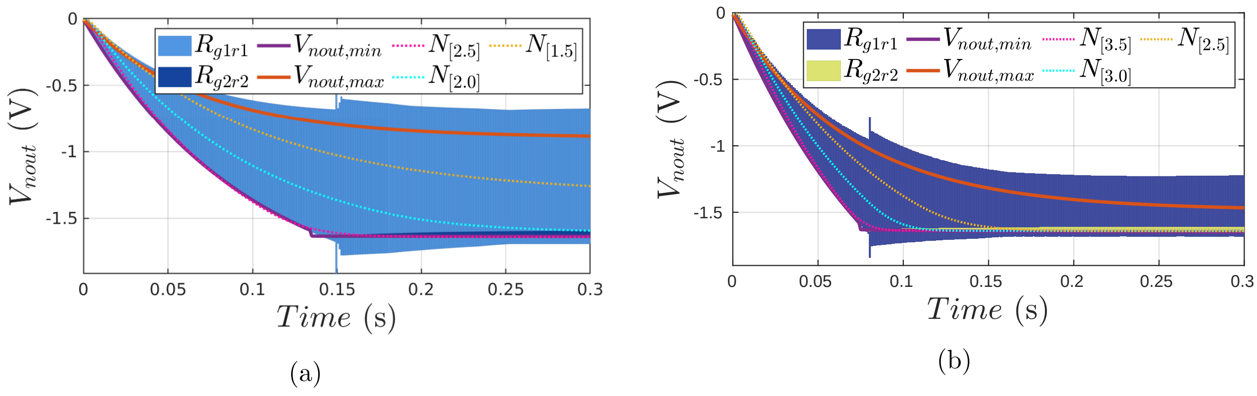


Fig. 5.8. Comparison of the results obtained with *Cora*, SystemC-AMS, and the Spice netlist. The output is presented for an input (a) $V_{nin} = 2 + 0.5\beta$ V and (b) $V_{nin} = 3 + 0.5\beta$ V for both HAs. Additionally several simulations (labeled N) were performed on the Spice netlist.

HAs contain parameter variations. However, while the system behavior of the HA used for the reachability analysis was described with matrix zonotopes, the system behavior of the HA used for the symbolic simulation was described similarly to using an interval matrix. That is, for each element of the system and input matrices, affine forms with single independent symbolic uncertainties were used. As observed, even though the HA in SystemC-AMS was described with a more over-approximative system description, the results are more accurate than the results obtained by the reachability analysis conducted in *Cora* on the more accurate HA. Several simulations were additionally performed on the Spice netlist with various input voltages. These simulations are labeled as N in Fig. 5.8 and subscipted according to the amplitude of the input step function. As observed, these simulations lie for both HAs in the reachable regions. The over approximations of both HAs can be partially traced back to the over approximative system description as observed in Fig. 5.1.

In Section 7.3, a SystemC-AMS model is created according the methodology presented in this section. Moreover, a case study is stated, that starts with modeling the basic components using the AADD library, like the elements of the system an input matrix, then additionally models the location variable *loc*, and finally the elements of the transformation matrices using this extension.

5.2 Modeling With Process Parameter Variations

The second extension to the approach from Chapter 4, enhances a HA to model the parameter variation due to the process parameters used. An overview of the approach is presented in Fig. 5.9. As illustrated, the process parameters along with the netlist are passed to *Vera* for sampling. From

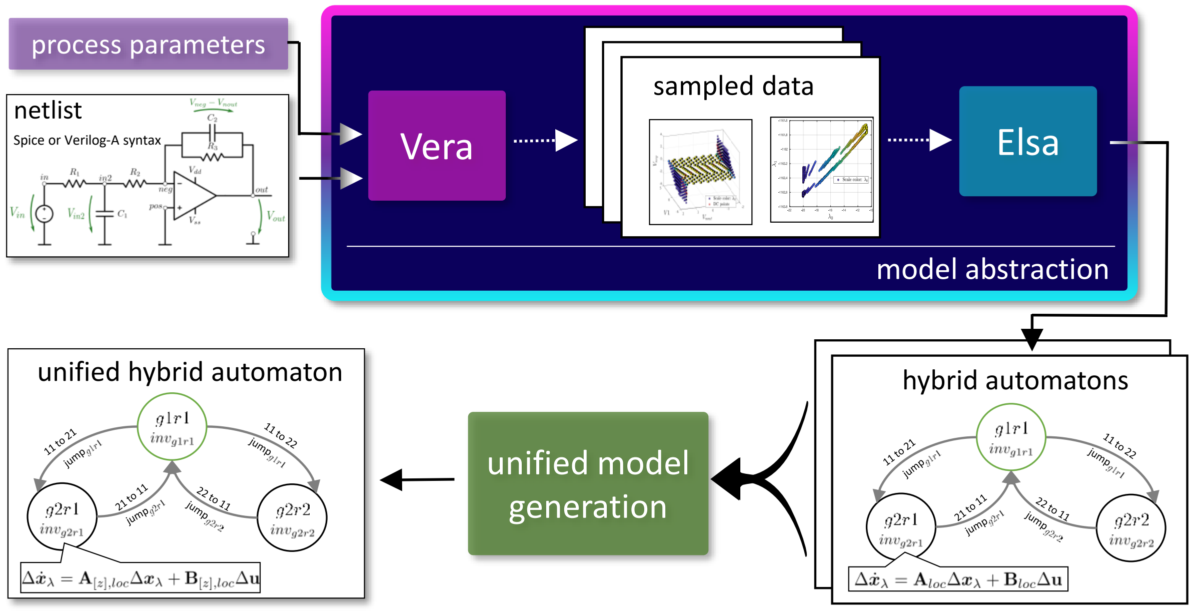


Fig. 5.9. Overview of modeling the system with the process parameter variation.

this netlist (initial netlist), *Vera* generates randomly, in a Monte Carlo fashion, various versions of the netlist with different parameters according to the specified process parameters. The netlists are afterwards sampled in parallel to generate several *acv* files. These files are passed to *Elsa*, which in turn generates a HA from every netlist. This process is executed in parallel in Matlab. Finally, the generated HAs are merged into a unified model, denoted as the unified HA.

The generation of the unified HA is executed in several steps. First the invariants and guards of the underlying models are merged. For the invariants, the task is to model a unified invariant that contains all invariants of the same locations. For the guards, the process is similarly. However, guards of type halfspace are not supported. Note that *Elsa* assigns names to the locations randomly, thus, before the guards and invariants are merged, the corresponding locations are first identified by using their operating points.

In the second step, the system and input matrices are generated as matrix zonotopes or interval hulls, thereby similar to Section 5.1.1, the elements of the system and input matrices of the underlying HAs are hulled by zonotopes or interval hulls, which are then transformed to either matrix zonotopes or interval matrices. There are two methods that can be used

exMat method	the system descriptions of the underlying HAs are modeled by matrices according to Section 4.5.2
exZ method	the system descriptions of the underlying HAs are modeled by matrix zonotopes or interval matrices as in Section 5.1.1

In case the *exMat method* is used, and thereby the system behaviors of the underlying HAs are described by matrices, the hulling is performed in each location on the elements of these matrices. For example, the matrix zonotope $\boldsymbol{A}_{[z]}$ of the unified HA is in this case obtained by first computing the zonotope $\mathcal{Z} \in \mathbb{R}^{2m}$ that contains the eigenvalues of the system matrices $\boldsymbol{A}_j$ of the $j \in \{1, \ldots, l_{Sys}\}$ underlying HAs in the same location *loc*:

$$\mathcal{Z}\left(\begin{bmatrix} Re(\lambda_{1,1}) & Re(\lambda_{1,2}) & \ldots & Re(\lambda_{1,l_{Sys}}) \\ Im(\lambda_{1,1}) & Im(\lambda_{1,2}) & \ldots & Im(\lambda_{1,l_{Sys}}) \\ \vdots & \vdots & \ldots & \vdots \\ Re(\lambda_{m,1}) & Re(\lambda_{m,2}) & \ldots & Re(\lambda_{m,l_{Sys}}) \\ Im(\lambda_{m,1}) & Im(\lambda_{m,2}) & \ldots & Im(\lambda_{m,l_{Sys}}) \end{bmatrix}\right), \tag{5.6}$$

followed by computing the matrix zonotope $\boldsymbol{A}_{[z]}$ according to Eq. (5.4). Each j^{th} column in Eq. (5.6) contains the eigenvalues of the system matrix $\boldsymbol{A}_j$. In case the *exZ method* is used, and thereby the system behaviors of the underlying HAs are described by matrix zonotopes or interval matrices, the matrix zonotopes or interval matrices are first transformed into zonotopes or interval hulls and the hulling is performed on all vertices of these shapes belonging to the same location. For example, in case the underlying HAs were modeled with matrix zonotopes, each matrix zonotope is first changed into a zonotope by reversing Eq. (5.4). The vertices of the obtained zonotopes are then computed, and a zonotope that hulls these points is calculated. Finally, the zonotope is transformed into a matrix zonotope according to Eq. (5.4). Hence, instead of considering for each of the underlying HAs of the same location the eigenvalues of the system matrix in Eq. (5.6), the eigenvalues contained in the matrix zonotopes $\boldsymbol{A}_{[z],j}$ of the $j \in \{1, \ldots, l_{Sys}\}$ underlying HAs are in general considered. Thus, while the first method models only the abstracted behaviors of the system, the second method considers every possible behavior the underlying systems attained during sampling. In the last step, the transformation matrices of the unified model are found by computing the mean value over all transformation matrices belonging to the same location.

To illustrate the approach, the running example from Section 4.1 is considered. The operation amplifier was modeled in a 350 nm CMOS technology. The detailed description of the operation amplifier is given in Appendix C. The nominal netlist, specifies the parameters R1, R2, and R3 as 99.6 kΩ, 999.6 kΩ, and 999.6 kΩ, while the capacitors C1 and C2 are set to 7.32 nF and 73.28 nF, respectively. This netlist is provided to *Vera*. According to the process parameters, selected parameters are deviated. The parameter deviated include all parameters that can influence the deviation for the capacitors and resistors, and 9 parameters for p and n channel transistors, including the most significant ones as the threshold voltage, width and length, thickness of the oxide, and the field mobility. Beside the global deviations, mismatches between all resistors, capacitors, and the two transistors at the input stage are considered. Note that the values of the parameters are drawn similarly as during a Monte Carlo (MC) simulation. Moreover, the parameter selection was limited as they were manual selected. This can be extended to cover all varying process parameters.

For this demonstration, 100 models were generated that abstracted 100 netlists. The distributions of the three resistors, the two capacitors, as well as the transconductance (gm) of the two input stage transistors are illustrated in Fig. 5.10.

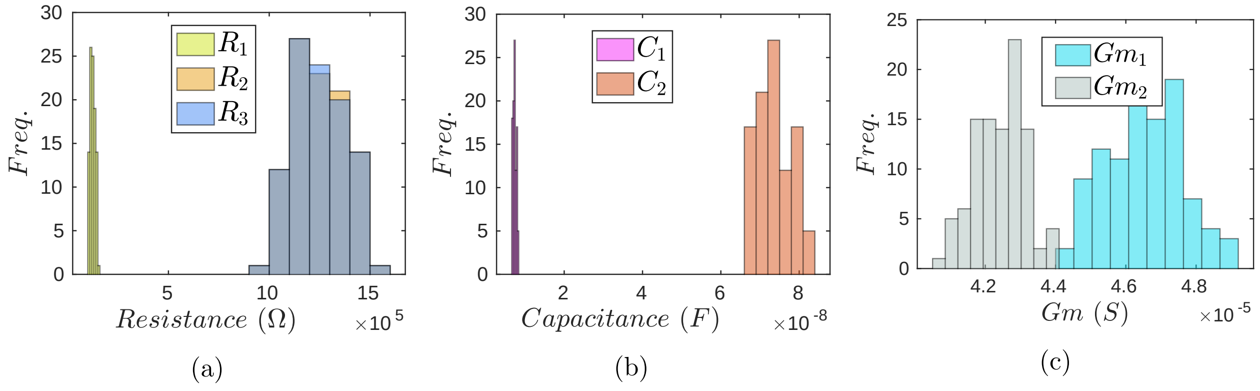
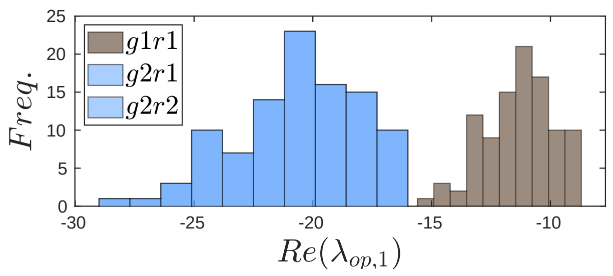


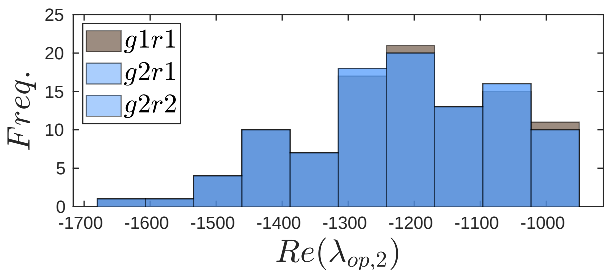
Fig. 5.10. Density plots based on the process parameters of 100 MC samples for the (a) resistance, (b) capacitance, and (c) transconductance of the two transistors at the input stage.

On a Linux machine with 64 cores (intel Xeon @ 2.10 GHZ) with 256 GB of RAM, the sampling of 100 transistor level circuits was executed in parallel in 17 min and 43 s. On the other hand, the model abstractions to 100 HAs were executed in parallel in 25.57 s using *Elsa*. The unified HA was created from these 100 models in 9.04 s. For each HA, the guards were identified as polytopes using the *distance method* from Section 4.5.7. The invariants were modeled as polytopes as well. Both identifications were performed in the $\mathcal{S}_{virt}$ space. The system and input matrices were modeled by using the mean values of the sampled points (*mean method*). As the system has two real eigenvalues, $\boldsymbol{A}_{loc}$ is a diagonal matrix with the mean of the eigenvalues, $\lambda_{op,1}$ and $\lambda_{op,2}$, on the diagonal. As stated, the unified HA merges the results. The values of the representing eigenvalues, $\lambda_{op,1}$ and $\lambda_{op,2}$, of the 100 HAs are represented in Fig. 5.11a and Fig. 5.11b, respectively. The system description of the unified HA uses the zonotope *exMat method* as illustrated in Fig. 5.11c. As locations belonging to same groups have the same system behavior, they are represented with the same color in Fig. 5.11. Note that, zonotopes are used in Fig. 5.11c to hull the eigenvalues in each location, thereby considering the correlations between the eigenvalues in contrast to interval hulls. These zonotopes are later transformed to matrix zonotopes according to Eq. (5.4).

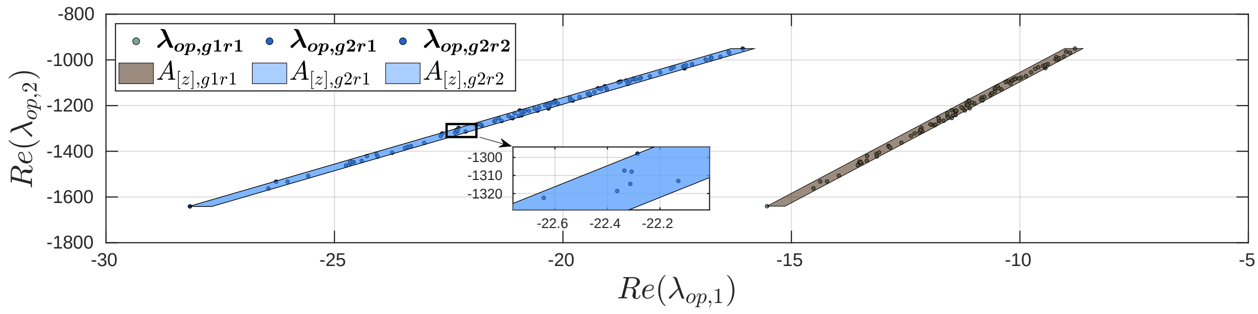
Instead of modeling the system behavior of the underlying HAs with matrices, the system behavior can be as well be modeled using the *exZ method*. In this case the generation of 100 HAs, which is executed in parallel, consumed 28.99 s. The unified HA was then created from these 100 models in 16.63 s. The result is illustrated in Fig. 5.12c. The matrix zonotopes of the underlying HAs are first transformed into zonotopes in $\mathbb{R}^4$. As the system has only real eigenvalues, the second and fourth dimensions of the zonotopes can be neglected (see Eq. (5.6)). In general each HA yields 4



(a)



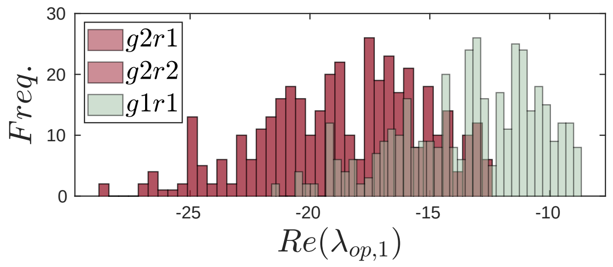
(b)



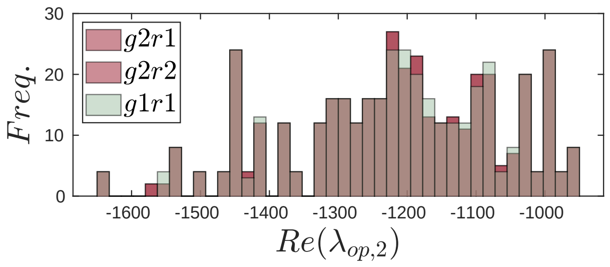
(c)

Fig. 5.11. The operating eigenvalues for each location are hulled by a zonotope as shown in (c) which is later transformed into a matrix zonotope. In (a) and (b) the distributions of the operating eigenvalues that describe the system matrices of the HAs are illustrated.

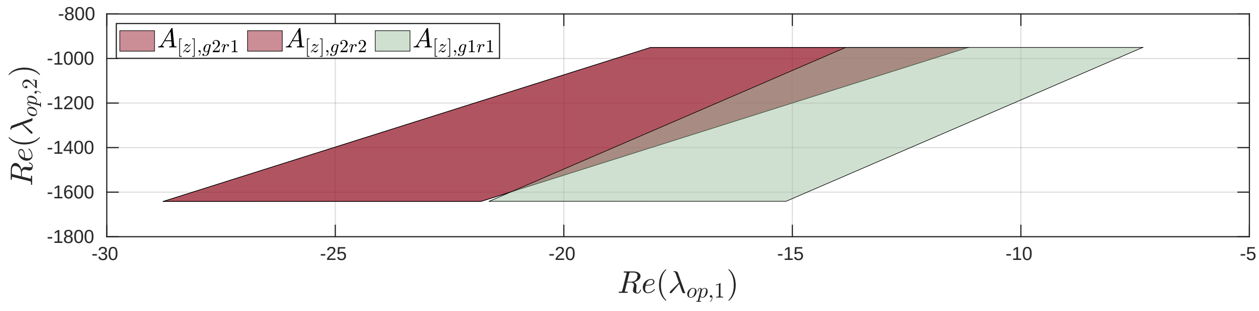
points, thus, 4 times the data is used for each location as represented in Fig. 5.12a and Fig. 5.12b. The obtained zonotopes are then transformed in each location to matrix zonotopes according to Eq. (5.4).



(a)



(b)



(c)

Fig. 5.12. Modeling the unified HA from 100 HAs whose system behaviors were described by matrix zonotopes.

Considering the invariants of the unified HA, for each location the invariant is found by hulling all the points of the invariants of the underlying HAs by a polytope. For the guards, the same procedure can be performed. The results are illustrated in Fig. 5.13a.

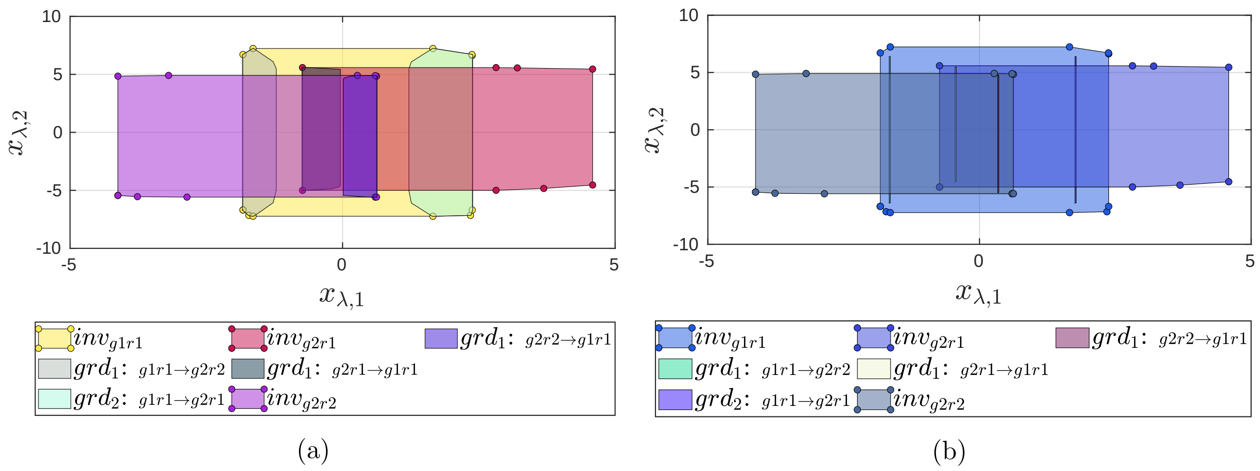


Fig. 5.13. Invariants and guards of the unified HA (a) with guards calculated from the underlying guards of the HAs or (b) with guards selected from the nominal model.

As *Cora* performs guard projections when the guards are intersected during a reachability analysis, the results can be over approximated in some cases during the projection into the target location, especially when the guards are cover large portions of the state space. For that, an additional option can be set to use the guards from the nominal model, thereby skipping the guard enlargement performed previously as illustrated in Fig. 5.13b.

For the running example from Section 4.1, three unified HAs were generated with the stated approach from this Section as described in Table 5.1. All three unified HAs were modeled using

Table 5.1: Unified HAs created

Model	reachability analysis	System description	Guards
HA_1	Fig. 5.14a	*exMat method*	merged
HA_2	Fig. 5.14b	*exZ method*	merged
HA_3	Fig. 5.14c	*exZ method*	nominal

100 underlying HAs. While HA_1 uses 100 HAs whose system behaviors were modeled by matrices, HA_2 and HA_3 use 100 HAs whose system behaviors were modeled by matrix zonotopes. Moreover, HA_1 and HA_2 merge the guards of the underlying HAs, while HA_3 takes the guards from the nominal model. Note that the previously stated modeling time of 16.63 s were measured for the generation of the HA_2. Since the HA_3 skips the merging of the guards, the modeling time is reduced slightly to 14.27 s. The results of the reachability analysis performed with *Cora* for a step function at $V_{nin} = 4$ V on the three models are illustrated in Fig. 5.14a, Fig. 5.14b, and Fig. 5.14c for HA_1, HA_2, and HA_3 respectively.

The over approximation of the reachable regions illustrated in Fig. 5.14 can be controlled by splitting up the previous used system description shown in Fig. 5.12c for the first location $g1r1$ as

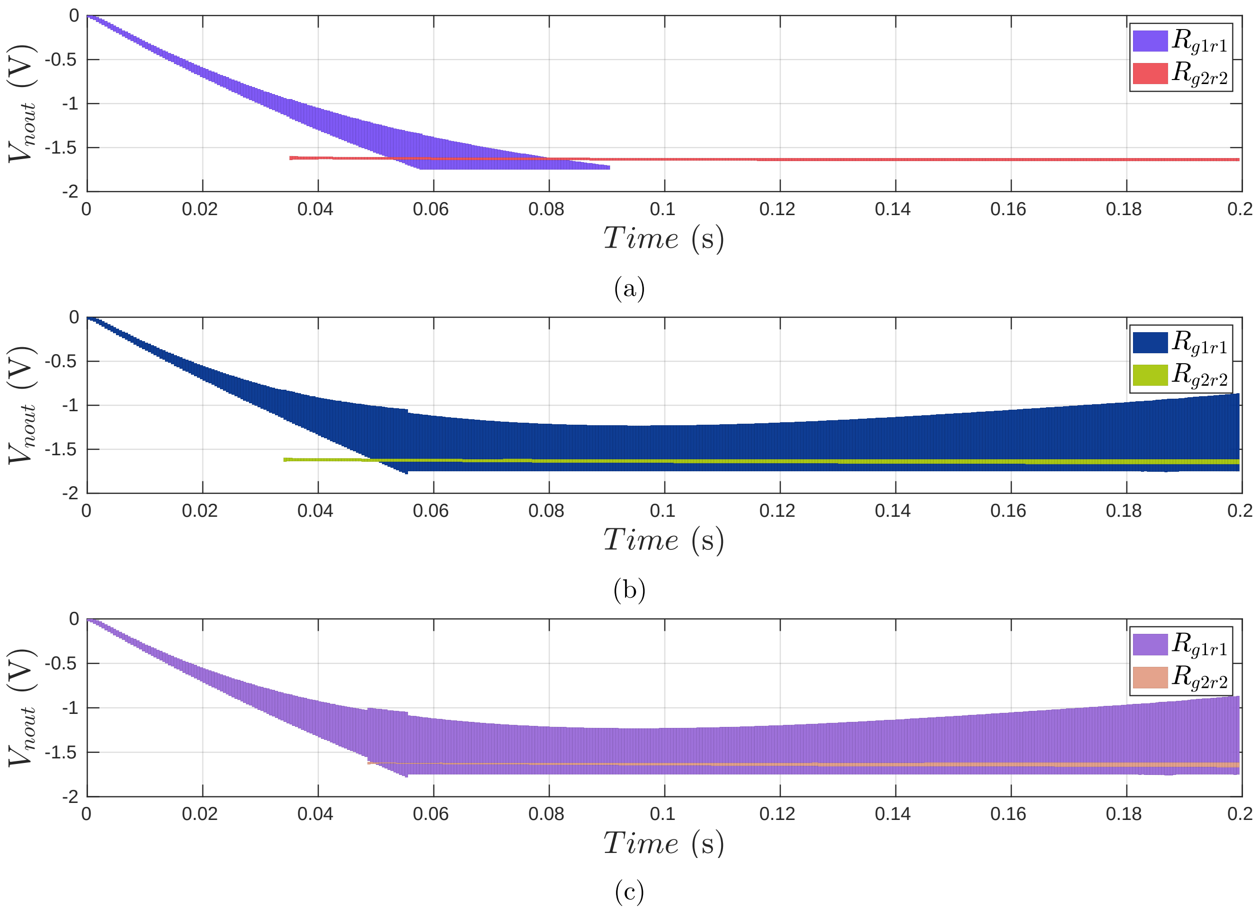


Fig. 5.14. Reachability analysis performed with *Cora* for the three generated unified HAs with different options for an input voltage step $V_{nin} = 4$ V.

illustrated in Fig. 5.15. As presented, the matrix zonotope is divided into four portions: a, b, c, and d. Note that the system description of the reaming locations is unchanged. This is performed

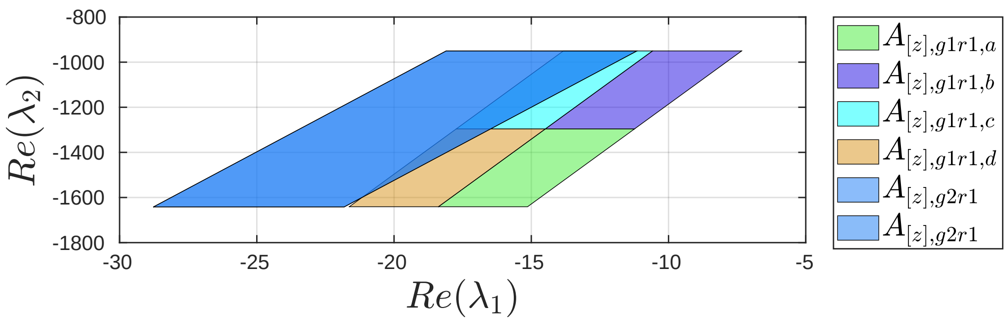
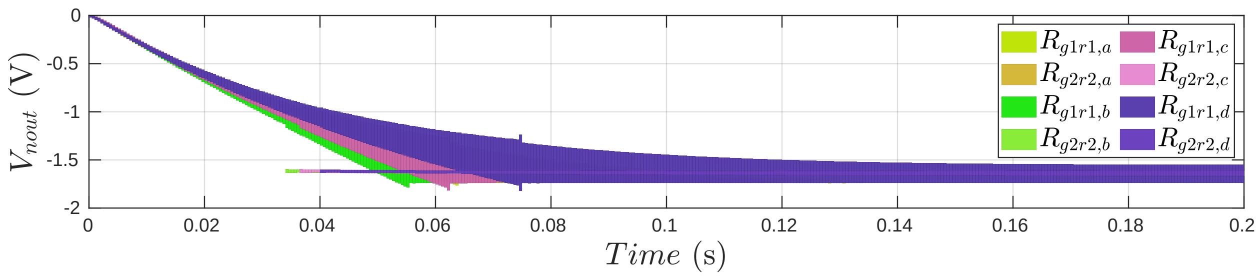
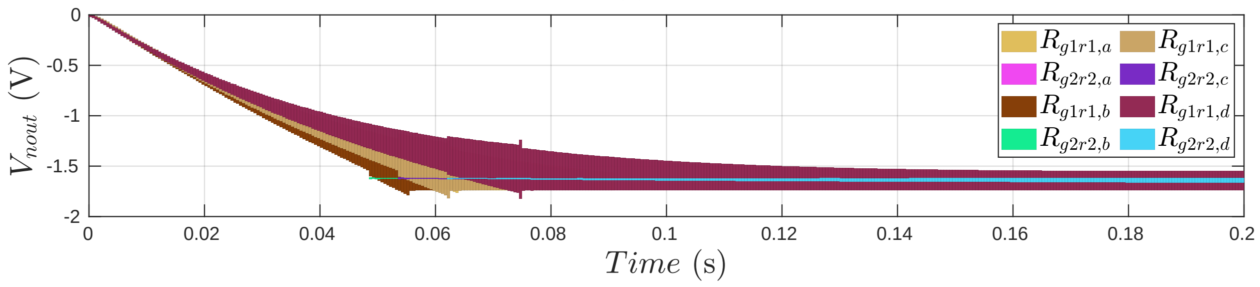


Fig. 5.15. Zonotopes that model the system behavior of the generated HA. Compared to Fig. 5.12c, the zonotope which is later transformed to the matrix zonotope of the $g1r1$ is divided into four portions.

for HA$_2$ and HA$_3$. Note that, while HA$_2$ merges the guards of the underlying HAs, HA$_3$ skips this step. The results for the same input voltage $V_{nin} = 4$ V are shown in Fig. 5.16. For each model, four simulation runs have been performed. In each run, only one portion (a, b, c, or d) for the system description as illustrated in Fig. 5.15 was used. As observed, the results are less over



(a)



(b)

Fig. 5.16. Reachability analysis performed in *Cora* for (a) HA_2 and (b) HA_3.

approximative for both models, HA_2 and HA_3, compared to the results presented in Fig. 5.14b and Fig. 5.14c, respectively. This is due to the better modeling of the correlations of the parameters.

5.3 Compositional HA

A compositional hybrid automaton (CHA) can be easily created from the Verilog-A and SystemC-AMS models, as the generated automatons are pin-wise compatible. This lies in the fact that the HAs are generated as modules with input and output pins. For the models used in *Cora*, the algorithm presented in Chapter 4 was extended to generate a compositional model from several sampled netlists.

Typically, when creating a CHA consisting of several parts, two types of connections should be distinguished: internal and external connections. Internal connections are between the components of the CHA as illustrated in red in Fig. 5.17, while external connections lie between the CHA and its surrounding test bench, like for example the input and output connections to and from the CHA. Note that the internal connections can be also feedback loops as will be handled in Section 7.4 and illustrated in Fig. 7.38.

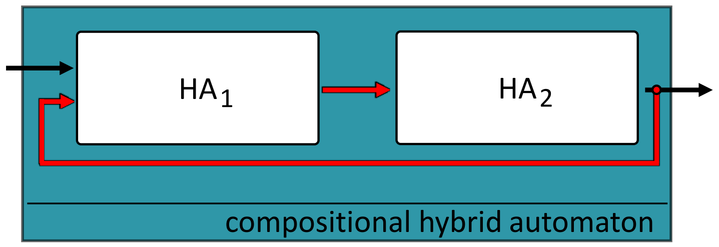


Fig. 5.17. A compositional HA consisting of two HAs.

The CHA in this section consists basically of HAs that are generated according to Chapter 4.

Additionally, these models are extended by output equations.

As stated in Chapter 4, the generated HAs are simulated in the $\mathcal{S}_\lambda$ space. The result is then transformed to the original state space of the system $\mathcal{S}_o$ via a back-transformation for each location. This happens for the *Cora* models post simulation. However, when looking at a CHA from a different perspective, the components of the CHA are connected in the original state space $\mathcal{S}_o$. More precisely, the interconnection between the components marked in red in Fig. 5.17 is in the $\mathcal{S}_o$ space, while the HAs are simulated in the $\mathcal{S}_\lambda$ space. Therefore, an output equation is added to each HA that basically performs the back-transformation of the simulated results into the $\mathcal{S}_o$ space during the reachability analysis of the CHA. However, since not all variables from the state vector $\boldsymbol{x}$ are needed, only the indexed variables of the output signals as provided to *Elsa*, are calculated during the reachability analysis. With $\boldsymbol{C}_{tmp} \in R^{p\times n}$ representing the output matrix computed according to the system description methods from Section 4.5.2, the output equations of each HA in a location *loc* is described via:

$$\begin{aligned} \boldsymbol{y} &= \boldsymbol{C}_{tmp}\boldsymbol{x} \\ &= \underbrace{\boldsymbol{C}_{tmp}\boldsymbol{F}_{loc}}_{\boldsymbol{C}_{loc}}(\boldsymbol{x}_\lambda - \boldsymbol{x}_{\lambda,op}) \underbrace{-\boldsymbol{C}_{tmp}\boldsymbol{L}_{loc}}_{\boldsymbol{D}_{loc}}(\boldsymbol{u} - \boldsymbol{u}_{op}) + \underbrace{\boldsymbol{C}_{tmp}\boldsymbol{x}_{op}}_{\boldsymbol{k}_{loc}} \end{aligned} \tag{5.7}$$

Where $\boldsymbol{C}_{loc} \in \mathbb{R}^{p\times m}$ represents the output matrix used with the $\boldsymbol{x}_\lambda$ state vector, while $\boldsymbol{D}_{loc} \in \mathbb{R}^{p\times k}$ and $\boldsymbol{k}_{loc} \in \mathbb{R}^p$ represent the feedthrough matrix and offset vector, respectively.

Each HA is thus described using Eqs. (4.10, 5.7). The CHA is initialized using the *parallelHybridAutomaton* class of *Cora*. A reachability analysis can now be performed on the CHA. The results in the $\mathcal{S}_\lambda$ are then transformed into the $\mathcal{S}_o$ space via the back-transformation from Eq. (4.11) for each HA separately. Note that this time all $\boldsymbol{x}$ values are calculated.

The back-transformations of the obtained results need special attention, especially if the system contains feedback loops. At the current time, *Cora* does only yield the states of the system ($\boldsymbol{x}_\lambda$) for the specified inputs of the CHA. Specifically, the internal connections are not included in the results. In order to execute the back-transformation according to Eq. (4.11) for each of the underlying HAs of the CHA, the inputs of each HA must be known. Using Eq. (5.7), the input from an internal connection can be calculated from the corresponding output of the HA at the starting point of this connection. For example, considering Fig. 5.17, the input to HA$_2$ can be calculated from the output of HA$_1$. However, in case the compositional system contains a feedback loop, this is only possible if feedthrough matrix $\boldsymbol{D}_{loc}$ is a zero matrix. That is, with $\boldsymbol{L}_{loc}$ calculated according to Section 4.5.2:

$$\begin{aligned} \boldsymbol{D}_{loc} &= -\boldsymbol{C}_{tmp}\boldsymbol{L}_{loc} \\ &= -\boldsymbol{C}_{tmp}\boldsymbol{F}_\infty\boldsymbol{H}_\infty\boldsymbol{B}_{\infty,red}\,, \end{aligned}$$

must be a zero matrix, which is usually the case. Hence, in case the system contains a feedback loop, the output of the subsystem at the starting point of the feedback loop is first computed. An complex example from the automotive industry, in which the components were connected in a control loop, is provided in Section 7.4.

To illustrate the basic process of building a CHA, two HAs generated from the example from Section 4.2 are connected in series as presented in Fig. 5.18. The invariants of both HAs were

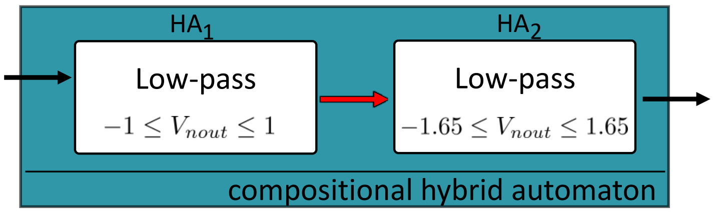

HA$_1$
HA$_2$
Low-pass
$-1 \leq V_{nout} \leq 1$
Low-pass
$-1.65 \leq V_{nout} \leq 1.65$
compositional hybrid automaton

Fig. 5.18. A compositional HA consisting of two lowpass filters connected in series.

modeled as intervals, while the guards were modeled as polytopes identified in the $\mathcal{S}_{virt}$ space using the *distance method*. HA$_1$ represents the same circuit from Section 4.2 with the difference that V_{dd} is set to +1 V, while V_{ss} is set to −1 V, thereby limiting V_{nout} to the range $[-1, 1]$ V. For HA$_2$ on the other hand, the capacitors C1 and C2 are set to 0.001 μF and 0.01 μF, while the reaming of the circuit is identical to the circuit from Section 4.2. The output voltage V_{nout} of HA$_1$ represents the input to HA$_2$ at V_{nin}. Note that due to the limiting behavior of HA$_1$, HA$_2$ will never go into the limiting behavior. More precisely, the locations $g2r1$ and $g2r2$ of HA$_2$ will never be reached.

A reachability analysis was conducted in *Cora* for an input voltage of 3 V and an uncertain initial range $[-0.01, 0.01]$ for the $\boldsymbol{x}_\lambda$ values of both HAs. The result at the output of HA$_1$ is illustrated in Fig. 5.19a, while the result of the output of HA$_2$ is represented in Fig. 5.19b. For reference, a Spice circuit was constructed that contains both circuits connected in series. The result of the Spice simulation at the corresponding output nodes is illustrated for each HA in Fig. 5.19.

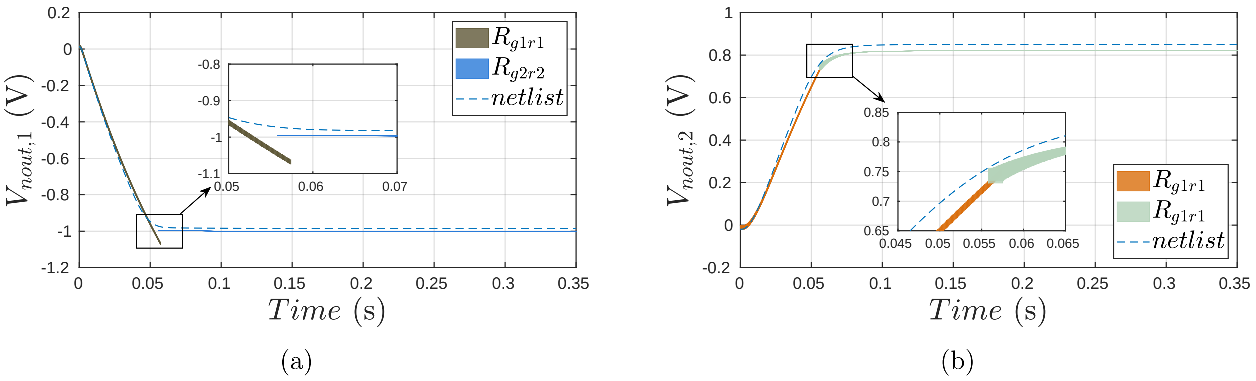


Fig. 5.19. Reachability analysis performed with *Cora* on the created CHA model. The voltage at the output (a) of HA$_1$ and (b) of HA$_2$ are illustrated. Note that R_i represent the reachable sets computed in *Cora* for HA$_i$, while $V_{nout,i}$ represents the corresponding Spice simulation.

As observed, HA$_1$ switches the location from $g1r1$ to $g2r2$ (Fig. 5.19a), while HA$_2$ remains in the same location $g1r1$ during the entire analysis. In Fig. 5.19b, the zonotopes are colored in orange before and in green after HA$_1$ switches the location. HA$_2$ does not change the location. As illustrated, both HAs exhibit system behaviors close to the behavior of the real circuit.

Using this approach, a large circuit can be abstracted to a CHA by abstracting their underlying subcircuits to HAs, permitting thereby scalability to the HAs generated for *Cora*.

6 Formal Verification

Analog verification is a promising research field with a long history. At its simplest, the aim of the verification task is to prove that a circuit behaves as desired by the designer. As simple as this sounds, the analog verification task is still an open problem. Traditionally, verification is often used in the context of testing, fault analysis, and error detection. This type of verification is referred to as simulation based verification, which establishes the design correctness using simulation results. Even though these computational expensive analysis and simulations deliver a solid understanding in the functionality of the developed circuit, they do not verify the full system behavior, in contrast to formal verification. Formal verification aims to prove the correct functionality of the system with respect to formal specifications or a golden model, using formal methods. Hence, formal verification uses mathematical proofing methods to ensure that the circuit matches the specified specifications.

As illustrated in Fig. 1.1, there are several formal verification methods available in the AMS domain. As mentioned in Chapter 1, formal verification methods suffer mainly from the state space explosion. On top of this problem, there is a continuous explosion, as analog signals can attain variable values from the continuous domain. Compared to the two possible values a digital variable can attain, there are infinite many values for an analog variable. Hence, analog verification approaches must overcome both problems, which is a challenging task.

On promising approach that permits verification of large circuits is behavioral abstraction. On one hand, accurate abstracted models are mandatory for the verification task. As the verification of complex AMS systems heavily relies on the use of behavioral models [GGKF18], accurate abstracted models can speed-up simulation routines. On the other hand, behavior models are often abstracted to a degree that they do not accommodate the full system behavior nor the circuit specific parameter changes due to the process parameters or the operating conditions (temperature, process variations, EMI, . . .).

In this dissertation, we aim to use the accurate models from Chapter 4 and Chapter 5 in various verification routines. But before using these models, we aim to formally verify their functionality, thereby assuring a correct model abstraction. Hence, the generated models can then be considered correct by construction.

In the following, the generated Verilog-A models are formally verified against the original Spice netlist in Section 6.1. In the remaining of this chapter, several verification methods will be explored that can be applied on the generated models from Chapter 4, as well as partially on the models from Chapter 5.

6.1 Equivalence Checking

To close the tool chain, the generated abstract models can be verified against their original Spice or Verilog-A netlists. As stated in Chapter 3, *Vera* [SH10b] can be used to perform an equivalence checking (EC) in the analog domain. In this process, a Verilog-A generated abstract model as well as the original netlist (Spice or Verilog-A) are provided to *Vera*. As *Vera* supports Verilog-A models, the generated abstract model in this syntax can be directly used.

To illustrate this process, the abstract model generated from the running example from Section 4.2 is compared to the original netlist (Listing C.1). The model is generated by sampling 18444 points using *Vera*. This time, the sampling has been performed with little more effort by specifying a different sampling method in *Vera*. This results in a $\mathcal{S}_\lambda$ space with minimal overlapping locations. With the data points at hand, an abstract model has been generated using *Elsa*. To simplify the modeling approach, the invariants were modeled as intervals, while the guards were specified as halfspaces. Due to the extra effort spend during sampling, the guards were directly identified in the $\mathcal{S}_\lambda$ space using the *distance method*. Note that the *lsq filter* was used on the $\mathcal{S}_\lambda$ space. The result of the modeling approach with three locations $Loc = \{g1r1, g2r1, g2r2\}$ is presented in Fig. 6.1. Using the *grdV method*, the model was deployed in Verilog-A syntax, with a system behavior described using the *mean method*.

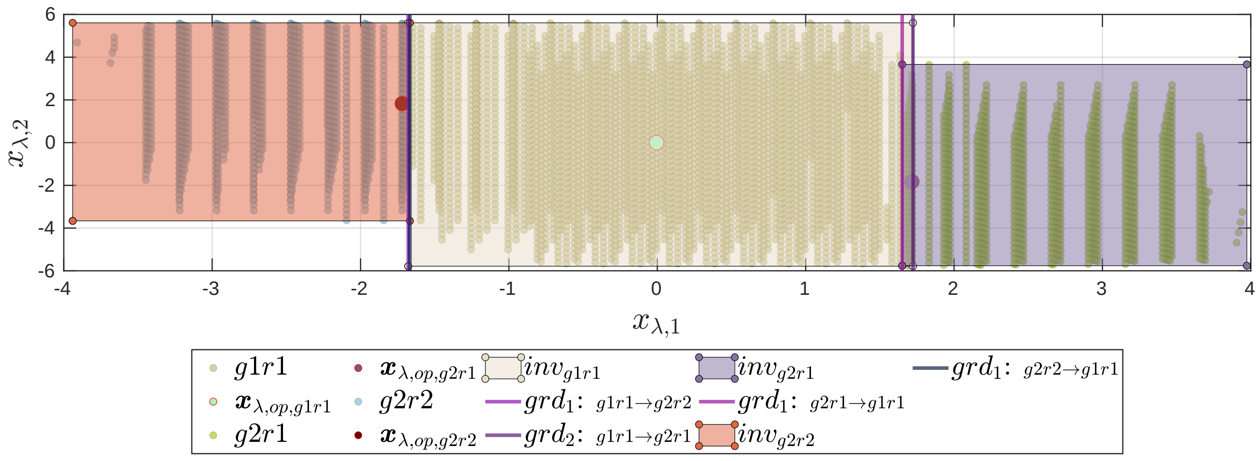


Fig. 6.1. $\mathcal{S}_\lambda$ space of the abstract Verilog-A model generated with 3 locations, invariants modeled as intervals, and guards modeled as halfspaces.

This behavior model is passed to *Vera* along with the original netlist. The tool then executes the described EC algorithm as presented at the end of Section 3.3. The results in the $\mathcal{S}_{virt}$ space versus the relative errors are illustrated in Fig. 6.2, with the first row representing the relative output error, while the second one shows the relative derivative error. The detailed results of the EC are stated in Table 6.1.

Table 6.1: Modeling errors of a HA with 3 locations

Sampled points	$max(\delta_{\dot{x}})$ (V)	$max(\delta_{\dot{x},r})$ (%)	$max(\delta_y)$ (V)	$max(\delta_{y,r})$ (%)	$mean(\delta_y)$ (V)
19096	27.65	1.845	0.0668	3.959	0.0036

As observed in Fig. 6.2 and stated in Table 6.1, the output error $\delta_{y,r} \leq 3.95$ % and the derivative error $\delta_{\dot{x},r} \leq 1.84$ % are relative small. Moreover, the maximum error at the output computed is at $max(\delta_y) = 0.0668$ V. In addition to the errors from Eqs. (3.29, 3.30), the mean error $mean(\delta_y)$,

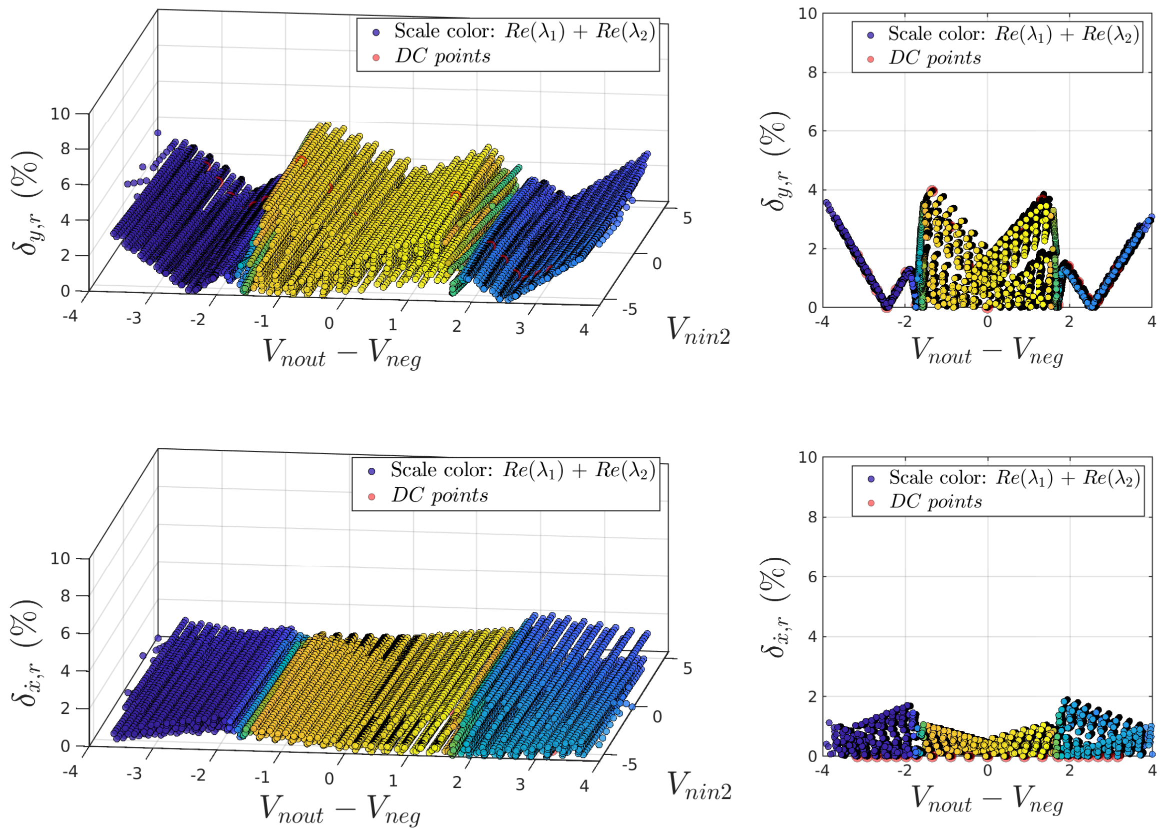
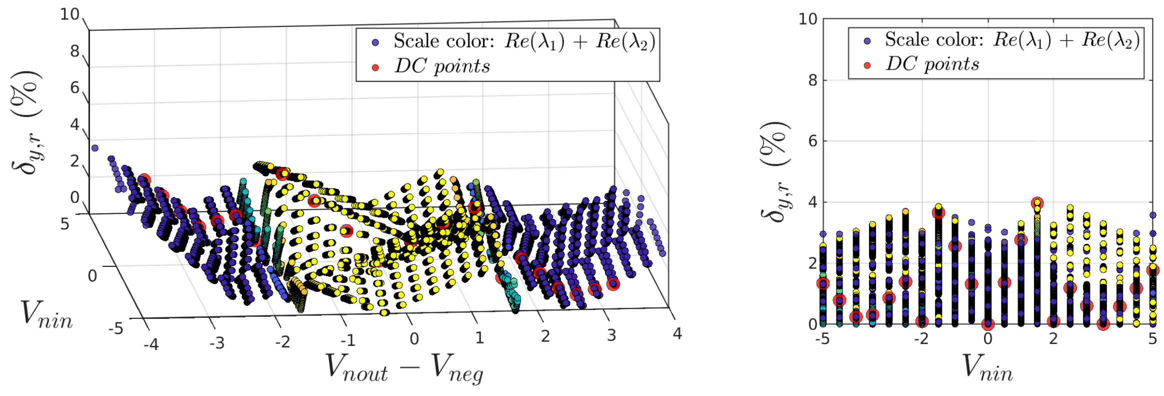


Fig. 6.2. EC of the Verilog-A generated model with 3 locations versus the original netlist in Spice syntax. The first row shows the relative output error, while the second one shows the relative derivative error, illustrated both versus the $\mathcal{S}_{virt}$ space.

which is computed by calculating the mean of the output error δ_y over all sampled data point, is stated in Table 6.1.

At the operating points, these errors are at their minimums as is illustrated in Fig. 6.3. Specif-



(a) (b)

Fig. 6.3. Result of the EC shown (a) versus the input voltage V_{nin} and the voltage $V_{nout} - V_{neg}$, and (b) versus only the input voltage V_{nin}. The large red dots indicate the DC points.

ically, in Fig. 6.3 the relative output error $\delta_{y,r}$ is illustrated against the input V_{nin} and one of the two variables from the $\mathcal{S}_{virt}$ space. The large red dots indicate the DC points which were calculated by *Vera*. These points are spaced 0.5 V apart. The HA with three locations $g1r1$, $g2r1$, and $g2r2$, has its operating points at $V_{nin} = 0$, -2, and 2 V, respectively. As shown in Fig. 6.3b, the relative output error $\delta_{y,r}$ is at its lowest at the operating points. In general, the further the system moves from the operating points, the greater the previous stated errors become. Thus, increasing the number of locations of the HA should result in more accurate models.

To emphasize this point, an abstracted model was created with the same settings as previously, with the difference that instead of modeling the HA with 3 locations, the HA was modeled with 5 locations $Loc = \{g1r1, g2r1, g2r2, g3r1, g3r2\}$ as illustrated in Fig. 6.4.

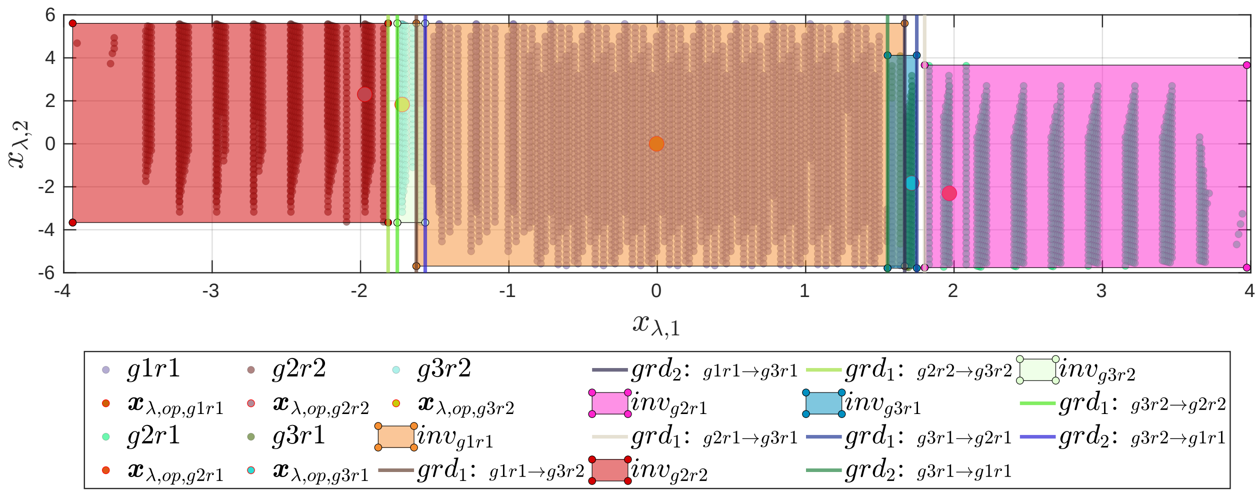


Fig. 6.4. $\mathcal{S}_\lambda$ space of the abstract Verilog-A model generated with 5 locations.

As observed in Fig. 6.4, two intermediate locations $g3r1$ and $g3r2$ were additionally identified by dividing the state space this time into 5 locations instead of 3 locations. The group identification (Section 4.4.1) first identified 3 locations via eigenvalue clustering, followed by the region identification (Section 4.4.2), which used the *Gdist method* to yield for each of the groups $g2$ and $g3$ two regions.

Similarly to the previous analysis, the results of the EC with the original Spice netlist are presented in Fig. 6.5 and Table 6.2.

Table 6.2: Modeling errors of a HA with 5 locations

Sampled points	$max(\delta_{\dot{x}})$ (V)	$max(\delta_{\dot{x},r})$ (%)	$max(\delta_y)$ (V)	$max(\delta_{y,r})$ (%)	$mean(\delta_y)$ (V)
20324	73.19	1.324	0.04721	2.803	0.0034

As observed, both errors decreased compared to the EC of the model with 3 locations; the relative output error is now $\delta_{y,r} \leq 2.80$ % while the relative derivative error is $\delta_{\dot{x},r} \leq 1.32$ %. Moreover, the maximum output error is at $max(\delta_y) = 0.0472$ V. These results make sense, as with more linearized locations the system behavior becomes more accurate and thus closer to the behavior of the original circuit.

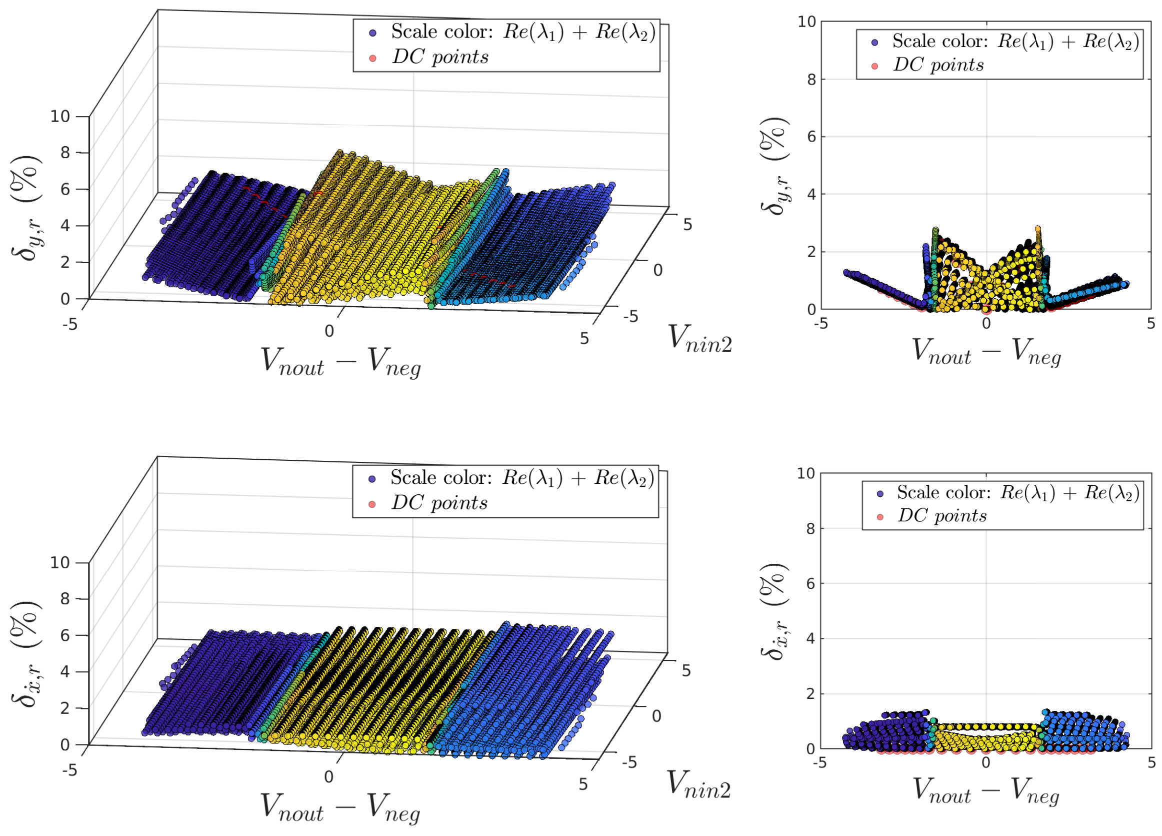


Fig. 6.5. EC of the Verilog-A generated HA with 5 locations versus the original Spice netlist, illustrated in the $\mathcal{S}_{virt}$ space.

With an increasing number of locations, the modeling error seems to be decreasing. To examine this statement, a third model with 7 locations (4 groups) was generated as illustrated in Fig. 6.6. Similarly to the 5-location version, an additional group was identified in the portions of the state space were the eigenvalues change rapidly (see Fig. 4.7).

The results of the EC between the 7-location model with the original Spice netlist are presented in Fig. 6.7 and in Table 6.3.

Table 6.3: Modeling errors of a HA with 7 locations

Sampled points	$max(\delta_{\dot{x}})$ (V)	$max(\delta_{\dot{x},r})$ (%)	$max(\delta_y)$ (V)	$max(\delta_{y,r})$ (%)	$mean(\delta_y)$ (V)
19282	73.2	1.324	0.0303	1.804	0.0030

As Table 6.3 reveals, the relative output error $\delta_{y,r}$ as well as the output error δ_y decreased compared to the errors of the previous generated models.

On a Linux machine with an i5-7300HQ CPU at 2.50 GHz with 16 GB of RAM, the model generation via *Elsa* consumed 14.43 s for the 3-location version, 18.32 s for the 5-location version, and 18.48 s for the 7-location version. On top of the modeling time of *Elsa*, the sampling time consumed by *Vera* must be considered. The circuit was sampled in 130, 11 s with 18444 sampled

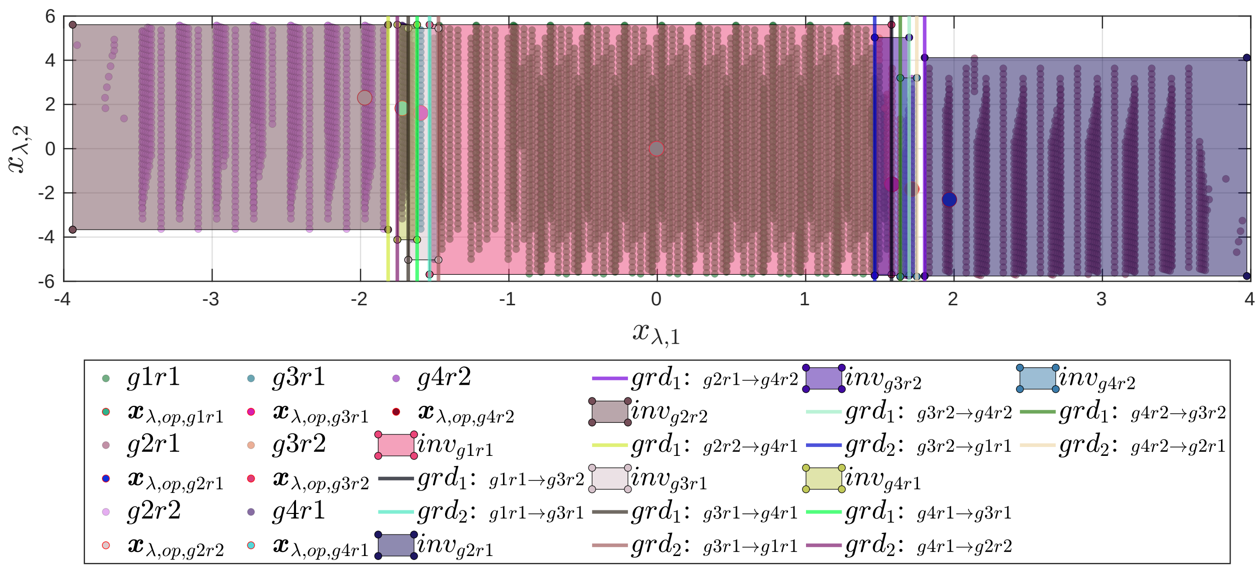


Fig. 6.6. $\mathcal{S}_\lambda$ space of the abstract Verilog-A model generated with 7 locations.

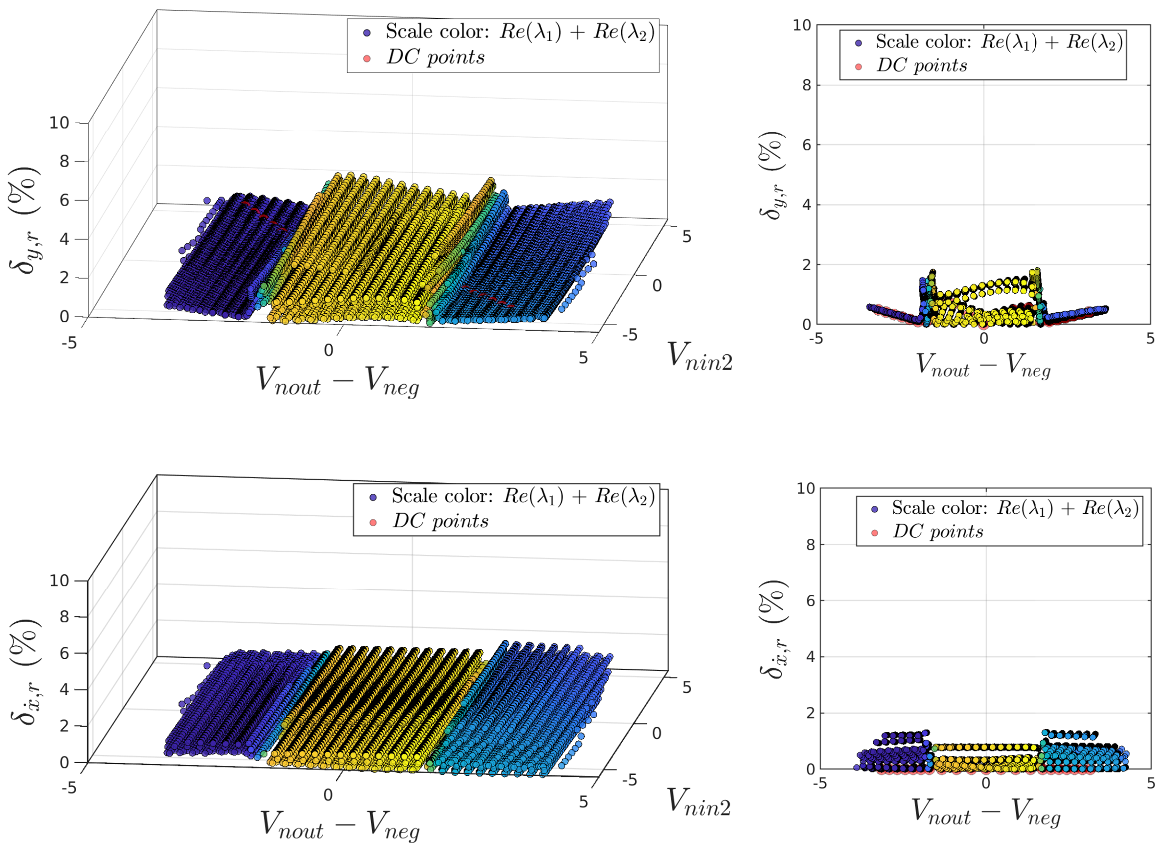


Fig. 6.7. EC of the Verilog-A generated HA with 7 locations versus the original Spice netlist.

data points. This data was imported into Matlab memory in 2.51 s, thereby the whole modeling process consumed in total 147.05 s for the 3-location, 150.94 s for the 5-location, and 151.10 s for the 7-location version.

Considering the EC of the netlist and the Verilog-A models, *Vera* consumed nearly the same time, which is $159,39$ s for the EC of all three models. In view of the discretizing nature of *Vera* presented due to the state space stepping through a continuous state space, fault can be still presented between the sampled points. Decreasing the step size yields a higher confidence in the obtained results. Another approach that uses the models generated by this dissertation was presented in the [TKR+20]. In this contribution reachable sets were used to perform EC between two circuits in the continuous state space of the system. However, since that the original netlist is in Spice, the algorithm could not perform an EC between the original netlist and the abstracted model. Hence, a model generated with *Elsa* was trained with the Spice simulations and the real circuit measurements to obtain a conformant model [KTR+20]. In Fig. 6.8, the deviation between the outputs of the models at V_{nout} is illustrated. As observed in Fig. 6.8, EC was then

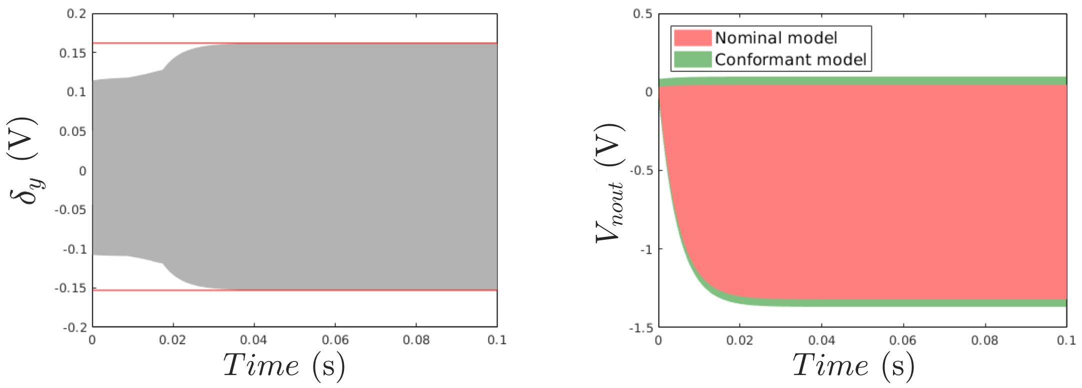


Fig. 6.8. The difference between the output of the conformant model and the abstract HA it was generated from for an input range $V_{nin} = [0, 4]$ V [TKR+20].

performed between the conformant model and the abstract HA used to train this model. Hence, using continuous reachable sets, EC has been executed in the continuous domain, combining thereby reachability analysis with EC. For more details see [TKR+20].

Using the approach stated in this section, the equivalence of the netlist and the abstracted models can be verified. Moreover, with this process the error between the abstracted model and the original netlist can be calculated. Note that, due to the definition of the errors (Eqs. (3.29, 3.30)) and the nature of the domain where the EC is executed, a 0 % error is in general not feasible. However, as the results in Section 7.1.5 show, models generated with error bounded to 3.9 % usually yield good results.

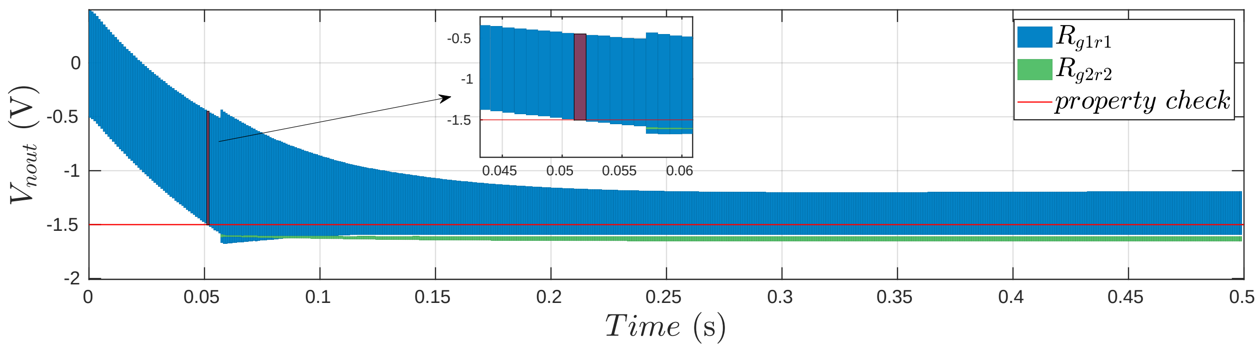
In general, this applies to the Verilog-A models, as these models were checked for equivalence with the original netlist. For the Matlab models, a different approach has to be chosen, as the one handled in the next section. Note that in the case halfspace guards were used, the Verilog-A models exhibit nearly identical behaviors to the *Cora* models. Thus, the results validated on a Verilog-A model could be generalized in specific cases for the SystemC-AMS and *Cora* models. The differences that arise in the behavior of these models results from the tool used to perform a simulation/reachability analysis.

6.2 Reachability Analysis and Model Checking

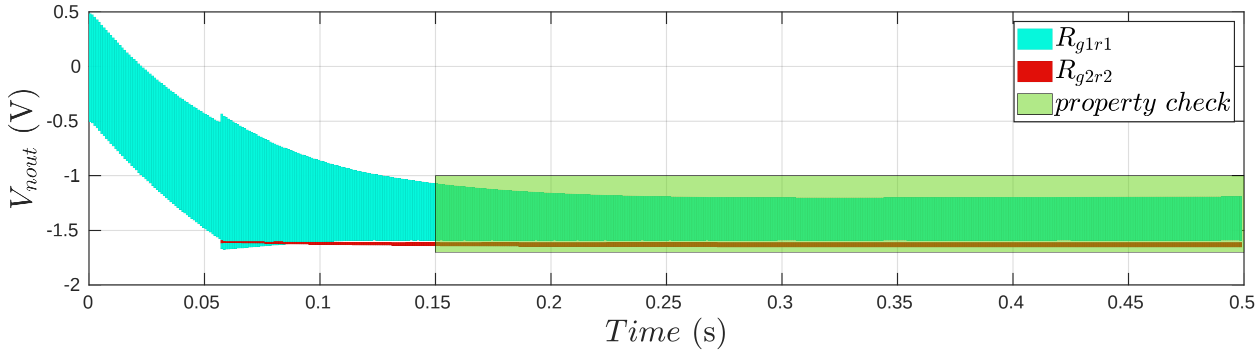
For the in Matlab syntax created HAs, a reachability analysis can be conducted using *Cora*. On top of this, a post simulation model checking algorithm has been implemented. The algorithm basically detects intersections between the reachable set of a dimension in the $\mathcal{S}_o$ space with a specified condition. The conditions can be either specified as top or bottom margins, or specific regions of interest.

Even though the reachability analysis could verify the system behavior on its own, performing a post simulation model checking (MC) allows for the interpretation of the results with and without generating plots. This comes in handy when a small step size is specified for the reachability analysis, thereby increasing the time needed to plot the results.

To demonstrate this, the model with three locations from Section 6.1 is used. This time however, the guards are modeled as polytopes and the system is described via a matrix zonotope that, as described in Section 5.1.1, encloses (over approximates) the whole abstracted system behavior. The model is deployed in Matlab (*Cora*) syntax. With the initial state space vector of $\boldsymbol{x}_\lambda$ lying in $[-0.5, 0.5]$ for each dimension, and an input voltage of $V_{nin} = 4$ V, the MC algorithm is launched to check that the output voltage V_{nout} is always greater than -1.5 V as shown in Fig. 6.9a. The MC algorithm detects from the reachable set the zonotope at which this condition fails. In this case, the time of failure is returned. In Fig. 6.9b the model checking approach has be launched to check that the output voltage is always bounded by the interval $[-1.7, -1]$ after the time $t = 0.15$ s. In



(a)



(b)

Fig. 6.9. Reachability analysis in *Cora* with post simulation MC.

TCTL that is:

$$AG_{[0.15,0.5]}(-1.7 \leq V_{nout} \leq -1)$$

As observed in Fig. 6.9b, the algorithm detects that the condition is satisfied.

This approach can be improved by performing the MC during the runtime of *Cora*. However, at the current time, the model checking is performed after the reachability analysis performed by this tool.

6.3 Runtime Verification

For the formal verification using online monitors of the SystemC-AMS models, an CTL-A extension has been developed in [Dra20]. This part summarizes the main contributions. Note that for the Verilog-A models, a similar study was conducted in [Wag17].

Compared to traditional CTL, CTL-A is first extended by the analog operators greater than ($>$) and less than ($<$) [HHB02]. Moreover, the introduced extension contains additional to the states true (T) and false (F), the unknown (X) state, which is neither true nor false. Additionally, three temporal operators can be used:

- Once $O_{[t_{start},t_{stop}]}(expr)$: if the condition ($expr$) up till the current time was at least once valid, the operator returns intermediately true
- Historically $H_{[t_{start},t_{stop}]}(expr)$: if the condition up till the current time was and is still valid, the operator returns true at t_{stop}. If this is not the case, the operator intermediately returns false
- Since $(expr1)S_{[t_{start},t_{stop}]}(expr2)$: if $(expr2)$ was true during the time interval, and during this time step or intermediately after $(expr1)$ becomes valid and stays valid till t_{stop}, the operator returns true at t_{stop}

There are two modes for the time that can be used with the temporal operators. Either an interval $[t_{start}, t_{stop}]$ is specified, or a duration $t_{duration}$. The previously stated descriptions of the operators were explained for the interval mode. Before the monitor reaches t_{start}, all operators return the unknown state X. After that, each operator returns the state according to the expression. In case the duration mode was used, the Once-operator $O_{t_{duration}}$ returns intermediately true if the expression is valid. During the entire duration, the operator returns true. In case the expression is again valid, the operator returns true till the end of the duration. For the Historically-operator, the operator returns true at t_i if the expression is valid during $[t_{i-duration}, t_i]$. Hence, the Historically-operator checks the expression in a time window from the current time till a duration back in the past. Similarly for the Since-operator, for the current time t_i if $expr2$ is true or was true until $expr1$ became true in the time interval $[t_{i-duration}, t_i]$, the operator returns true. However, if $expr1$ was true and in case $expr2$ becomes false during the current inspected time window, the operator returns false at t_i.

The online monitors are realized as automatons with several states. The automaton for the Once-operator in the interval mode ($O_{[t_{start},t_{stop}]}(expr)$) is illustrated in Fig. 6.10. The detailed description will be skipped here. Depending on the time as well as the evaluation of the expression, the automaton from Fig. 6.10 switches states. This automaton is attached to the SystemC-AMS

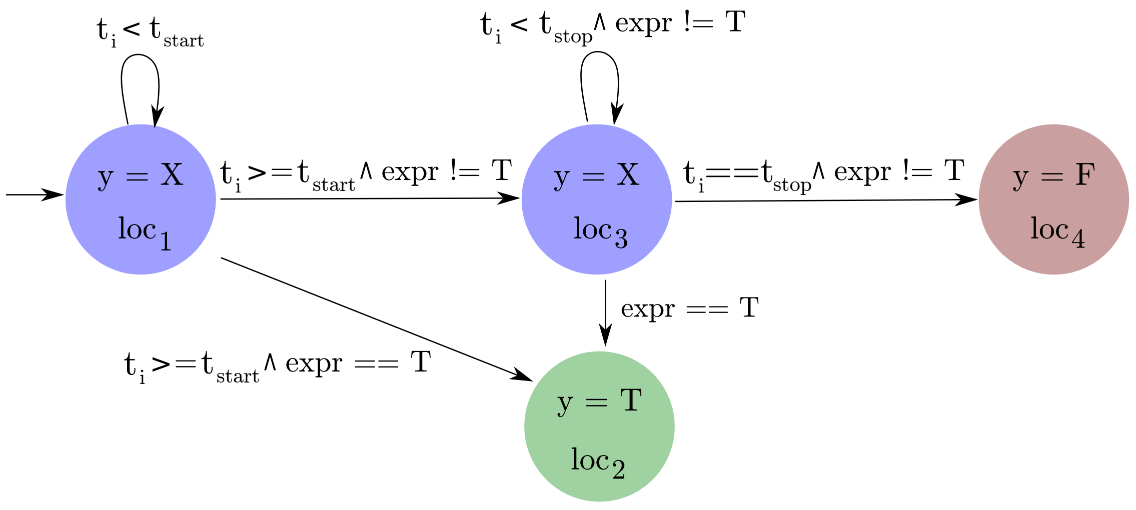
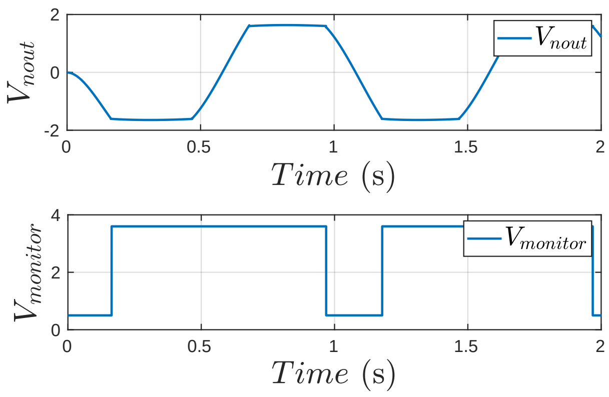

$t_i < t_{start}$
$t_i < t_{stop} \wedge$ expr != T
y = X
loc$_1$
$t_i >= t_{start} \wedge$ expr != T
y = X
loc$_3$
$t_i == t_{stop} \wedge$ expr != T
y = F
loc$_4$
expr == T
$t_i >= t_{start} \wedge$ expr == T
y = T
loc$_2$

Fig. 6.10. Finite state automaton for the Once-operator in interval time mode [Dra20].

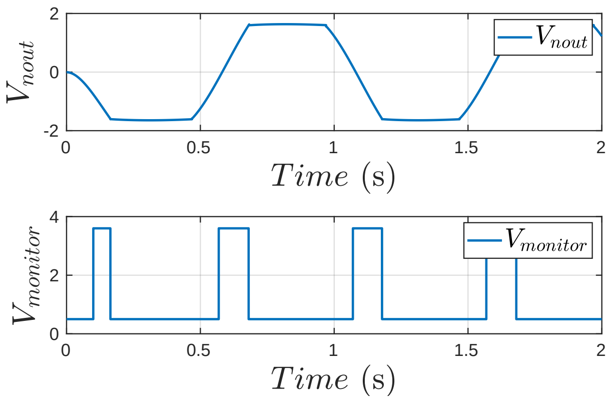
simulation as a header file. Hence, during simulation the monitor evaluates the specified specifications and assigns the value of the output y accordingly.

The developed monitors support SystemC-AMS simulations with affine forms. The monitors are extended using the AADD library [RGJR17]. This is done by using affine forms in the expressions (*doubleS*), enhancing the guards conditions (*ifS* and *elseS*), and tracing the outputs of the monitors with min and max values. An output voltage of 0.5 V at the output of the monitors corresponds to a logical *false*, a voltage of 3.6 V corresponds to a logical *true*, while a voltage of 1.8 V is obtained if the expression is still not evaluated, thus corresponding to undefined (X) state.

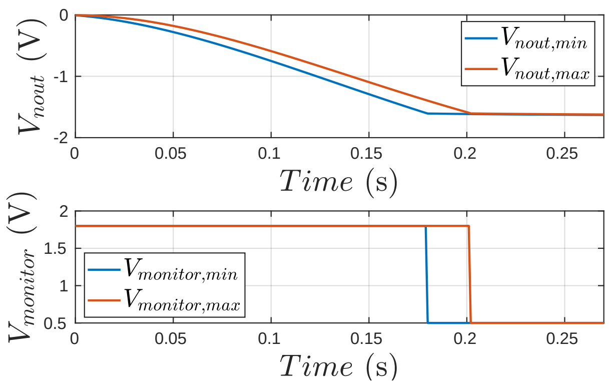
In Fig. 6.11, the results of several SystemC-AMS simulations, including online monitors, are illustrated. In all four simulations, the HAs were generated from the running example from Section 4.1 in SystemC-AMS syntax. The simulations in Fig.6.11a and Fig. 6.11b correspond to a HA generated according to Section 7.1.6 with an input voltage $V_{nin} = 4 \cdot sin(2\pi \cdot t)$ V. In Fig. 6.11c and Fig. 6.11d, the HA used was generated with affine forms according to Section 5.1.2 (see Fig. 5.5) and symbolically simulated in with $V_{nin}{=}3 \cdot sin(2\pi \cdot t) + 0.15\beta$ V. For all simulations, the output of the HA (V_{nout}) as well as the output of the monitors $(V_{monitor})$ are illustrated. In Fig. 6.11a, the output of the system is monitored with $O_{[0.5]}(V_{nout} <= -1.6)$. As a duration is specified with the temporal operator Once, the monitor returns intermediately false as the specified expression is false. At the instance V_{nout} becomes smaller than -1.6 V, the monitor returns true and stays true as long as V_{nout} was once smaller than -1.6 V in the last 0.5 s from the current simulation time. If this in not the case, the monitor evaluates to false. In Fig. 6.11b, the Historically operator is used in the monitor $H_{[0.1]}((V_{nout} > -1.6) \& (V_{nout} < 1.6))$. As the monitor is set in duration mode, if the expression returns true in the last 0.1 s from the current simulation time t, the monitor returns true. In contrast to the previous simulations, a symbolic simulation was performed as illustrated in Fig. 6.11c. The temporal operator Historically is used in interval mode in the monitor $H_{[0,0.25]}(V_{nout} > -1.6)$. At start, the monitor returns X and intermediately evaluates to false at the instance the expression becomes invalid. As an affine monitor is used, the expression is evaluated with affine forms, and two values corresponding to the maximum and minimum evaluations of the expression are returned. Several operators can be cascaded, for example as the monitor shown in Fig. 6.11d which examines $(H_{[0.1]}(V_{nout} > -1))S_{[0.05]}(V_{neg} < 0.1)$. The operator Since is used in duration mode. The monitor returns true if in a time window of 0.05 s from the current inspected time $V_{neg} < 0.1$. This condition is true until $t = 0.18$ s. At this instance, the expression $H_{[0.1]}(V_{nout} > -1)$ is examined. The monitor returns true as long as in a time window of 0.05 s the condition $(V_{neg} < 0.1)$ is true, or was once true in this window until $H_{[0.1]}(V_{nout} > -1)$ became true. However, at the instance the condition concerning V_{neg} is false, the condition about V_{nout} is always false. At $t = 0.2$ s, the condition involving V_{neg} becomes as well false. At $t = 0.201$ s, with a simulation time step is set to 10^{-3} s, the monitor returns false, as the condition $V_{neg} < 0.1$ is not valid in the time window of [0.151,0.201] with the duration of the monitor set to $t_{duration} = 0.05$ s.



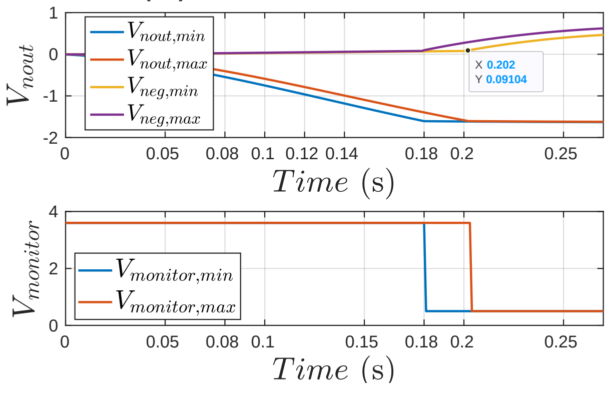
(a) $O_{[0.5]}(V_{nout} <= -1.6)$



(b) $H_{[0.1]}((V_{nout} > -1.6)\&(V_{nout} < 1.6))$



(c) $H_{[0,0.25]}(V_{nout} > -1.6)$



(d) $(H_{[0.1]}(V_{nout} > -1))S_{[0.05]}(V_{neg} < 0.1)$

Fig. 6.11. Online monitoring the output of the HA. The first row shows standard monitors, while the second row shows affine monitors.

As the example demonstrates, the monitors can be used for the runtime verification of the generated abstract models.

7 Experimental Results

This chapter demonstrated the abstraction approach from Chapter 4 upon various examples. Starting with the sampling of various transistor level circuits with full BSIM4 accuracy, several models are generated in different programming languages and simulated with the corresponding simulation tool. The Verilog-A models are first compiled using the Verilog-A compiler ADMSXml [LMH02], then simulated using the Spice simulator Gnucap [Dav03]. For the model generated in Matlab (*Cora*) syntax, a reachability analysis is performed using *Cora* [Alt15]. The SystemC-AMS models are simulated using the standard SystemC (2.3.3) and SystemC-AMS (2.1) libraries. The SystemC-AMS HAs which are modeled with affine forms (Section 5.1.2) additionally use the AADD library from [RGJR17]. All simulations were performed on a Linux PC with a four core i5-7300HQ CPU at 2.50 GHz with 16 GB of RAM.

An overview of the handled examples and the used methodologies are listed in Table 7.1. For all examples listed in Table 7.1, the groups of the locations are identified with eigenvalue clustering.

In Section 7.1, the running example from Section 4.1 will be used to examine the modeling methods from Chapter 4, and evaluated the accuracy and speed-up of the abstract models. Moreover, a

Table 7.1: Overview of the examined methods in Section 7.1

		Lowpass filt. Section 7.1	Diode circuit Section 7.2	GmC filter Section 7.3	Single track Section 7.4.1	PI-controller Section 7.4.2
Methods	Region ident.[1]	Table 7.2	*Gdist*	*Gdist*	-	DBSCAN[9]
	System descr.[2]		*mean*	*mean*	*mean*	*mean*
	S_λ modification[3]		*lsq*, S_λ *re.*	*lsq*, *reach*	-	-
	Invariant type[4]		polytope	polytope	polytope	interval
	Guard type[5]		polytope	halfspace	halfspace	polytope
	Guard ident.[5]		*intersectionI*	*distance*	*distance*	*distance*
	Object transf.[6]		-	-	-	-
	Jump function[7]		reset ($\boldsymbol{v}_r$)	shift ($\boldsymbol{v}_s$)	-	reset ($\boldsymbol{v}_r$)
	Model method[8]		-	*grdV*	-	-
Netlist	Dynamic order[10]	14	1	26	2	53
	Complexity	17 trans., 24 variab.	0 trans., 4 variab.	46 trans., 38 variab.	0 trans., 2 variab.	68 trans., 63 variab.
HA	Reduced order[11]	2 / 4	1	2	2	2
	Locations	3 / 5 / 7	2	3	1	9
	Output lang.	All	Matlab	Verilog-A	Matlab	Matlab

[1] Section 4.4.2 [2] Section 4.5.2 [3] Section 4.5.5 [4] Section 4.5.6 [5] Section 4.5.7 [6] Section 4.5.8
[7] Section 4.5.9 [8] Section 4.6 [9] Section 4.4.2 [10] r in Eq. (3.19) [11] m in Eq. (3.19)

4^{th} order abstract HA is generated in this section that demonstrates how the compositional system from Section 5.3 (see Fig. 5.18) can be abstracted as a whole. In Section 7.4.3, a compositional system is build that abstracts a control loop consisting of a PI-controller (Section 7.4.2) and a single track model (Section 7.4.1). While in Section 7.3, a complex industrial OTA-based GmC filter is abstracted.

7.1 Running Example: Second Order Lowpass Filter

According to Chapter 4, several methodologies can be used to model the HAs. In the following these methodologies are analyzed by abstracting the running example from Section 4.1. Table 7.2 shows an overview of the examined and used methods in this section. Note that, X marks the section where the corresponding method is examined. On top on the examination of the methodologies,

Table 7.2: Overview of the examined methods in Section 7.1

		Section 7.1.1	Section 7.1.2	Section 7.1.3	Section 7.1.4	Section 7.1.5	Section 7.1.6
Methods	Region ident.[1]	*Gdist*	*Gdist*	*Gdist*	*Gdist*	*Gdist*	*Gdist*
	System descr.[2]	param.[9]	*mean*	X	*mean*	*mean*	*mean*
	S_λ modification[3]	*lsq*, S_λ *re.*	X	*lsq*, S_λ *re.*	*lsq*	*lsq*	*lsq*
	Invariant type[4]	interval	interval	interval	polytope	polytope	polytope
	Guard type[5]	interval	halfspace	halfspace	X	halfspace	halfspace
	Guard ident.[5]	*distance*	*distance*	*distance*	X	*distance*	*distance*
	Object transf.[6]	*halfT*	*halfT*	*halfT*	-	-	-
	Jump function[7]	reset	reset	reset	reset	X	shift
	Model method[8]	-	-	-	-	X	*grdV*
HA	Reduced order	2	2	2	2	2	2
	Locations	3	3	3	3	3 / 5 / 7	3
	Output lang.	Matlab	Matlab	Matlab	Matlab	Verilog-A	SysC-AMS

[1] Section 4.4.2 [2] Section 4.5.2 [3] Section 4.5.5 [4] Section 4.5.6 [5] Section 4.5.7 [6] Section 4.5.8
[7] Section 4.5.9 [8] Section 4.6 [9] Section 5.1.1

a fourth order model in generated in Section 7.1.7, do demonstrate how a compositional system (Section 5.3) can be abstracted as a whole.

But before getting into the effect of the modeling methodologies, first the influence of the sampling performed by *Vera* (Chapter 3) on the model generation is examined in Section 7.1.1.

7.1.1 Influence of the Sampled State Space From Vera

In accordance with Section 4.5, depending on the sampling settings specified in *Vera*, the identified locations in the $\mathcal{S}_\lambda$ space can overlap strongly or slightly (or not at all) as illustrated in Fig. 7.1. This comes at a trade-off for the accuracy of the calculated sampled points, the transformation matrices, and the sampling time needed. Note that the sampling method has not been handled in Chapter 3 for simplicity. This method basically influences the transformation of the DC points from the $\mathcal{S}_o$ space to the $\mathcal{S}_\lambda$ space.

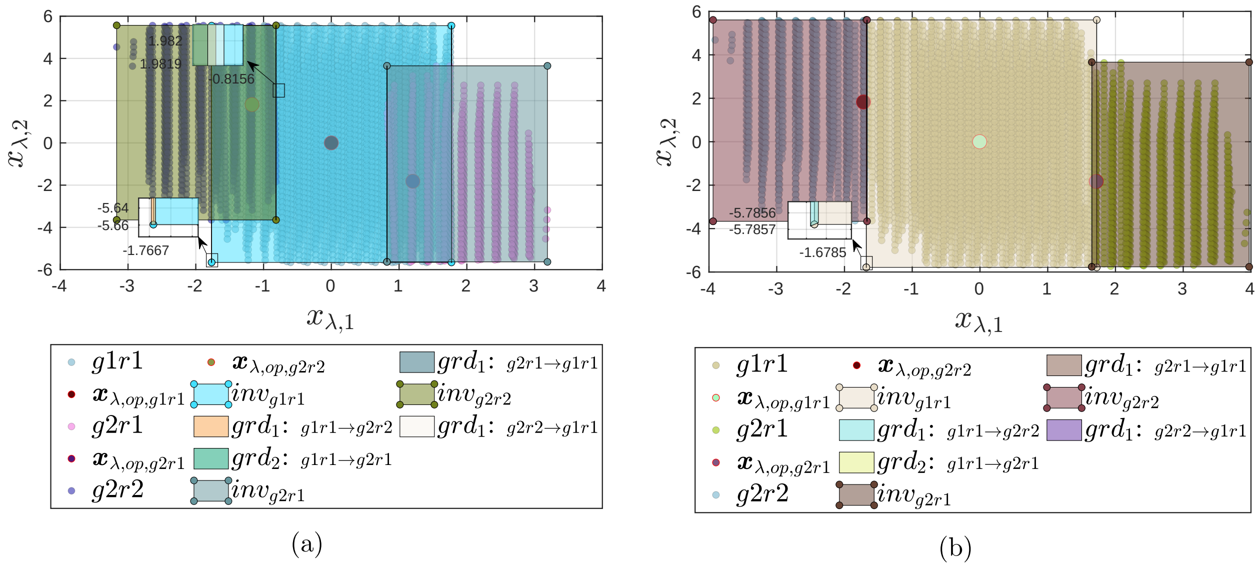


Fig. 7.1. Result of the abstraction performed by *Elsa* before applying the jump functions. In (a) the locations in the $\mathcal{S}_\lambda$ space overlap strongly, while in (b) the overlapping is minimal.

Two models were generated to examine the effect of the sampling method. For the first model shown in Fig. 7.1a, 18513 have been sampled by *Vera* in 126.67 s, while 18444 points have been sampled in 130,11 s for the model in Fig. 7.1b. To compare the generated models from these sampled data, the same options are used for the model generation process from Chapter 4. More precisely, both models have invariants and guards modeled as interval hulls. The guards were found using the *distance method*. These guards have narrow thicknesses as shown in Fig. 7.1. For both models, the guards and invariants were calculated in the S_{virt} space and transformed to the $\mathcal{S}_\lambda$ space using the *halfT method*. Moreover, the *lsq filter* was used and the $\mathcal{S}_\lambda$ space was recalculated using the *transPt method*. As both systems were modeled with jump functions for *Cora*, the locations are adjusted similarly to Fig. 4.38, by subtracting the operating points from the geometric shapes (see Section 4.5.9). The system descriptions were modeled by the approach from Section 5.1.1 with matrix zonotopes, thereby capturing (over approximating) the whole system behavior, and omitting the abstraction errors performed from using single representing values (Section 4.5.2). Note that the model from Fig. 7.1a actually represents the same model used in Section 5.1.1.

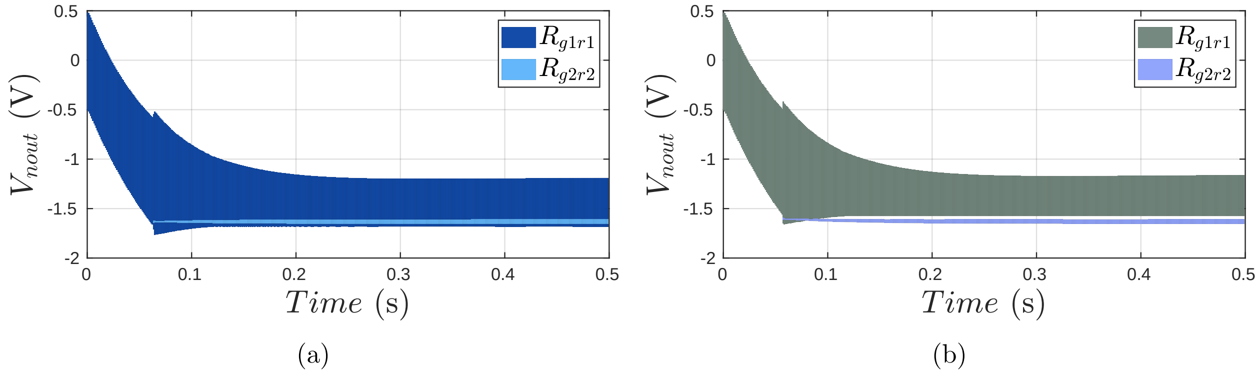


Fig. 7.2. Results of the reachability analysis performed on (a) the model shown in Fig. 7.1a and (b) the model from Fig. 7.1b.

Using *Cora*, a reachability analysis was executed with an range for the initial value of the state variables $\boldsymbol{x}_{\lambda,i} = [-0.5, 0.5]$ and an input voltage $V_{nin} = [2.5, 3.5]$ V. The results are illustrated in Fig. 7.2. As observed, even though both models have (in this case) exactly the system behaviors ($\boldsymbol{A}_{loc}$ and $\boldsymbol{B}_{loc}$), as well as the same transformation matrices ($\boldsymbol{F}_{loc}$ and $\boldsymbol{L}_{loc}$), the results vary. This lies in the fact that the guards and invariants identified vary, which can be traced back to the sampled points as observed in Fig. 7.1. Moreover, the operating points differ as well, influencing the jumps performed.

7.1.2 Impact of the $\mathcal{S}_\lambda$ Space Manipulations

To compare the possible $\mathcal{S}_\lambda$ space modifications from Section 4.5.5, six models were created. All models start with a state space similar to the one shown in Fig. 4.20c, and were created with invariants modeled as intervals, and guards modeled as halfspaces using the *distance method* in the $\mathcal{S}_{virt}$ space. Moreover, all models use jump functions. Models A, B, and C use the *lsq filter*, while models D, E, and F don't. Models A and D use the *transPt method* to recalculate the $\mathcal{S}_\lambda$ space, while models B and E use the *transLoc method*. Models C and F skip this calculation. Table 7.3 summarizes the $\mathcal{S}_\lambda$ space modifications of the models. Note that, as mentioned is Section 4.5.2, the *lsq filter* uses by default the *transPt method* as the *transDC method* is not handled here.

Table 7.3: Comparison of the $\mathcal{S}_\lambda$ space manipulations of the generated models

methods	model A	model B	model C	model D	model E	model F
lsq filter	✓	✓	✓	-	-	-
$\mathcal{S}_\lambda$ *re.*	*transPt*	*transLoc*	-	*transPt*	*transLoc*	-

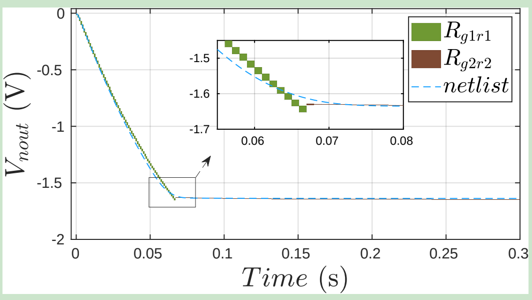
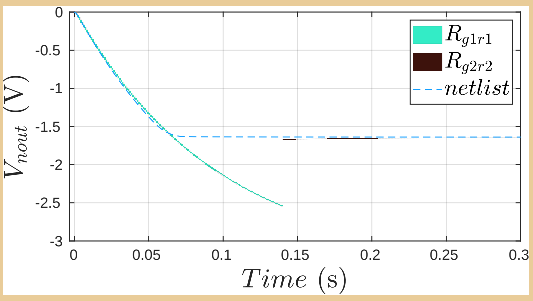
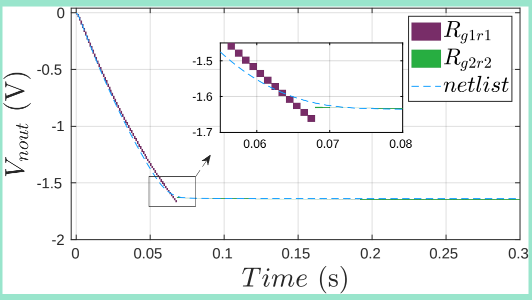
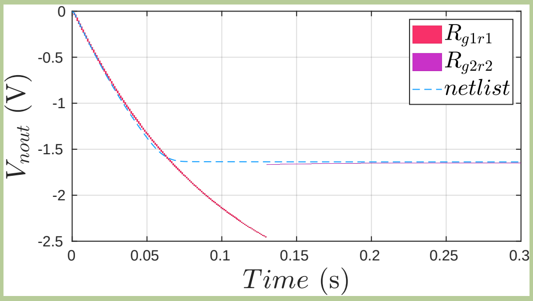
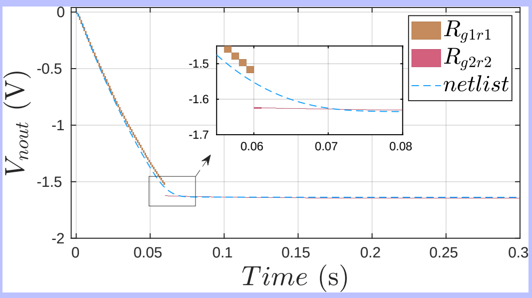
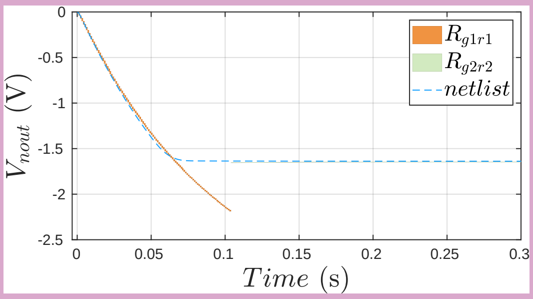
The $\mathcal{S}_\lambda$ space of models B, C, E, and F are illustrated in Fig. 4.21d, Fig. 4.21c, Fig. 4.20d, and Fig. 4.20c, respectively. The $\mathcal{S}_\lambda$ space for model D can be extracted from Fig. 4.16b when removing the bad points, while the $\mathcal{S}_\lambda$ space for model A is shown in Fig. 4.16b. Note that, all models were generated with the sampling method that yielded the $\mathcal{S}_\lambda$ space in Fig. 7.1a. The results of the reachability analysis performed on the models with $V_{nin} = 4$ V are shown in Table. 7.4.

The first column of Fig. 7.4 shows the results of the reachability analysis conducted with models generated with the application of the *lsq filter* (Section 4.5.2), while the second column of this figure shows the results for models generated without the *lsq filter*. As observed, this filter is usually necessary to obtain good results, especially when dealing with a overlapping locations in the reduced state space $\mathcal{S}_\lambda$. On top of this, recalculating the $\mathcal{S}_\lambda$ space usually yields better results as observed when comparing the results of models A and B with model C. Moreover, a closer look at models A and B shows that the discontinuity in the V_{nout} upon switching locations decreases in case the *transPt method* is used. Comparing models A and C, on the other hand, the best results are obtained if the *lsq filter* is used and the $\mathcal{S}_\lambda$ is recalculated using the *transPt method* as mentioned at the end of Section 4.5.5.

7.1.3 Effect of the System Description

To demonstrate the effect of the system description method from Section 4.5.2, four models were generated. The first model was created with the *op method*, the second with the *mean method*, the third with the *dc method*, while the forth was created with the *weight method* with $w_1 = 0.5$,

Table 7.4: Result of the reachability analysis performed on various models from Table 7.3.

	lsq filter	no lsq filter
transPt method	 model A	 model D
transLoc method	 model B	 model E
unchanged S_λ space	 model C	 model F

$w_2 = 0.3$, and $w_3 = 0.2$. The remaining settings are listed in Table 7.2, which are basically identical with the methods used to generate model A from Section 7.1.2. A reachability analysis was performed with an input $V_{nin} = 4$ V. The results are illustrated in Fig. 7.3. As observed in Fig. 7.3, the closest results to the Spice netlist are obtained when using the *mean method*. Moreover, the remaining methods produce large discontinuities. Both aspects can be traced back mainly to the transformation matrices, as the *mean method* considers all points belonging to a location equally, thereby modeling these matrices by the sampled values of all points and not a single representing one. Therefore, the *mean method* is used as the standard system description method.

As stated in Section 4.5.2, the *lsq filter* can also be applied on the system description, thereby removing flagged points from the calculation of the matrices. However, the results are not strongly affected as shown in Fig. 7.4, during the same reachability analysis with $V_{nin} = 4$ V performed on two models whose system description is modeled with the *mean method*.

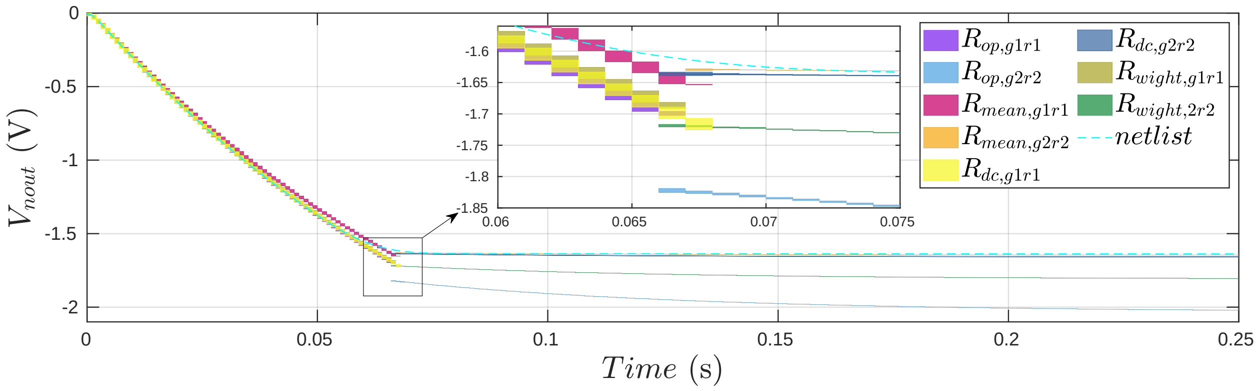


Fig. 7.3. Results of the reachability analysis performed on four models generated with different determined system descriptions compared to the simulation of original Spice netlist.

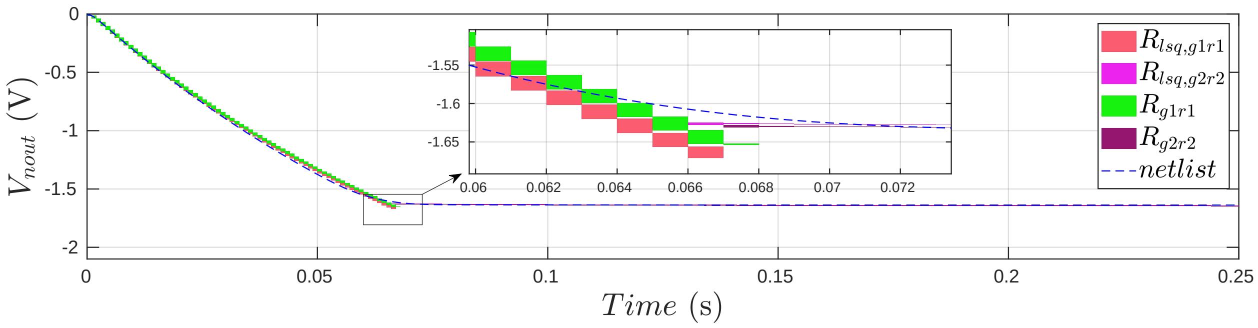


Fig. 7.4. Results of the reachability analysis performed on two abstract models with (subscript lsq) and without applying the *lsq filter* in the calculation of the system description.

7.1.4 Guards and Invariants

There exist according to Section 4.5.6 various forms to represent the invariants. Depending on the type chosen, the guard calculations are influenced as described in Section 4.5.7. Moreover, using the *intersection method* from Section 4.5.7 to find the guards, the size (volume) of the guards can be influenced.

To exemplify these points, three HAs in *Cora* syntax were generated, from the same $\mathcal{S}_\lambda$ space. Three minimal overlapping locations were identified for each HA (Fig. 7.1b). For all three models the invariants were specified as polytopes. Two models were generated with guards specified as polytopes and identified via the *intersectionI method*. While the *limit* (line 12 of Algorithm 7) for both models was specified as the maximum number of iterations performed after the first intersection, this number was set to 1 for the first and to 10 for the second model. The HAs in the $\mathcal{S}_\lambda$ space, before adjusting the location by the jump functions, are shown in Fig. 7.5a and Fig. 7.5b, respectively. The guards of the third model were identified by the *distance method* and model as polytopes as shown in Fig. 7.5c. Note how the number of maximum iterations influences the thickness of the identified guards. Moreover, the edges of the guards are analogously to the edges of the invariants.

A reachability analysis was performed with the same input step function as previously (V_{nin} = 4 V) on all three models. The results at the output node are illustrated in Fig. 7.6 for the two

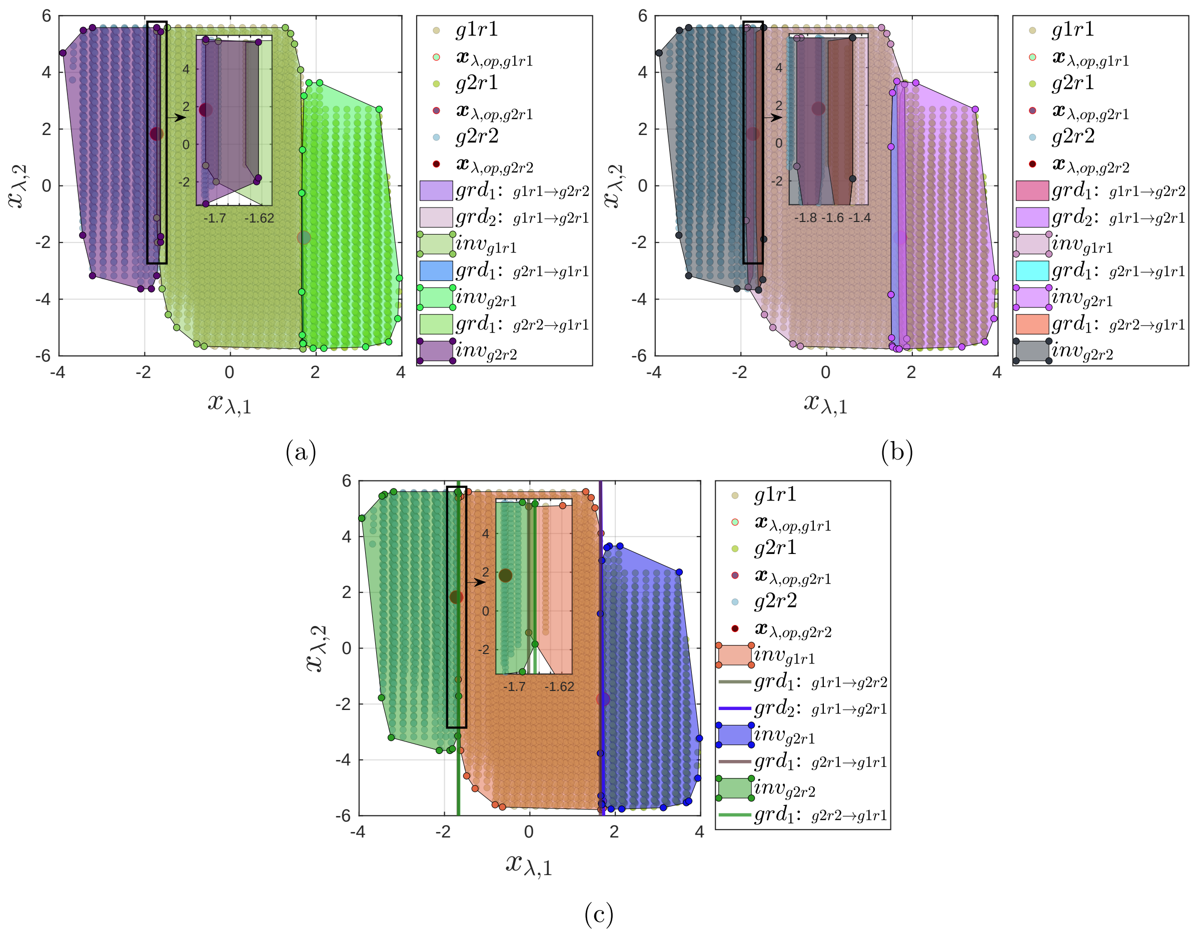


Fig. 7.5. Guards and invariants of the HAs in the $\mathcal{S}_\lambda$ space before adjustment. The polytopic guards are found using the *intersectionI method* with the maximum number of iterations set to (a) 1 and (b) 10. In (c) the halfspace guards are found using the *distance method*.

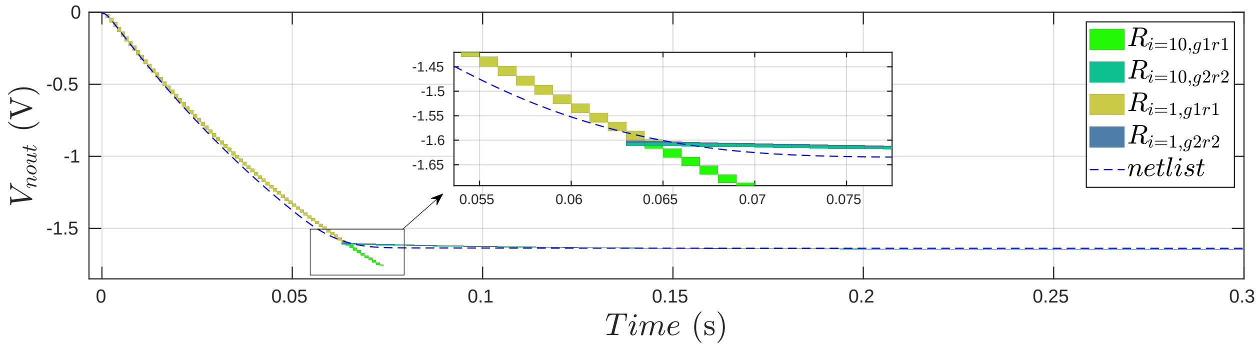


Fig. 7.6. Result of the reachability analysis performed on two abstract models using 1 and 10 maximum iterations for the guard construction using the *intersectionI method*. The subindex i on the reachable sets indicates the number of maximum iterations.

models generated with the *intersectionI method*. As observed, the size of the guards influences the results of the reachability analysis. For a larger number of iterations, the invariants are enlarged more by the *intersectionI method*, yielding guards that are further away from the original boarders of the invariants described by the sampled points. Hence, the location switch happens at a further instance. As observed in Fig. 7.6, with a maximum number of iterations set to 10, the HA stays in $g1r1$ too long resulting in discontinuities during the transition of the location to $g2r2$. Moreover, after the transition, the reachable set of the model with 10 iterations is larger than the reachable set obtained by the model with 1 iterations. This basically lies in the nature *Cora* computes the reachability analysis for the guard *intersection method* specified as *polytope* [ASB10b]. Note that, for halfspace guards the method in [AK12a] is used here for the analysis in *Cora*.

In Fig. 7.7, the results obtained with two models described with different guard identification methods are illustrated. While one models uses halfspace guards identified via the *distance method*

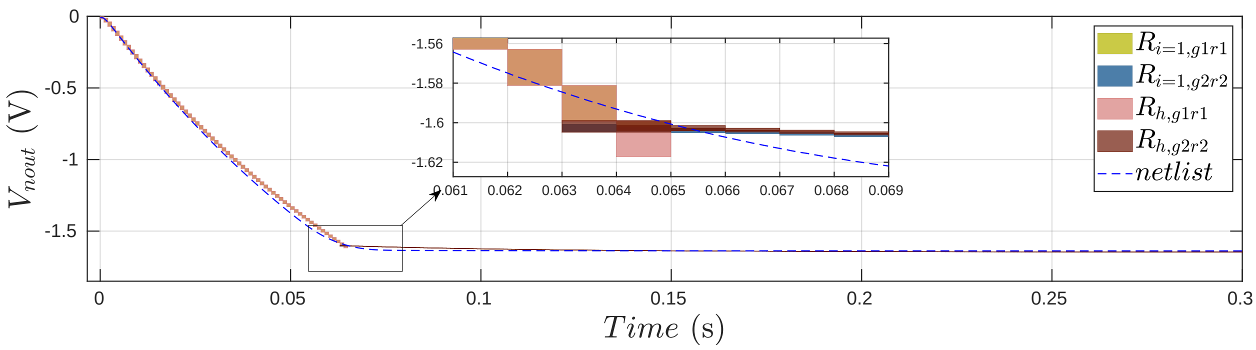


Fig. 7.7. Result of the reachability analysis performed on a HA with halfspace guards identified with the *distance method* (subscript h) and on a HA with polytopic guards identified with the *intersectionI method* (subscript i).

(Fig. 7.5c), the other model uses polytopic guards identified via the *intersectionI method* with a maximum number of iterations set to 1 (Fig. 7.5a). As observed, the difference is barely visible. This lies in the fact that the polytopic guards obtained have a very narrow width, and are almost on the boarders of the unmodified invariants, where the halfspace guards lie. With an increasing number of iterations the results obtained with the *intersectionI method* start to vary significantly compared to results obtained with the *distance method*. Note that the *distance method* with halfspace guards usually yield the most accurate results.

The previous results were obtained with jump functions. Specifically, since the models were created for *Cora*, the jump functions are described by Eq. (4.33), and the geometric objects (guards and invariants) in the $\mathcal{S}_\lambda$ space were adjusted similarly to Fig. 4.38. If the modeled with guards identified via the *intersectionI method* with the maximum iterations set to 1 (Fig. 7.5a) was created without jump functions, the result of the reachability analysis with the same input signal contains large discontinuities as illustrated in Fig. 7.8. The corresponding results shown in the $\mathcal{S}_\lambda$ space drawn versus time are illustrated in Fig. 7.9a. As observed Fig. 7.9a, the $\boldsymbol{x}_\lambda$ variables remain unchanged, leading to bad results as shown in Fig. 7.8. However, in case the system is modeled with jump functions, the $\boldsymbol{x}_\lambda$ undergo a reset upon switching the location according to Eq. (4.33) (Fig. 7.9b), yielding better results as illustrated in Fig. 7.6. In disregard of the guard identification method used, similarly results are obtained. For the Verilog-A and SystemC-AMS models, the jump functions are described by shift vectors, as described in Section 4.6.2, which is constantly subtracted from the system behavior. This aspect will be emphasized in the following section.

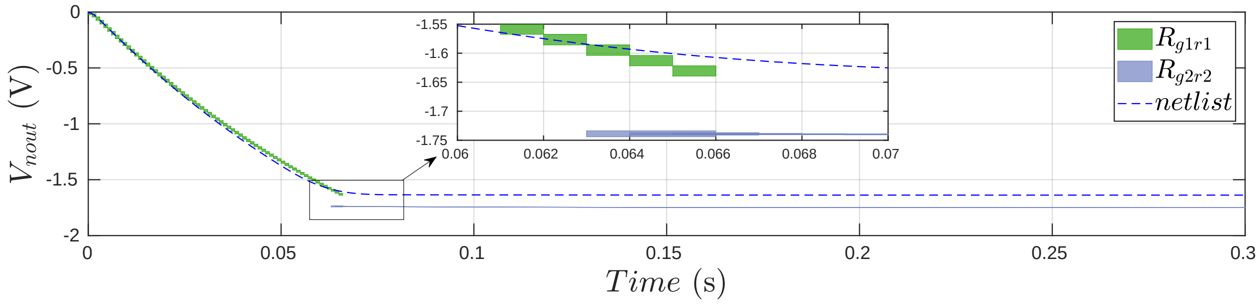


Fig. 7.8. Result of the reachability analysis performed on the HA with polytopic guards identified with the *intersectionI method* ($i = 1$) with no jump functions.

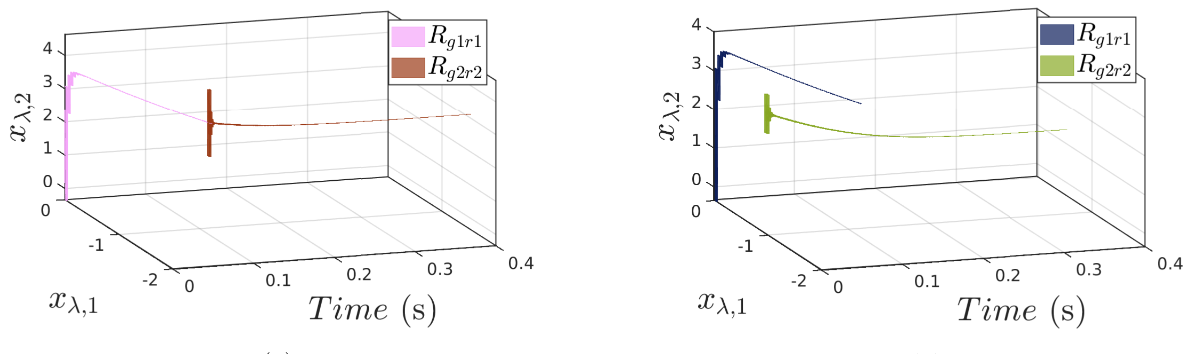


Fig. 7.9. Result of the reachability analysis illustrated in the S_λ space. The results are shown for a HA modeled (a) without jump functions and (b) with jump functions.

7.1.5 Verilog-A Models and the Jump Functions

In this section, the presence and absence of the jump functions are examined based on Verilog-A generated abstract models. Moreover, the impact of using *grdV* and *invV method* from Section 4.6.2 are examined. For this, 4 models were generated. All models are based on the state space presented in Fig. 7.10. For all models, the invariants are modeled as intervals while the guards, identified via

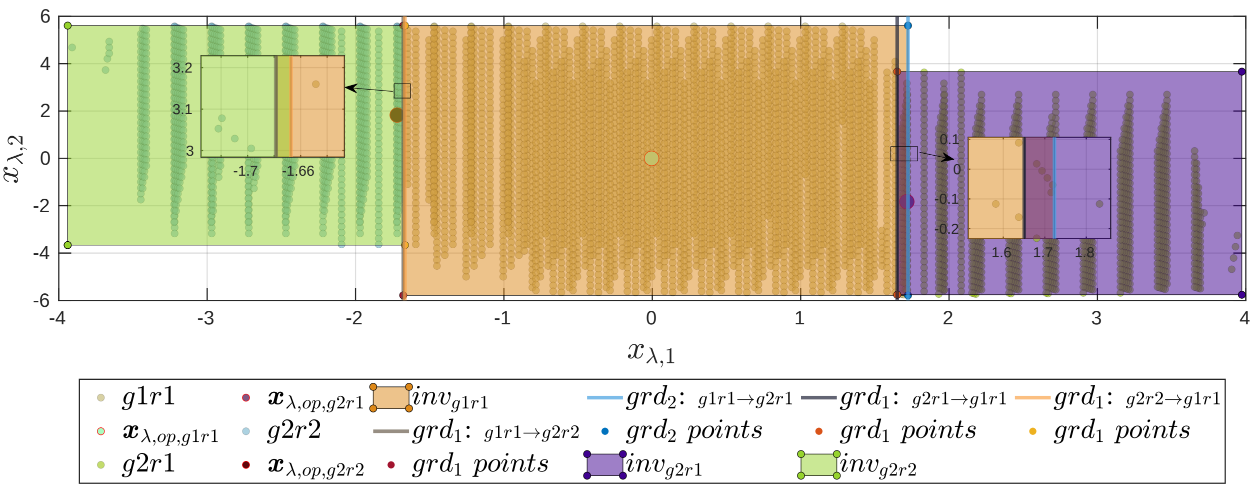


Fig. 7.10. Result of the behavioral abstraction performed by *Elsa* in the S_λ space.

the *distance method* in the $\mathcal{S}_\lambda$ space, are modeled as halfspaces.

Four models have been created: two with the *grdV method* and two with the *invV method*. In each case, one model was created with and one without jump functions according to Section 4.6.2. Fig. 7.11 shows the results of the simulations performed with a sine wave of amplitude 4 V and $freq$ of 1 Hz at the input V_{nin}. As observed, even with a well sampled $\mathcal{S}_\lambda$ space, the results are better if the systems are modeled with jump functions. Moreover, the *grdV method* usually yields better

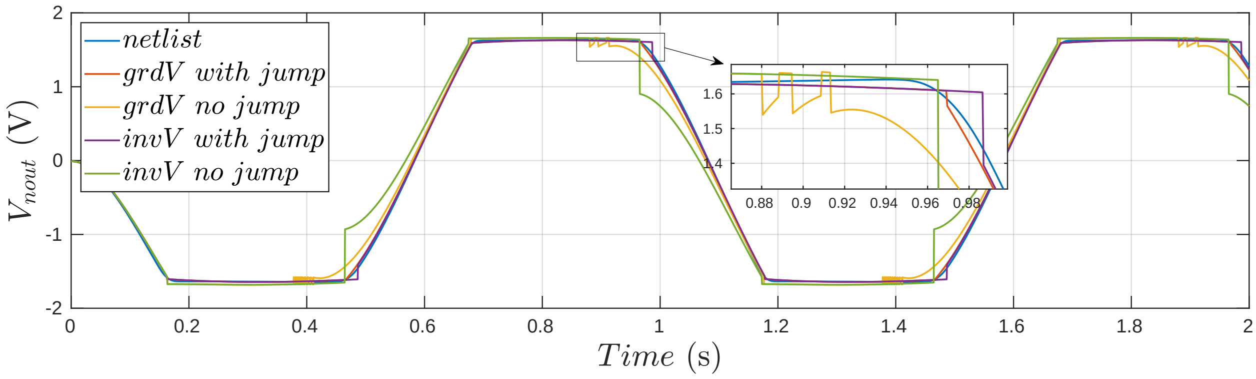


Fig. 7.11. Result of the simulations performed on the four generated HAs in Verilog-A for an input $V_{nin} = 4 \cdot sin(2\pi t)$ V.

results than the *invV method*. This can be further analyzed in Fig. 7.12, which shows the first $\boldsymbol{x}_\lambda$ variable and the corresponding element from the shift vector $\boldsymbol{v}_s$. According to Section 4.6.2, the current location of the system is identified by the $\boldsymbol{x}_\lambda$ variables, guards or invariants, and optionally the jump functions. As observed in Fig. 7.12, the HAs modeled without jump functions are unable

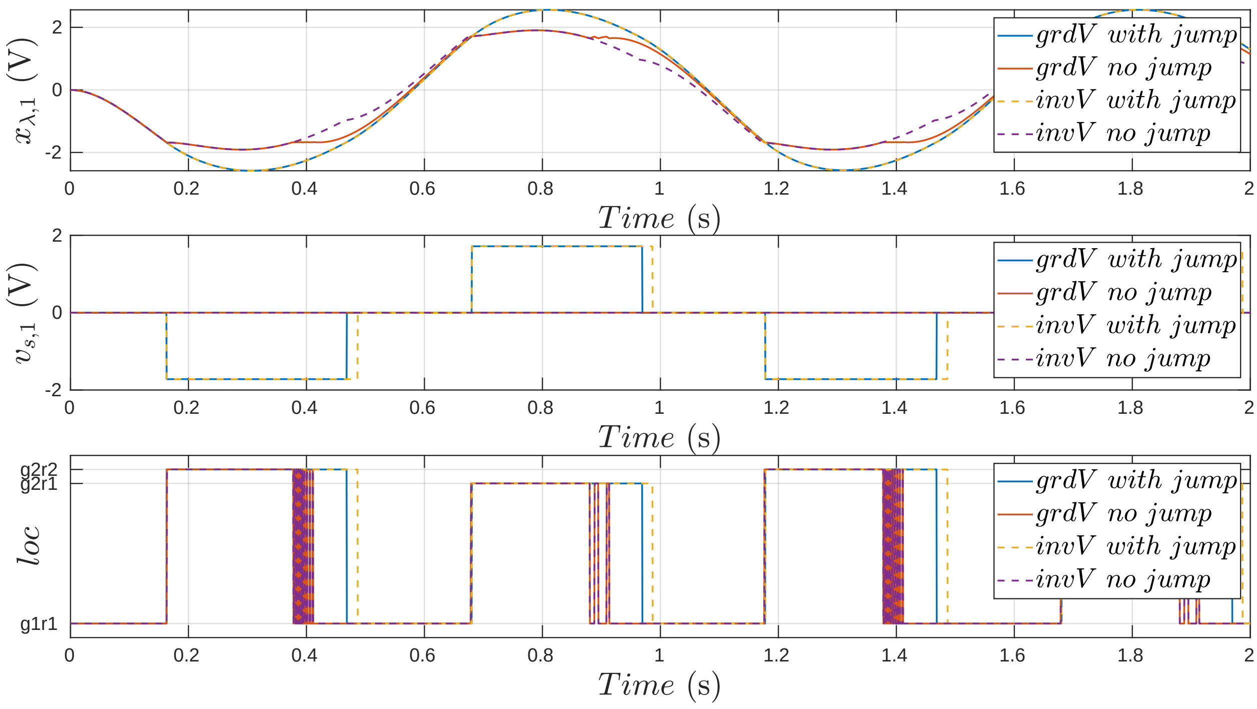


Fig. 7.12. Result of the previous simulations of four generated HAs illustrated upon $\boldsymbol{x}_{\lambda,1}$ and the corresponding element from the shift vector $\boldsymbol{v}_{s,1}$.

to determine the current location correctly, leading to the fluctuation of the location variable *loc*. This in terms result in inaccurate results as observed in Fig. 7.11. On the other hand, according to Algorithm 10, the *invV method* decides based on the operating points the current location the target location once the HA leaves the current invariant. As the location variable is contained in the conditions at lines 3, 5, and 7 of Algorithm 10, the HA is forced to stay in a location until the current invariant becomes invalid. On one hand, this helps the Verilog-A simulator to converge rapidly, on the other hand, the accuracy of the HA is diminished. As the results show in Fig. 7.11, in case the guards are used to find the current location (*grdV method*), the most accurate results are obtained. Thus, this method is adapted as the standard method in the remaining examples. Note that, as observed in Fig. 7.12, in contrast to the Matlab models, the $\boldsymbol{x}_\lambda$ variables are only shifted by a shift vector. Specifically, the value of $\boldsymbol{x}_\lambda$ is not modified by a reset.

In Table 7.5 a comparison is presented of the results obtained for several simulations performed on the original Spice netlist and an abstract HA. The HA was created using the *grdV method* (Algorithm 8) with jump functions and 3 locations. Three simulation were performed for a time

Table 7.5: Comparison of the abstracted Verilog-A model and the original Spice netlist

Circuit	Step 1 V				Step 4 V				Sine 4 V at 1 Hz			
	Time	Iter.	$\hat{\delta}_y$	$\hat{\delta}_{y,r}$	Time	Iter.	$\hat{\delta}_y$	$\hat{\delta}_{y,r}$	Time	Iter.	$\hat{\delta}_y$	$\hat{\delta}_{y,r}$
HA	0.15	31761	0.047	2.8	0.15	31765	0.047	2.8	0.21	40075	0.058	3.5
Netlist	4.54	42078	-	-	5.19	49743	-	-	5.72	50325	-	-

$\hat{\delta}$ stands for the maximum value of δ. Time is in s, $\hat{\delta}_y$ in V, and $\hat{\delta}_{y,r}$ in %

window of 1 s with a fixed time step of 0.1 μs. The first simulation was carried out with a step function at V_{nin} with a rise time of 20 μs and an amplitude of 1 V. The second simulation was performed similarly, but with an amplitude of 4 V. The third simulation was performed with a sine wave of amplitude 4 V and $freq$ of 1 Hz. The run time, the number of Newton iterations, and the maximum and maximum relative output errors, $max(\delta_y)$ and $max(\delta_{y,r})$, are used to compare the simulations. As the system has only one output, the calculation of the errors is straight forward. The maximum output error ($max(\delta_y)$), which is computed by finding the maximum difference between the output voltages of the HA and the netlist, is very similar in all simulation as observed in Table 7.5. Moreover, this is as well true for the maximum relative output error ($max(\delta_{y,r})$). Note that this value is normalized over the range of output voltage, which is in this case roughly $max(|y_{cons,netlist}|) + max(|y_{cons,HA}|) = 1.65 + 1.65 = 3.3$ V for all three simulations.

The corresponding simulation figures from Table 7.5 are presented in Fig. 7.13. The vertical purple lines in Fig. 7.13 indicate when the HA changes the locations. For each performed simulation, the output error δ_y as well as the output voltage V_{nout} are illustrated versus time. In general, the output error increases the further the system moves from the operating point, to reach its largest values at the borders of the locations.

The HA model used here is in fact the same one used for EC in Section 6.1, for which *Vera* calculated the maximum output error $max(\delta_y) = 0.0668$ V and the maximum relative output error $max(\delta_{y,r}) = 3.95$ %. What is interesting to notice is that the errors from Table 7.5 are bounded by these values. Moreover, in all cases the Spice netlist performs more iterations compared to the HA, and the abstracted HAs yield a maximum speed-up of 27.2.

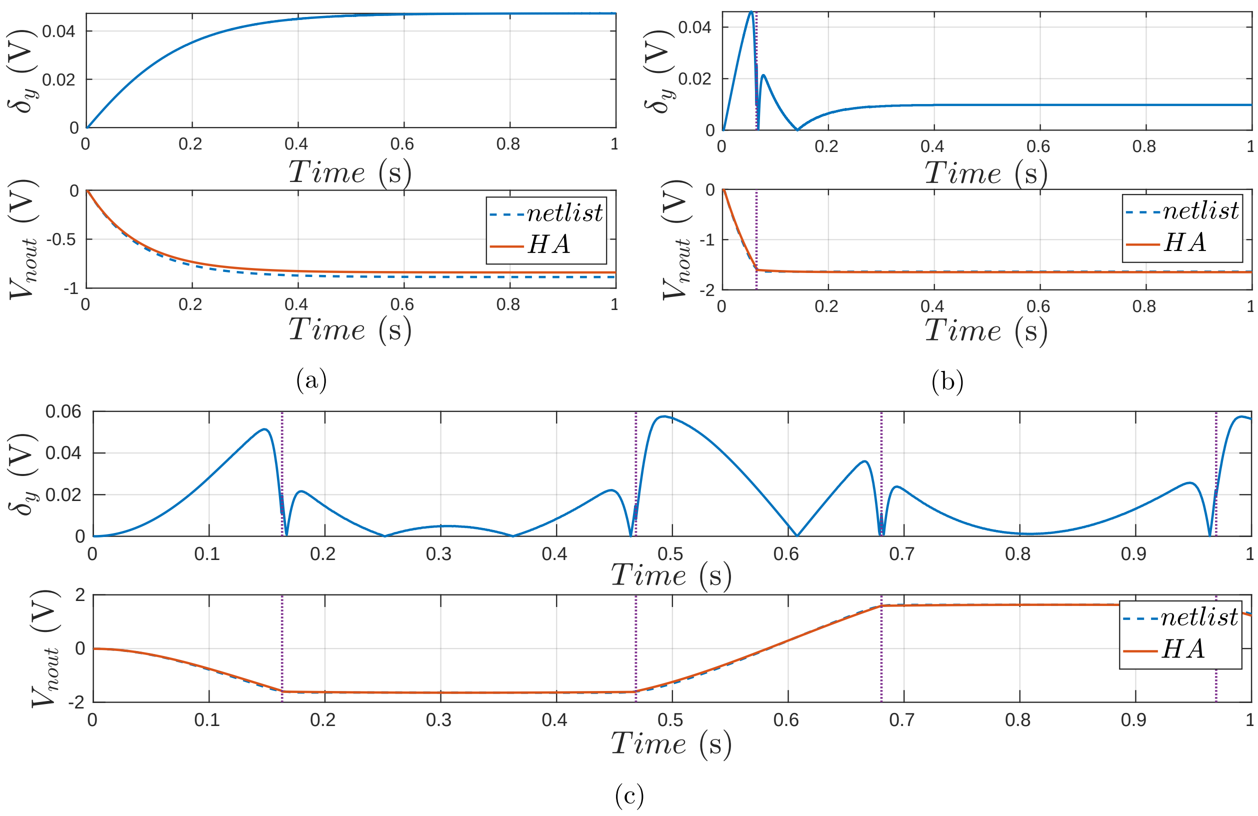


Fig. 7.13. Simulation results obtained with a step function of amplitude (a) $V_{nin} = 1$ V, (b) $V_{nin} = 4$ V, and (c) a sine wave of amplitude $V_{nin} = 4$ V and $freq = 1$ Hz on a Verilog-A HA modeled with the *grdV method* with jump functions and three locations. The vertical purple lines indicate when the current location is changed.

As stated in Section 6.1, increasing the number of locations usually yields better results and decreases the errors. To emphasize this aspect, three HAs were generated with 3, 5, and 7 locations, respectively. All HAs were model with the *grdV method* and with jump functions. A simulation is performed on these models as well as on the Spice netlist with a sine wave $V_{nin} = 4 \cdot sin(2\pi t)$ V for 2 s. The results are shown in the first row in Fig. 7.14 at the output of the system V_{nout}. The simulation are labeled according to the number of locations of the corresponding HA. The second row in Fig. 7.14 shows the first dimension of the shift vector $\boldsymbol{v}_s$ from Eq. (4.34). Whenever this value changes, the corresponding HA switches the current location. Note that, as illustrated in the first row of Fig. 7.12, the $\boldsymbol{x}_\lambda$ variables are only shifted by the shift vector $\boldsymbol{v}_s$ for the Verilog-A models, in contrast to the *Cora* models.

As observed in the first row of Fig. 7.14, a HA with more locations can transition smoother between the locations than a HA with fewer locations. This can be as well observed at the voltage at the negative terminal of the operation amplifier V_{neg} (see Fig. 4.2) presented in Fig. 7.15. In Fig. 7.16 the output error of the three generated HAs is illustrated for the same input voltage $V_{nin} = 4 \cdot sin(2\pi t)$ V. Specifically, in the first three rows of Fig. 7.16 the output errors $\delta_{y,3}$, $\delta_{y,5}$, and $\delta_{y,7}$ are illustrated which correspond to the HA generated with 3, 5, and 7 locations, respectively. In the last row of the figure, the three errors are illustrated against each other. As observed, the accumulated output error in general decreases with an increasing number of locations. However, as

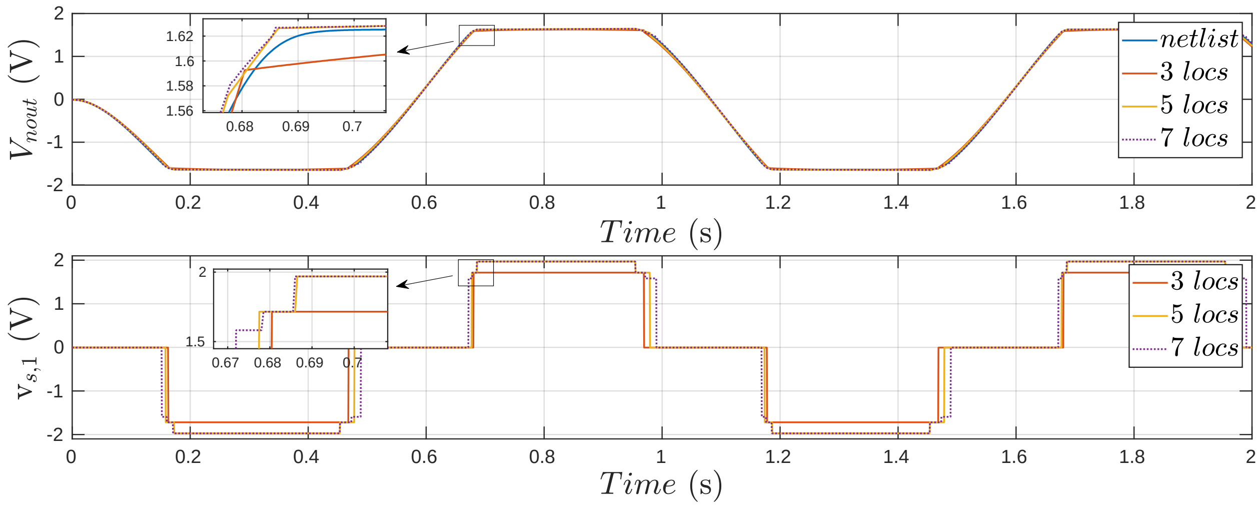


Fig. 7.14. Result of a simulation with $V_{nin} = 4 \cdot sin(2\pi t)$ V performed on three HAs with 3, 5, and 7 locations and on the Spice netlist. The first row shows the voltage at the output terminal, while the second row shows the first dimension of the shift vector $\boldsymbol{v}_s$.

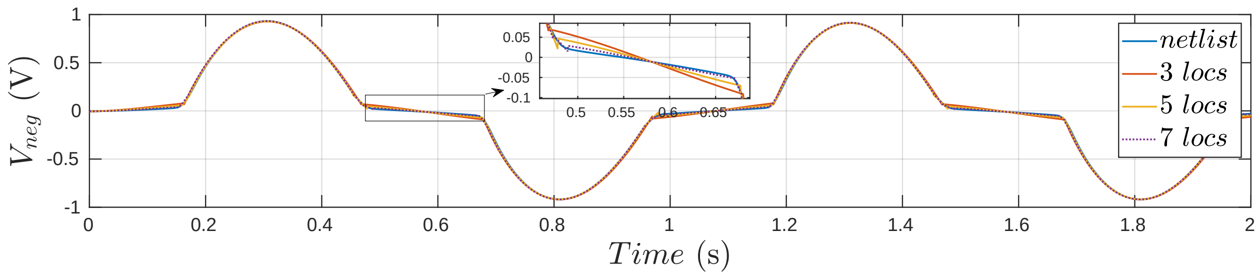


Fig. 7.15. Result of a simulation with $V_{nin} = 4 \cdot sin(2\pi t)$ V performed on three Verilog-A HAs with 3, 5, and 7 locations and on the Spice netlist at the negative terminal of the operation amplifier V_{neg} (see Fig. 4.2).

the number of locations increases, the error during the transition of the locations becomes sharper. This can be traced back to the guards, and in particular to the *grdV method* used to describe the HAs. Using the *invV method*, the error during the location switch can be decreased, however, this comes at the expense of the accumulated error as examined in Section 7.3. Another import aspect that Fig. 7.16 shows, is that the maximum output error calculated by *Vera* in Section 6.1 is violated for the HA with 7 locations. In Section 6.1, the EC of the netlist against each of the three HAs yield $max(\delta_y)$ to be 0.066, 0.047, and 0.030 V for the HAs with 3, 5, and 7 locations, respectively. This limit is exceed by the HA with 7 locations. This can be traced back to the settings specified for EC in Section 6.1. The HAs were compared against the netlist with a constant specified minimal state space step set to $\Delta x_\lambda = 0.25$ (see Section 3.2.3), which is relatively large. Decreasing this state space step to 0.1 yields $max(\delta_y)$ to be 0.067, 0.06, and 0.046 V for the HA with 3, 5, and 7 locations, respectively. The corresponding result of the EC in the $\mathcal{S}_{virt}$ space of the HA with 7 locations is illustrated in Fig. 7.17. Compared to first row of Fig. 6.7, the obtained errors in Fig. 7.17 increased, as the EC was performed with a higher precision. Thus, in order to correctly verify the generated HAs and obtain valid error margins, the EC must be executed with a well-chosen state step. In Section 7.3, an approach is illustrated that extends a model with an affine

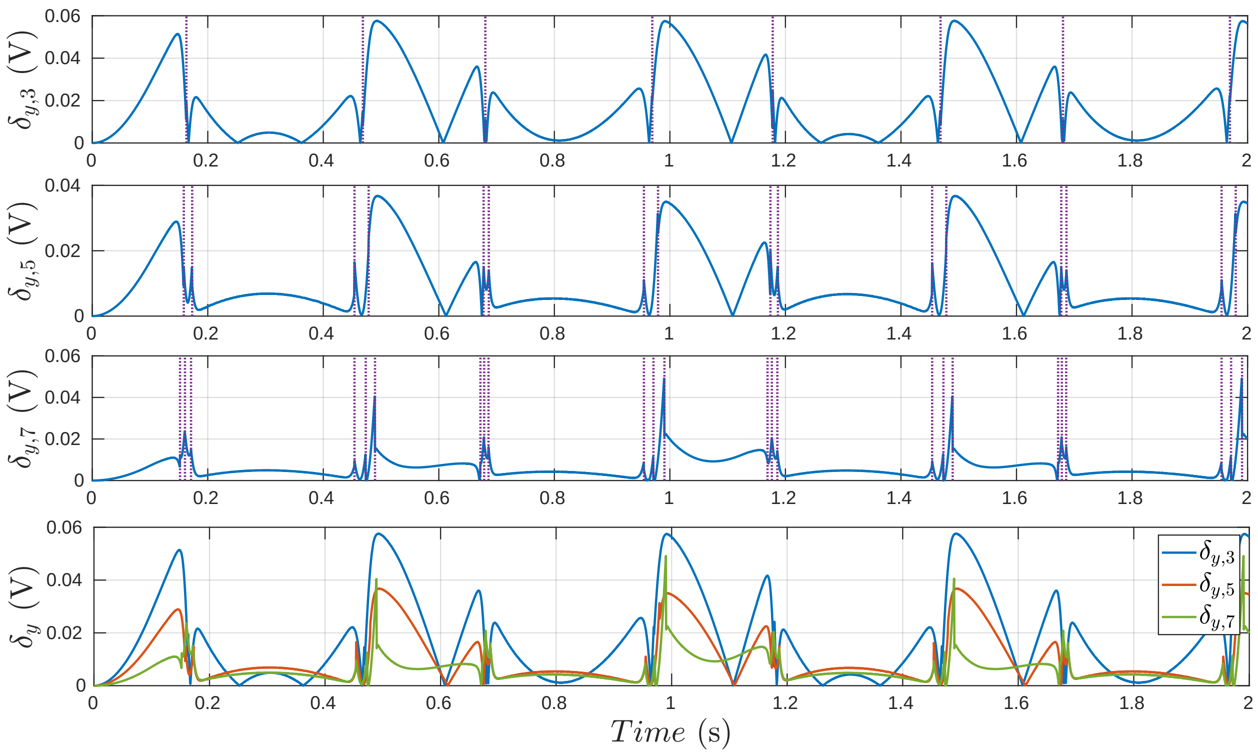


Fig. 7.16. In the first three rows, the output errors of the HAs generated with 3, 5, and 7 locations are shown, respectively. In the last row, these errors are plotted against each other.

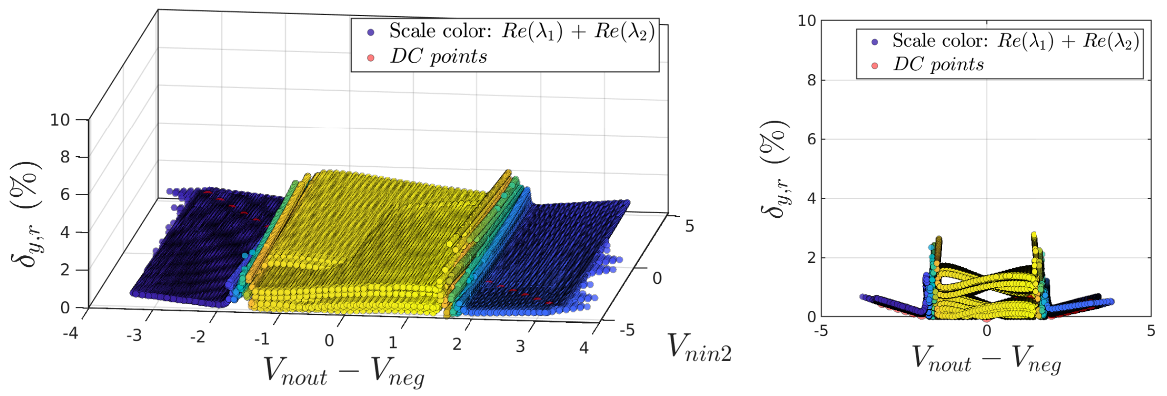


Fig. 7.17. EC of the Verilog-A generated HA with 7 locations versus the original Spice netlist with a minimal state step of 0.1 illustrated in the $\mathcal{S}_{virt}$ space.

description that integrates the state step into the modeling process.

7.1.6 SystemC-AMS Models

Previously, the Matlab as well as Verilog-A models were generated according to various options. In this section, the focus lies on the generation of SystemC-AMS models. As stated in Section 4.6, the final step in the model abstraction process is the creation of the HA in the desired syntax. Thus, till the model creation step, the model abstraction process is identical regardless of the desired file syntax. Consequently, the Verilog-A models generated in the Section 7.1.5 are similar to the models from this section if the same settings are used.

During the model creation, the models start to vary as for example the Verilog-A models describe the system behavior in the reduced state space of the system with the *ddt* operator, while the SystemC-AMS have 4 different methods (Section 4.6.3) to perform this task. These four system modeling methods are: *ssC method* (sca_ss), *eulC method* (backward Euler), *rukC method* (Runge-Kutta), and *disC method* (discretized state space). In the following, these methods are compared. For this purpose, four abstracted HAs have been generated that only vary in their method to describe the system behavior. The remaining settings are listed in Table 7.2. As in Section 7.1.5, these models were created with the *grdV method* (see Section 4.6.3), with three locations and jump functions. The models are compared based on simulations with a sine wave of amplitude 4 V and frequency of 1 Hz at the input V_{nin}. The simulations are performed with a simulation time step (sca_time) of 0.1 μs and a time window of 2 s. The results are presented in Table 7.6. The speed-up factor is given between the runtime of the generated models against the runtime of the original Spice netlist from Table 7.5. For all models, the circuit was sampled in 130.11 s and imported to Matlab's memory in 2.55 s. The modeling time in Table 7.6 represent the time consumed purely by *Elsa*. Note that to decrease the modeling time, a clustering method (k-means) can be used for the region identification (Section 4.4.2) instead of the *Gdist method*, while still yielding the same models for this example. If k-means is used for the region identification for example, the modeling time using the *disC method* is reduced from 12.79 s to 4.48 s.

Table 7.6: SystemC-AMS models of the running example

Modeling method	Modeling time (s)	Compile time (s)	Runtime (s)	Speed up	$max(\delta_y)$ (V)	$\max(\delta_{y,r})$ (%)
ssC method	12.62	1.56	0.177	32.3	0.0589	3.58
eulC method	12.76	1.58	0.141	40.5	0.0592	3.60
rukC method	12.81	1.56	0.150	38.1	0.0596	3.63
disC method	12.79	1.57	0.136	42.0	0.0586	3.56

Along with the test-bench, the model and the input signal are compiled into a single executable. The elapsed compiling time is presented in Table 7.6. This table holds as well for each HA the runtime of the generated executable, the maximum output error $max(\delta_y)$, and the maximum relative output error $max(\delta_{y,r})$. As observed, all these values seem to lie close to each other for the various models. However, as illustrated in Fig. 7.18, the *disC method* yields the best results.

As shown in Fig. 7.18, the output error (δ_y) of the model generated with the *disC method* exhibits less deviation than the models generated using the remaining methods. However, for this example, this model yields larger peaks during the transitions between the locations as illustrated in the second row of Fig. 7.18. In general, all four methods are suited for the creation of good models.

Compared to the Verilog-A model from Section 7.1.5, the simulation using the SystemC-AMS models consumed less time. The simulation for the sinus signal consumed 0.21 s for the Verilog-A model, which is larger than the simulation time of the SystemC-AMS models which ranges between 0.177 and 0.136 s, while maintaining nearly the same output errors. However, contrarily to the Verilog-A models which are compiled separated from their test-bench, the SystemC-AMS models

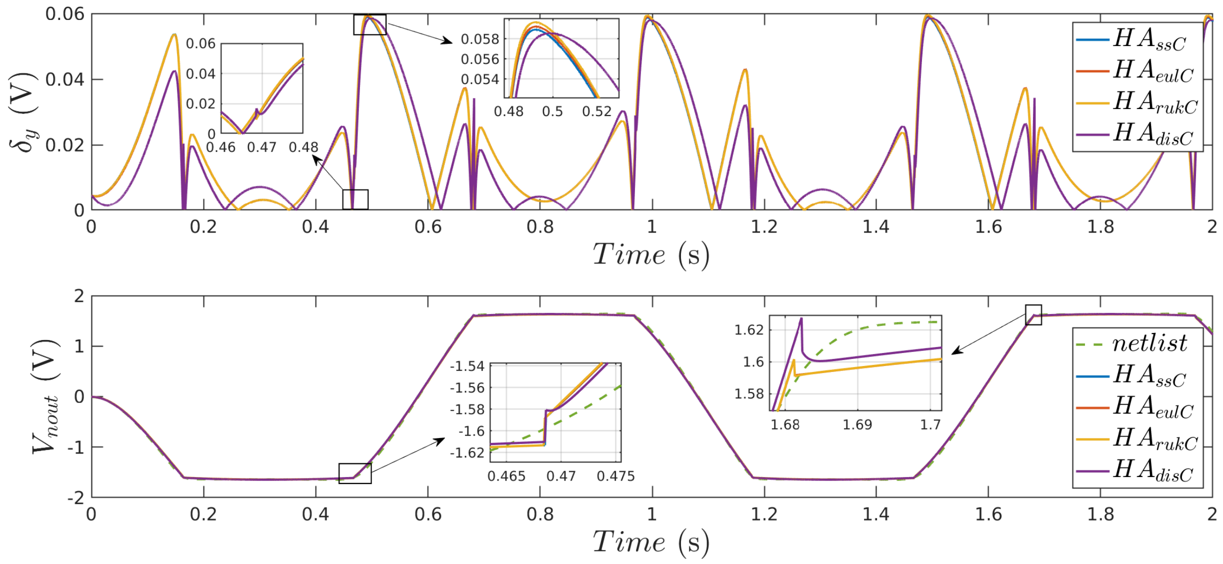


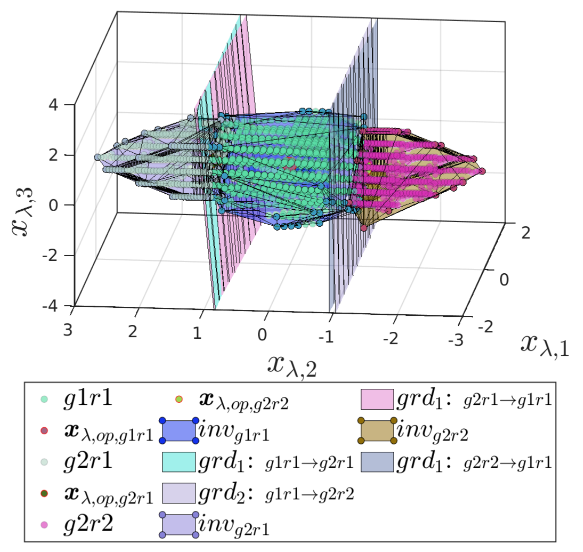
Fig. 7.18. Simulation results of the 4 generated SystemC-AMS models with $V_{nin} = 4 \cdot sin(2\pi t)$ V. In the first row, the output error δ_y is plotted, while in the second row the output voltages at V_{nout} of the models are compared to the output of the netlist.

are recompiled every time the input signal in the test-bench is changed. On the other hand, the SystemC-AMS models can be easily extended as stated in Section 5.1.2 to cover the deviations during the modeling process, thereby allowing for symbolic simulations with ranges in the inputs and the initial conditions on top of modeling the system behavior with parameter variations.

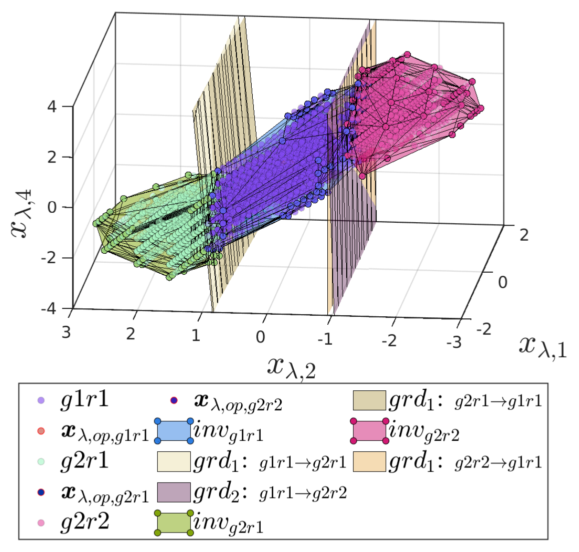
7.1.7 Fourth Order Lowpass Filter

Till now, the reduction order was set to two in the previous models. Increasing the reduction order in the example from Section 4.1 makes little sense, as the primary aim of the introduced methodology is to create an abstract model with a reduced order. However, in order to illustrate the presented approach from Chapter 4, a fourth order model was created by connecting to lowpass filters in series like in Sections 5.3 (see Fig. 5.18). In contrast to Sections 5.3, the two Spice netlists were connected and the obtained system was sampled and abstracted as one unit. Thus, the overall system has an order of $r = 28$, 34 transistors, and $n = 44$ nodal variables. An abstract HA in Verilog-A was generated with a reduction order set to $m = 4$, the *grdV method*, and with jump functions. The remaining settings are identical to the settings for the models from Section 7.1.5. The reduced state space of the system is illustrated in Fig. 7.19

The two filters are connected in series. The first filter has V_{dd} set to +1 V, while V_{ss} is set to −1 V, thereby limiting V_{nout} to the range $[-1, 1]$ V. The output voltage V_{nout} of the second filter is $[-1.65, 1.65]$ V, thus, this filter is similar to the running example from Section 4.1. However, in contrast to the filter from Section 4.1, the capacitors C1 and C2 are set to 0.001μF and 0.01μF for the second filter. As both filter have a gain of -0.88, the second filter will never go into limitation. Hence, the overall circuit has only three locations, in contrast to the compositonal model from Sections 5.3, which has for each filter three locations, thereby in total nine location (similar to product automaton). As observed in Fig. 7.19, the current location, which is determined by the current value of $\boldsymbol{x}_\lambda$ and its position with respect to the halfspace guards (see *grdV method*), is



(a)



(b)

Fig. 7.19. $\mathcal{S}_\lambda$ space of the generated abstract HA with three location. The first three dimensions are shown in (a), while (b) shows the forth dimension versus the first two dimensions.

mainly affected by the current value of $x_{\lambda,2}$.

For an input step function of amplitude $V_{nin} = -3$ V, the results of the simulation of the Verilog-A as well as the Spice netlist are illustrated in Fig. 7.20. The first column shows the voltages of the

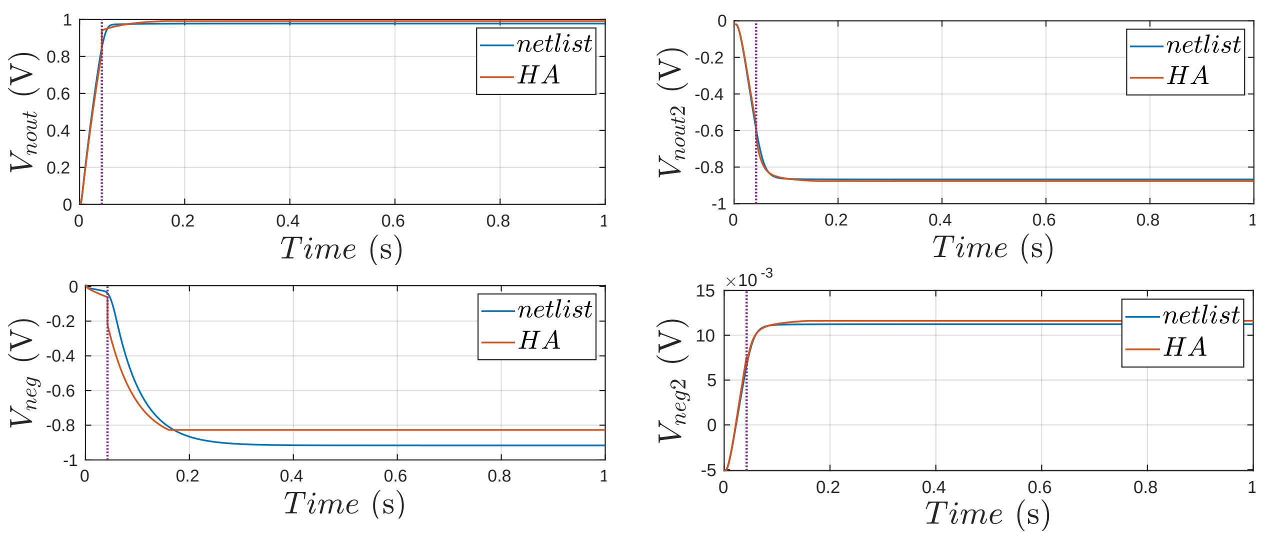


Fig. 7.20. Simulation results of the Spice netlist and the abstract HA. The first column shows voltages to the first filter, while the second column shows the voltages of the second filter. The purple vertical lines indicate when the HA changes the current location.

first filter, while the voltages at the second column correspond to the second filter. As observed, the first filter reaches saturation, in contrast to the second filter which stays in the linear region. While the voltage at the negative terminal of the operation amplifier corresponding to the second filter (V_{neg2}) is accurately modeled, V_{neg} corresponding to the first filter exhibits large deviations. These deviations can be reduced by either increasing the number of locations or the order of the abstracted HA.

The guards in the previous example were identified via the *distance method* and modeled as halfspaces. The examination of several aspects has been skipped in this dissertation for simplicity. This includes the over approximation of the *intersection method* compared to the *distance method* during the guard identification for HAs with large reduction orders.

7.2 Simple Circuit With a Diode

In Section 4.5.4, it was stated that the input can be modeled as an additional state in the $\mathcal{S}_\lambda$ space. This is sometimes necessary, especially if the system behavior varies significantly with the values of the inputs of the system. To emphasize this point, consider the netlist from Fig. 7.21.

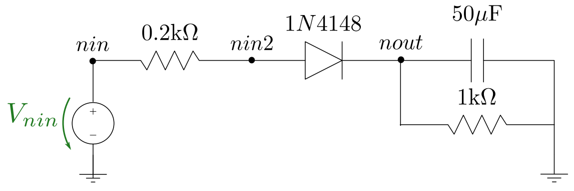


Fig. 7.21. Spice netlist containing a diode with a saturation current set to 100 fA.

Depending on the diode, the circuit from Fig. 7.21 exhibits a switching behavior between charging the capacitance when the diode conducts the current, and discharging the capacitance once the diode blocks the current. This netlist is provided to *Vera* with a reduced order set to $m = 1$.

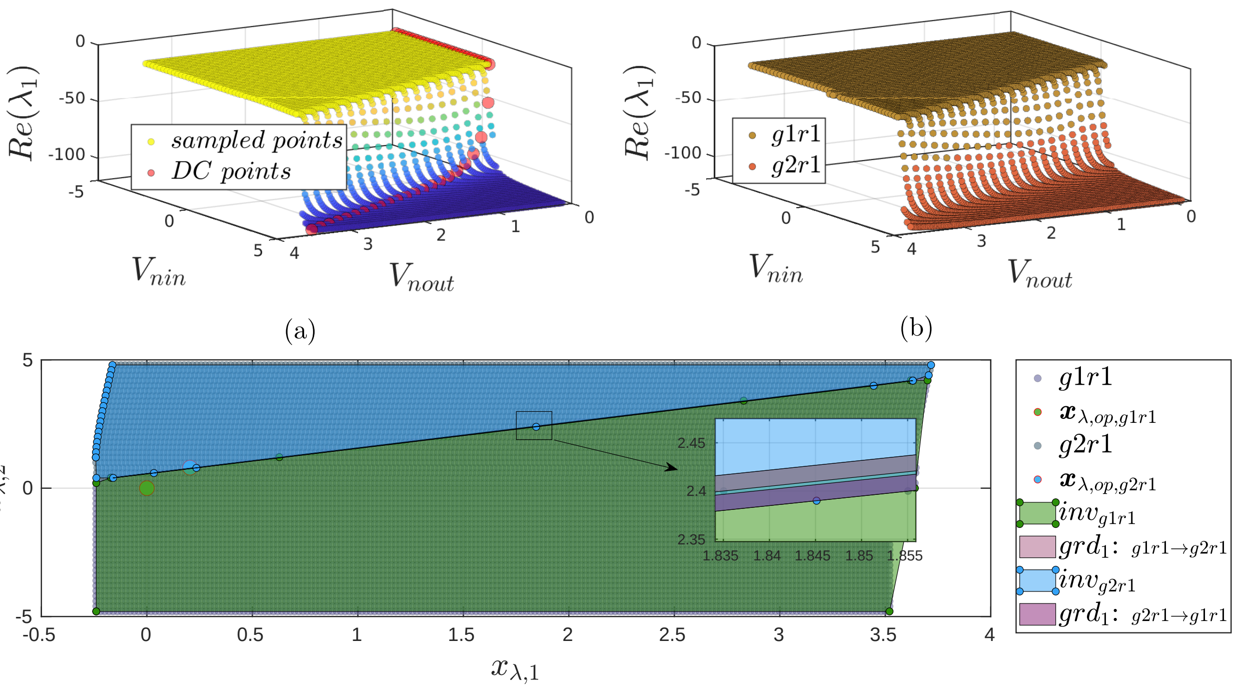


Fig. 7.22. Modeling the circuit in the $\mathcal{S}_\lambda$ space with the input as an additional state.

In Fig. 7.22a, *Amcvis* (Section 4.2) is used to examine the sampled data. As observed, when plotting the output of the system versus the input and real part of the only eigenvalue, a clear distinction in the behavior of the circuit can be observed, as the colors of the points are set according to the eigenvalue, and the behavior of the system varies depending on the value of V_{nin}.

The system is then abstracted via *Elsa* to a HA for the use with *Cora*. In Fig. 7.22b, the result of the location identification on the same previously used dimensions can be observed. Two locations have been thereby identified. The system at hand is modeled with polytopic guards and invariants. Moreover, the jump functions are to be considered. The result in the $\mathcal{S}_\lambda$ space before adjusting the location by the jump functions is illustrated in Fig. 7.22c. Note that $\boldsymbol{x}_{\lambda,2}$ represents the input of the system that has been modeled as a state according to Section 4.5.4. As observed in Fig. 7.22c, to distinguish the locations of the HA, it is not enough to consider only $\boldsymbol{x}_{\lambda,1}$, but both *states* must be considered simultaneously.

A reachability analysis has been performed in *Cora* using the generated model. The result of this analysis for a ramp at the input of the system, as shown in Fig. 7.23a, is illustrated in Fig. 7.23b. Both figures hold as well the simulation of the Spice netlist. Moreover, at time $t = 0$ s, the

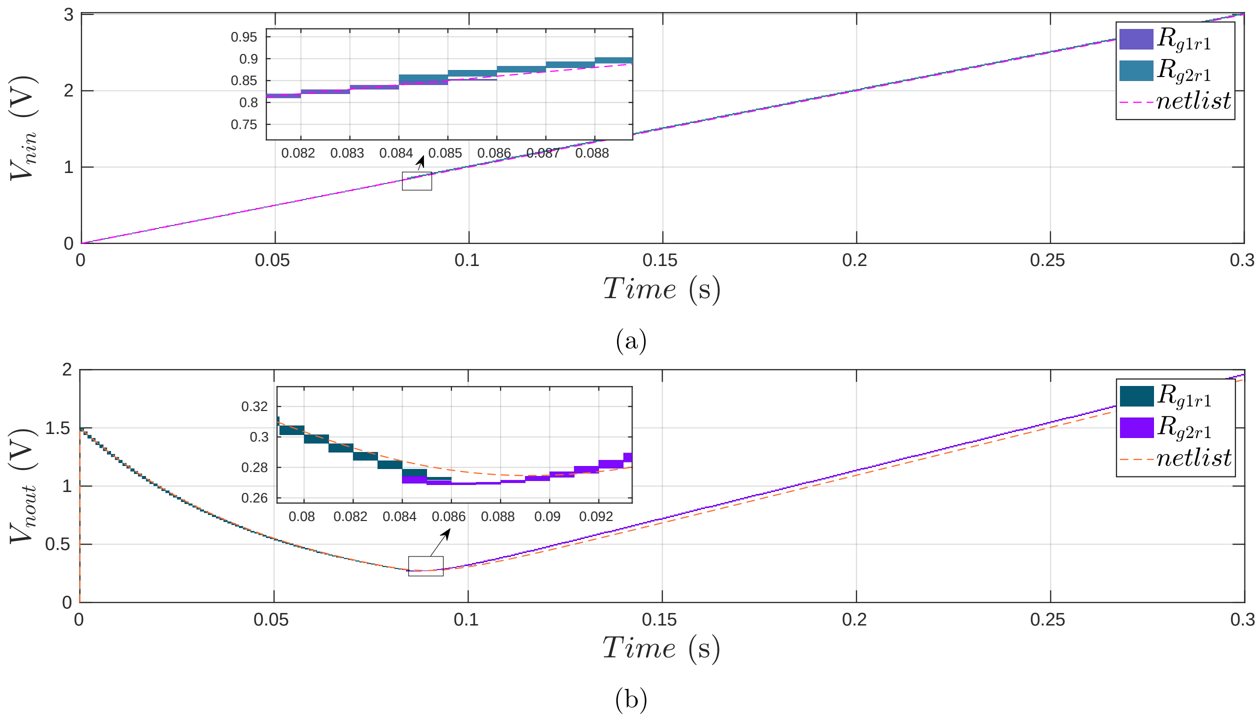


Fig. 7.23. Result of the reachability analysis (b) at the output node of the system for an input voltage shown in (a).

initial voltage at the capacitance has been set to $V_{nout} = 1.5$ V. As observed in Fig. 7.25, the abstract HA manifests a similar behavior as the original Spice netlist. In Fig. 7.24, the result of the reachability analysis is illustrated in the $\mathcal{S}_\lambda$ space which is illustrated against the location variable $loc \in \{g1r1, g2r1\}$. The system starts at $\boldsymbol{x}_{\lambda,1} = 1.5$ at location $g1r1$. The reachable set R_{g1r1} then intersects the guard $grd_1{:}_{g1r1 \to g2r1}$ at $\boldsymbol{x}_{\lambda,1} = [0.273, 0.279]$ and $\boldsymbol{x}_{\lambda,2} = [0.84, 0.85]$. At this instance a jump is performed into the next location $g2r1$, a reset is applied on the $\boldsymbol{x}_\lambda$ variables (Eq. (4.33)), and the reachability analysis is continued in the next location of the HA.

A second reachability analysis is performed with an uncertain initial state for the variable $V_{nout} = [1, 2]$ V. The result of the analysis is illustrated in Fig. 7.25b. Moreover, Fig. 7.25a presents the $\mathcal{S}_\lambda$

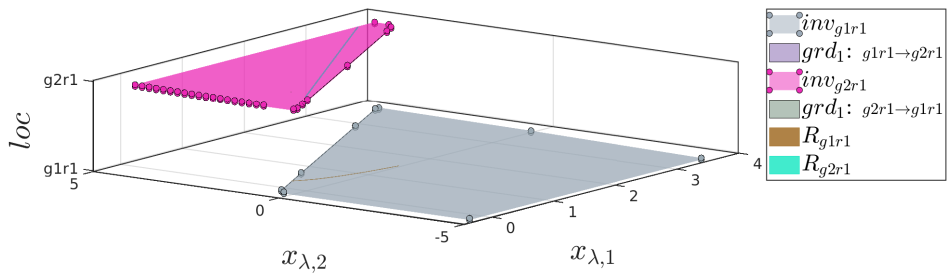


Fig. 7.24. Result of the reachability analysis with illustrated in $\mathcal{S}_\lambda$ space drawn versus the location variable $loc \in \{g1r1, g2r1\}$.

space with the adjusted locations of the HA (see Section 4.5.9). As observed, the system starts at $\boldsymbol{x}_{\lambda,1} = [1, 2]$. The reachable set R_{g1r1} then intersects the guard grd_1:$_{g1r1 \rightarrow g2r1}$ at $\boldsymbol{x}_{\lambda,1} = [0.22, 0.32]$ and $\boldsymbol{x}_{\lambda,2} = [0.8, 0.9]$. A jump is then performed into the second location $g2r1$ with the new reachable set R_{g2r1} starting at the marked region (moved due to jump functions). The reachability analysis continues at this location until the simulation time elapses.

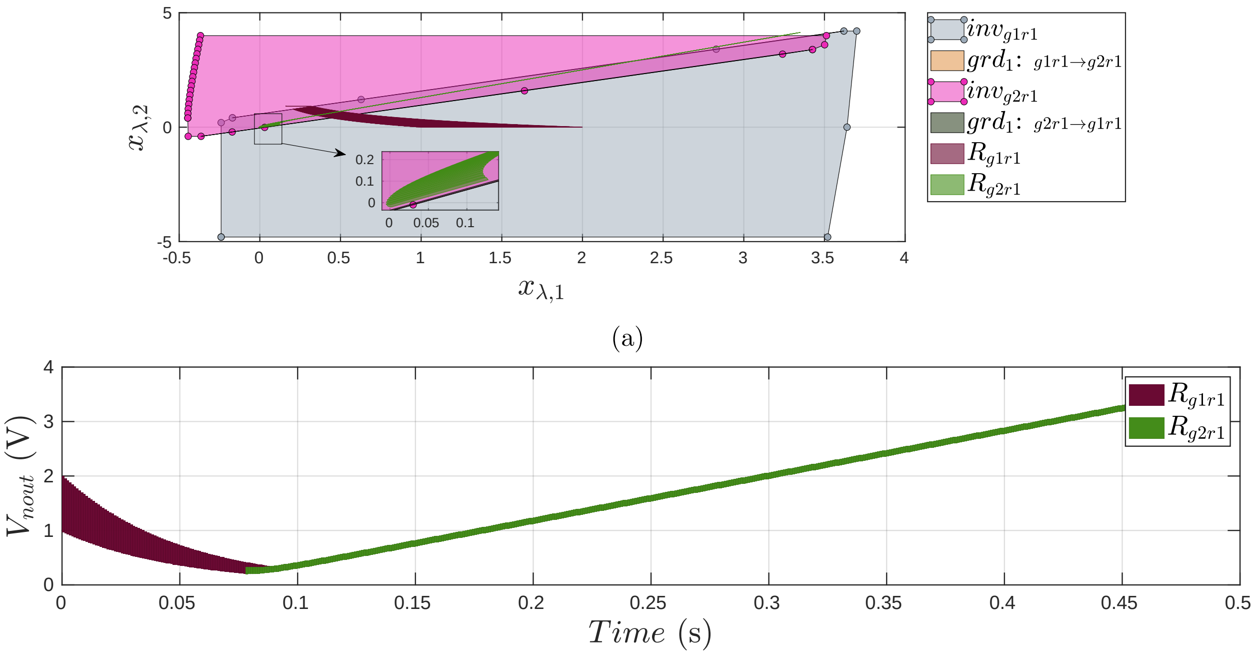


Fig. 7.25. Result of the reachability analysis at the (c) output of the system and (a) in the $\mathcal{S}_\lambda$ space for an input V_{nin} as shown in Fig. 7.23a with $V_{nout} = [1, 2]$ V.

7.3 GmC Filter

In this section, an industrial full differential second order GmC filter is abstracted with the proposed methodology into a HA in Verilog-A and SystemC-AMS syntax. An overview of the circuit is shown in Fig. 7.26. The circuit consists in total of 46 transistors and 38 nodes, and thus has $n = 38$ variables in the $\mathcal{S}_o$ space. The initial dynamic order of $r = 26$ of the netlist is reduced by *Vera* to $m = 2$. In 243.41 s, *Vera* sampled 27191 points, thereby marking only 5501 of these points as

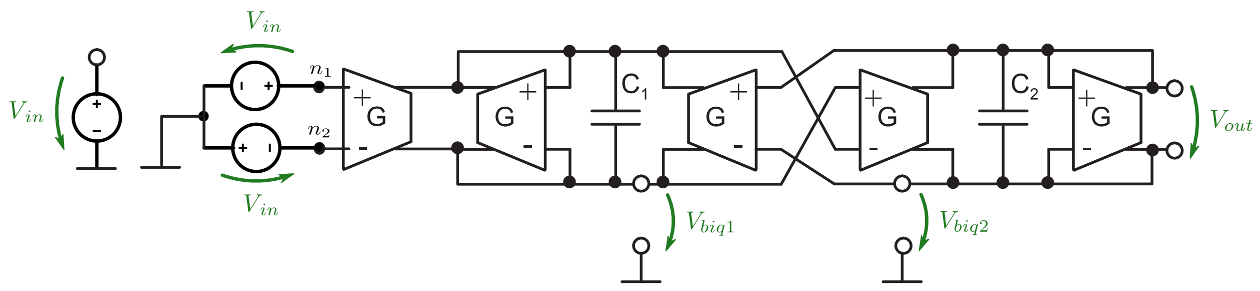


Fig. 7.26. Schematic of a Spice netlist describing a GmC filter.

reachable. In Fig. 7.27a the reachable points are illustrated, while Fig. 7.27b holds all the sampled points. Both figures shown the $\mathcal{S}_\lambda$ space versus the real part of the first eigenvalue. For the specified reduction order, two complex conjugated eigenvalues are obtained.

For this example, there is great difference if all points are considered in the model abstraction process, or only the reachable marked ones. In fact, all sample points that are not reachable might mislead the behavioral abstraction, as for example the range of the real part of the first eigenvalue increases significantly in case these points are considered. For the current abstraction, only the reachable points are used. Hence, the *reach filter* is used to modify the $\mathcal{S}_\lambda$ space (Section 4.5.5).

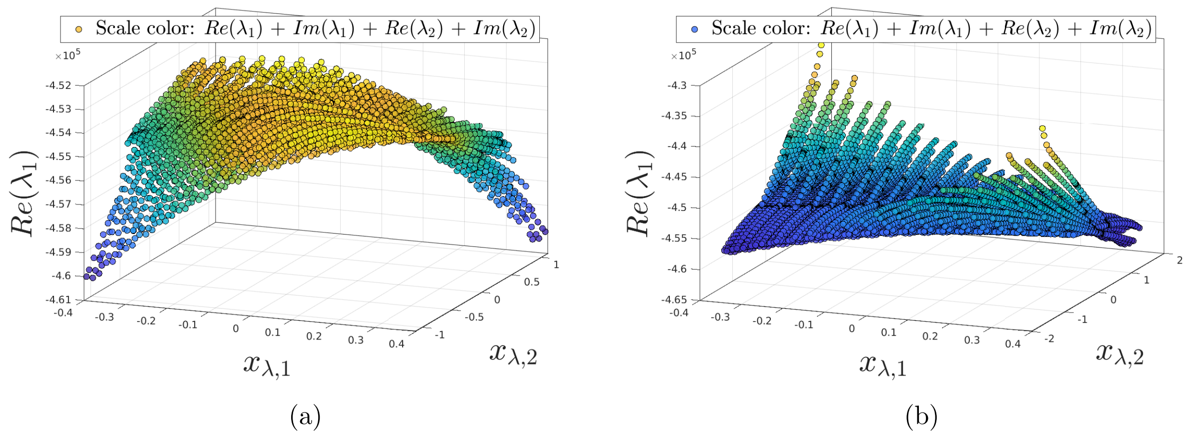


Fig. 7.27. The reduced state space of the system plotted versus the real part of the first eigenvalue. In (a) only the reachable points are illustrated, while in (b) all sampled points are plotted. The colors in both figures are controlled by the two eigenvalues of the system.

As the intention is to generate a Verilog-A model first, the *grdV method* (Algorithm 8) is used here. With the sampled data at hand, *Elsa* is launched to generate the abstract HA. Unlike demonstrated in Section 4.5, the modeling is performed directly in the $\mathcal{S}_\lambda$ space without using the S_{virt} space. The result of the location identification is presented in Fig. 7.28a. Three locations $loc \in \{g1r1, g2r1, g2r2\}$ have been identified by using eigenvalue clustering for the group identification (Section 4.4.1) followed by using the *Gdist method* for the region identification (Section 4.4.2).

Additionally to the *reach filter*, the *lsq filter* was used. However, the $\mathcal{S}_\lambda$ space was not recalculated. The system behavior was described by using the mean values of all obtained points belonging to a location (*mean method*). The invariants were specified as polytopes. The dominant guards

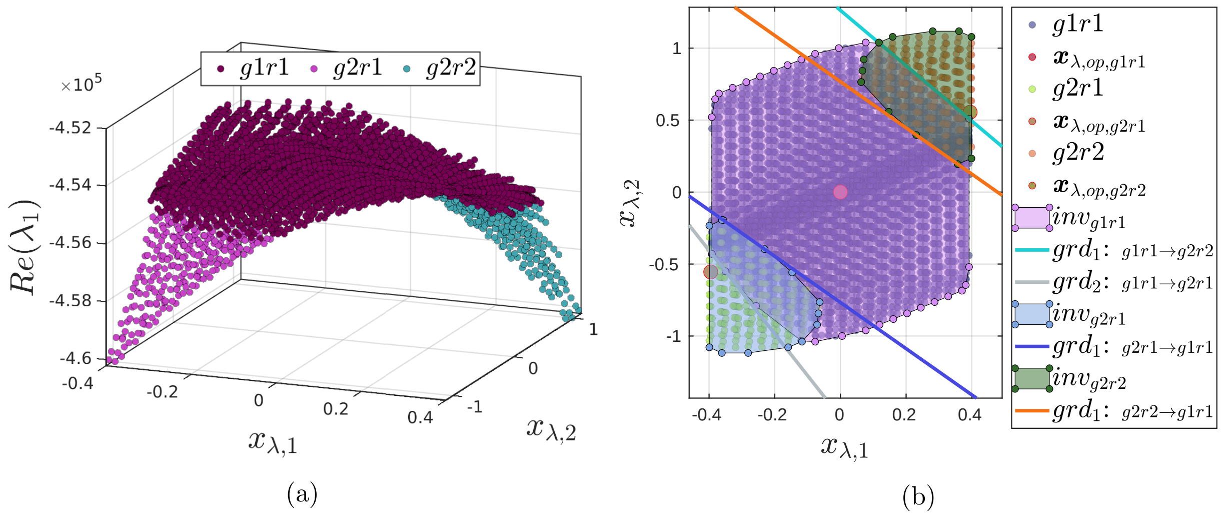


Fig. 7.28. In (a) the reduced state space of the system plotted against the real part of the first eigenvalue after the location identification is illustrated, while in (b) the identified guards and invariants of the HA are illustrated.

were identified by the *distance method* (Section 4.5.7) and modeled as halfspaces. The generated model is presented in Fig. 7.28b. The system is modeled with jump functions, which for Verilog-A models is realized with a shift vector $\boldsymbol{v}_s$ (Section 4.6.2). Finally, the model was deployed using the *grdV method* in Verilog-A syntax. The overall modeling process of *Elsa* consumed 3.08 s.

A simulation with a sine signal of amplitude 0.5 V and offset of 1.22 V with a frequency at 1 KHz is applied at V_{in}. The results of the simulation is illustrated in Fig. 7.29. The result of the simulation is illustrated in Fig. 7.29, which as well hold the Spice simulation of the original netlist. The output error δ_y between the netlist and the generated HA is shown in the first row of this figure. As observed, the output error reaches its maximum during the transitions of the location, which is indicated by the change in $\boldsymbol{v}_{s,1}$ and the vertical purple lines.

In Table 7.7,the simulation results are closer examined. The modeling time in Table 7.7 includes

Table 7.7: Comparison of the abstracted Verilog-A HA and the original Spice netlist

Circuit	Model abstraction time (s)	Run time (s)	Time steps	Newton Iter.	$\hat{\delta}_y$ (V)	$\hat{\delta}_{y,r}$ (%)	Speed up
HA	243.41+4.01+3.08	0.14	10010	37296	0.098	9.8	125.1
Netlist	-	17.41	10007	50204	-	-	-

$\hat{\delta}$ stands for the maximum value of δ.

the sampling time of *Vera*, the time needed to import the sampled data into Matlab using *spaceM* (Section 3.4), and the abstraction time needed by *Elsa* in this order. Compared to [TH19b], the data import time was significantly reduced. As observed in the table, the abstraction approach yields a speed up of 125 with an acceptable accuracy. As the GmC filter has been excited with a relative large input voltage for its design, the system has been pushed into the limiting behavior rapidly, favoring the deviations to the netlist.

Disregarding the output error, there is a downside to the abstracted model. Due to the consider-

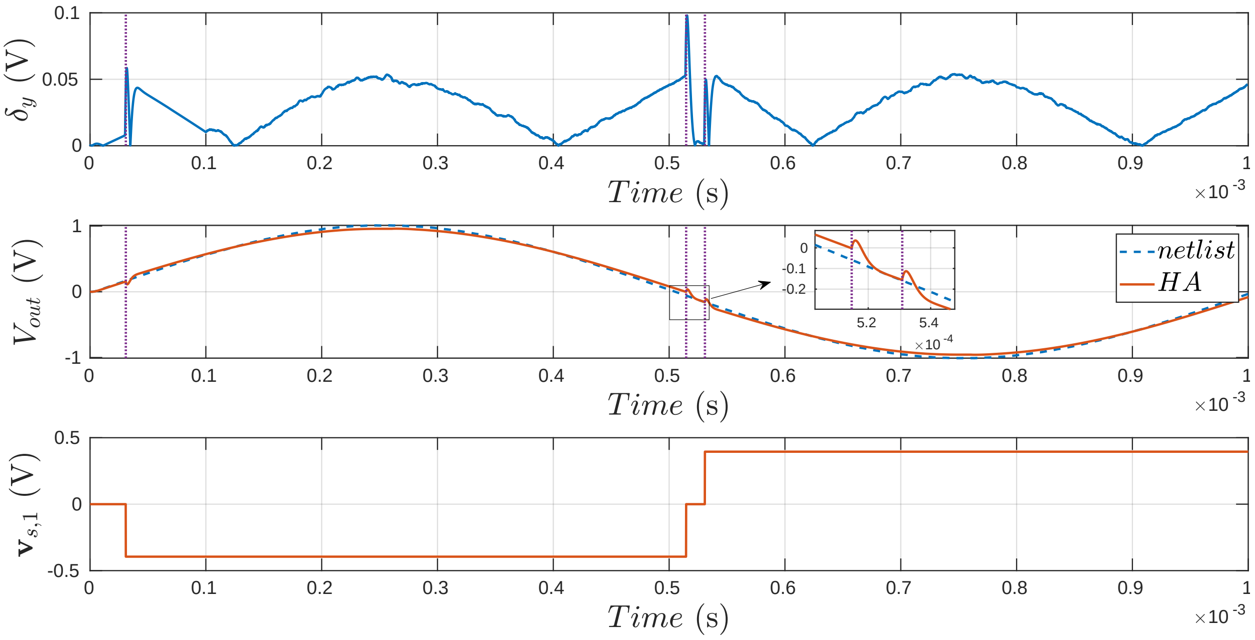


Fig. 7.29. Result of a simulation performed on the generated HA and the Spice netlist. The first row shows the output error δ_y, while the second row shows the voltages at the output of both systems. The third row shows the first dimension of the shift vector $\boldsymbol{v}_s$.

able order reduction, not all internal voltages can be exactly reconstructed as shown in Fig. 7.30. The first row of this Fig. 7.30 shows the voltages at the two internal nodes $n1$ and $n2$. As observed,

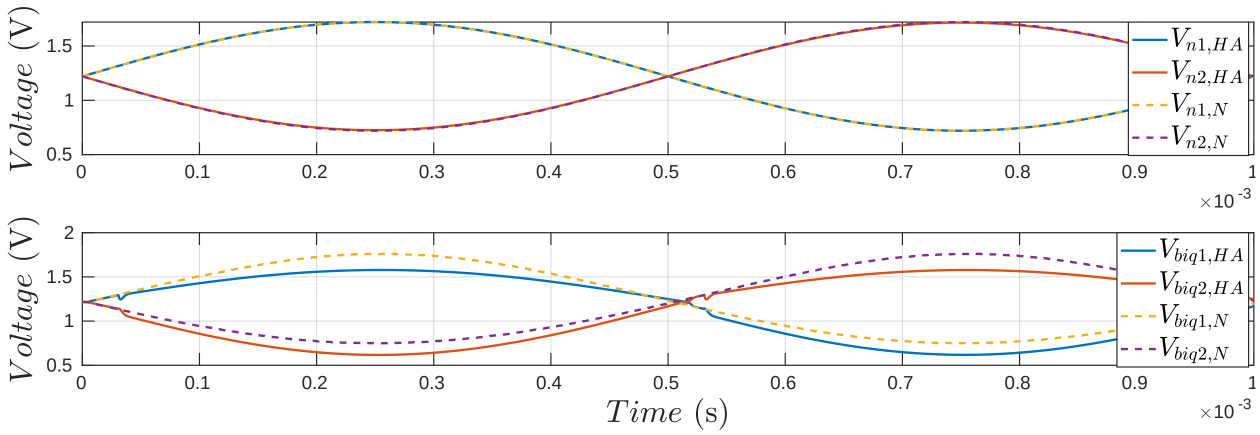
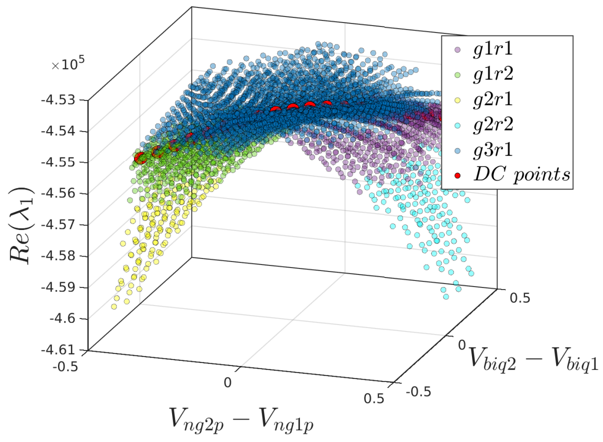


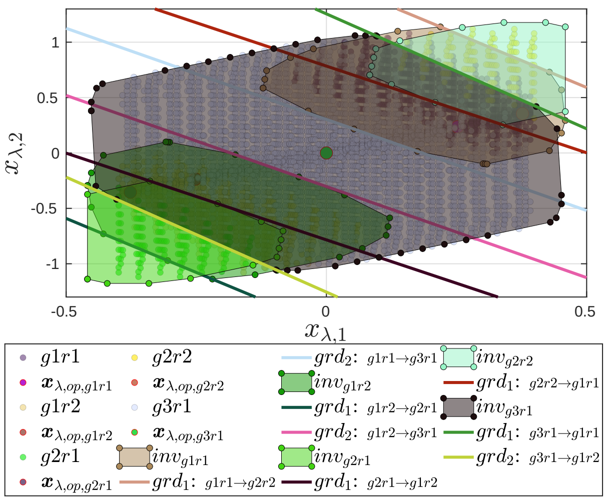
Fig. 7.30. Result of a simulation performed on the generated HA and the Spice netlist. The first row shows the voltages at the nodes $n1$ and $n2$, while the second row presents the voltages V_{biq1} and V_{biq2}. The subscripts N and HA denote the correspondences to the netlist and abstract model, respectively.

the abstract model successfully reconstructed these voltages. However, for the internal voltages V_{biq1} and V_{biq2}, there is a strong deviation from the signals of the netlist. Note that the difference between these signals is nearly the same. Thus, even though the abstract model failed to reconstruct these voltages, it preserved the difference between them. This is a result of the full differential architecture. The nonlinear behavior disturbs both signals V_{biq1} and V_{biq2} in the same way, which is absent in the linear location $g1r1$.

Usually, the accuracy of the model can be increased by increasing the number of locations as performed in Section 7.1.5 up till a certain degree. However, This is not always the case, especially if the additional obtained location do not contain DC points. To demonstrate this, the GmC filter was abstracted with the same settings as previously, with the difference that the HA was modeled with 5 locations. This is illustrated in Fig. 7.31a, which shows the $\mathcal{S}_{virt}$ space of the GmC filter drawn versus the real part of the first eigenvalue. In Fig. 7.31b the invariants and guards of the HA are shown in the $\mathcal{S}_{\lambda}$ space. Note that the modeling time consumed by *Elsa* increased to 5.45 s.



(a)



(b)

Fig. 7.31. The (a) $\mathcal{S}_{virt}$ space drawn versus the real part of the first eigenvalue and (b) $\mathcal{S}_{\lambda}$ space of the HA with 5 locations.

The same simulation as previously is performed on the generated HA. The result as well as the result of the simulation of the Spice netlist are presented in Fig. 7.33. As observed, the output error increased even though the HA was modeled with more locations than previously. Moreover, this error attains its maximum values during the transition of the current location, which is indicated by the vertical purple lines in Fig. 7.32. This can be mainly traced back to the selection of the operating point and the position of new found guards. Since the two locations $g2r1$ and $g2r2$ have no DC points (see Fig. 7.31a), the operating point is computed by taking the mean of the points belonging to this location. This does not assure that the best suitable point is selected. Even though different methods have been deployed to calculate a more suitable operating point, they are not handled here for simplicity. One approach to solve this problem is for example to sample more points, however, this is currently not the scope. Considering the guards, as the guards changed, the transition between the locations occur at different instances than previously. Beside the previous stated aspects and the errors arising during transitions, the accumulated error in a location can be traced back to the guards as well as to the transformation, system, and input matrices (*mean method*). Another downside which can be observed in Fig. 7.32, is the fluctuation of the current location at t=0.471 s. On the other hand, the internal voltages V_{biq1} and V_{biq2} modeled by the HA with 5 locations are slightly more accurate than obtained previously with the HA with 3 locations.

To control and decrease the error between the transitions, the *invV method* can be used to model

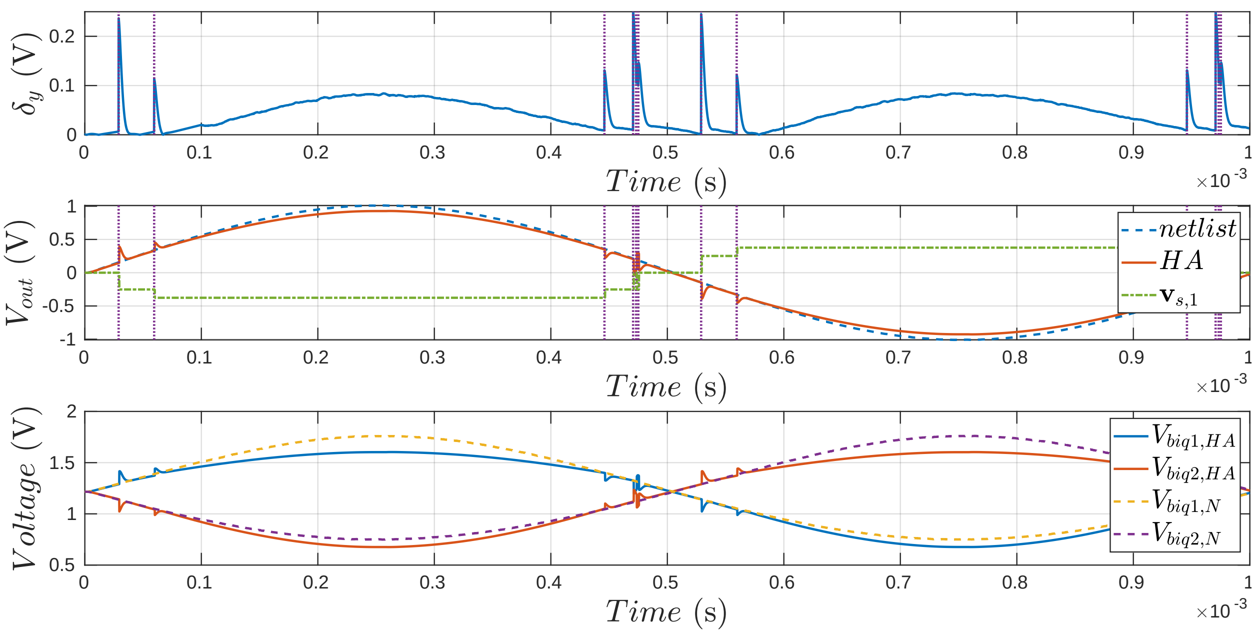


Fig. 7.32. Result of a simulation performed on the Spice netlist and the generated HA. The HA was generated with the *grdV method* with 5 locations. The first row shows the output error between the netlist and the HA. The second row shows the output voltage V_{out}, while the third row shows the internal voltages V_{biq1} and V_{biq2}.

the HA in Verilog-A. Using the same settings as previously except the model description method, a HA was generated with the *invV method*. For the same input voltage, a simulation was performed on the generated HA and the original Spice netlist. The result is illustrated in Fig. 7.33. As shown, the error during the transition of the locations decreased significantly. However, the accumulated

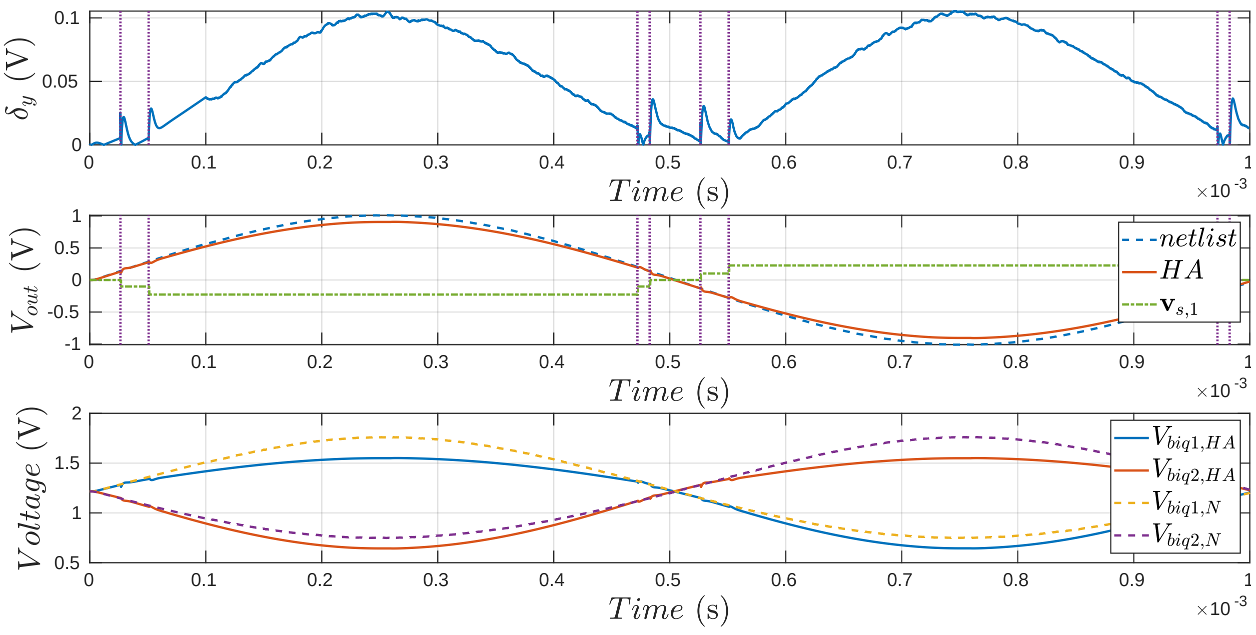


Fig. 7.33. Result of a simulation performed on the Spice netlist and generated HA. The HA was modeled with the *invV method* and with 5 locations.

error increased. This is due to the fact that current location is determined by the distance to the operating points (Algorithm 10), which often results in too early or too late transitions.

Another approach to increase the accuracy of the system and the modeled internal states, is to increase the reduced order m, thereby gaining more internal dynamic voltages. On one hand, this improves the exhibited behavior of the HA. On the other hand, since the current aim is to create a HA with a significant lower order (reduction orders) compared to the netlist, a different approach is taken that uses affine forms (Section 5.1.2). To demonstrate this, first a SystemC-AMS model was created with the same settings as one of the previous models with the *grdV method* and 5 locations. *Elsa* consumed 5.48 s for the generation of the HA. Note that, the SystemC-AMS version of the *grdV method* passes the location variable *loc* during the simulation (Algorithm 11). Hence, *loc* can be used to indicate the current location.

As previously, a simulation was performed on the netlist and the HA generated in SystemC-AMS syntax. The results are illustrated in Fig. 7.34. The first row shows the output error δ_y between the netlist and the generated HA. The second row shows the output voltages, while the third row shows the internal voltages V_{biq1} and V_{biq2} of both systems. The vertical purple lines indicate a location switch. As observed, the output of the HA exhibits deviations slight from the output of

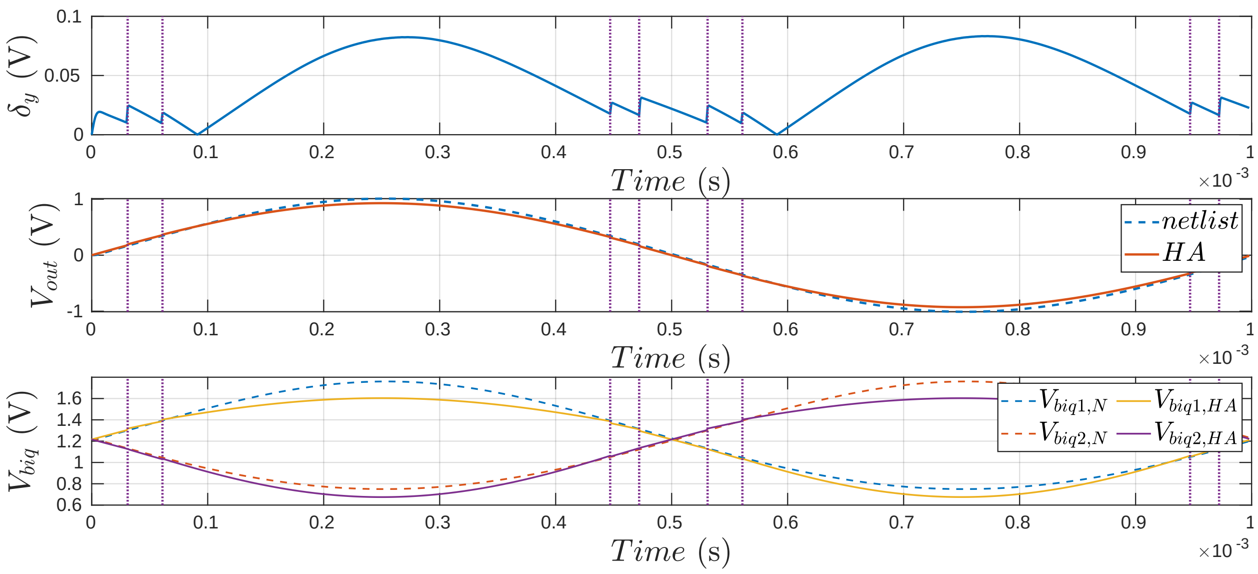


Fig. 7.34. Result of a simulation performed on the SystemC-AMS generated HA with 5 location and the Spice netlist.

the netlist. Compared to the results of the Verilog-A HA with 5 locations and generated with either method (*grdV* or *invV*), the System-AMS model controls the error during the transition of the location significantly better. However, there are still deviations from the behavior of the netlist, which can be traced back to the system and transformation matrices which were calculated by the *mean method*. This is as well true for the internal voltages V_{biq1} and V_{biq2}.

To obtain a better model, the approach from Section 5.1.2 is used to extend the model with affine forms from the AADD library. This is performed by hulling the elements of the matrices $\boldsymbol{A}_{loc_i}$ and $\boldsymbol{B}_{loc_i}$, $\boldsymbol{F}_{loc_i}$, and $\boldsymbol{L}_{loc_i}$ by the specified geometric shape. In this case for example, $\boldsymbol{A}_{loc_i} \in R^{2\times 2}$ has 4 dimensions corresponding to the real and imaginary values of its two eigenvalues. For simplicity,

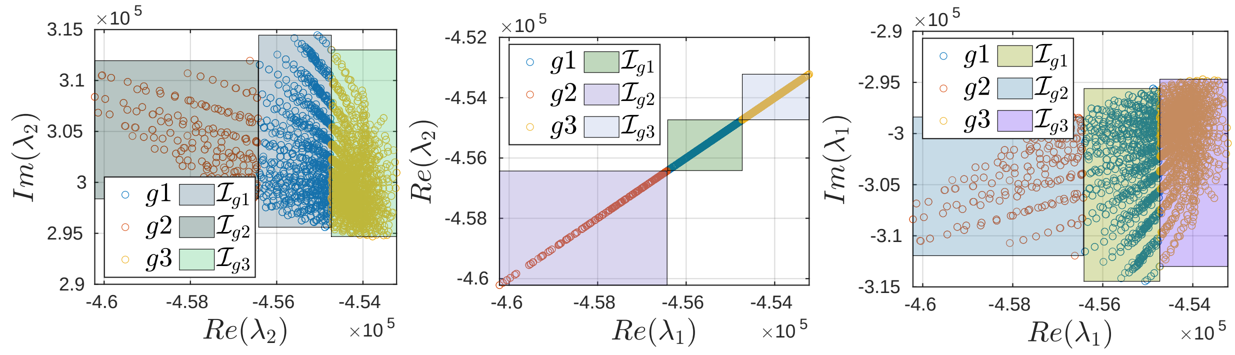


Fig. 7.35. Hulled eigenvalues of the sampled points of the GmC filter. For each group g_j, an interval hull $\mathcal{I}_{gj}$ is used to hull the eigenvalues.

an interval hull ($\mathcal{I}$) will be used here, which correspond to an affine description with a single symbolic variable. In Fig. 7.35, the hulled eigenvalues are illustrated. With the convex hull, in this case interval hull, at hand, the entities of the matrices are described according to Section 5.1.2. Additionally, to compensate for the sampling performed by *Vera*, the sampling state step is added to the $\boldsymbol{x}_\lambda$ variables as an uncertainty. Moreover, the guards descriptions are extended with ifS and elseS statements. For the generation of the model, *Elsa* consumed this time 5.58 s. For the same input voltage as previously, the simulation results are illustrated in Fig. 7.36. In the third row of

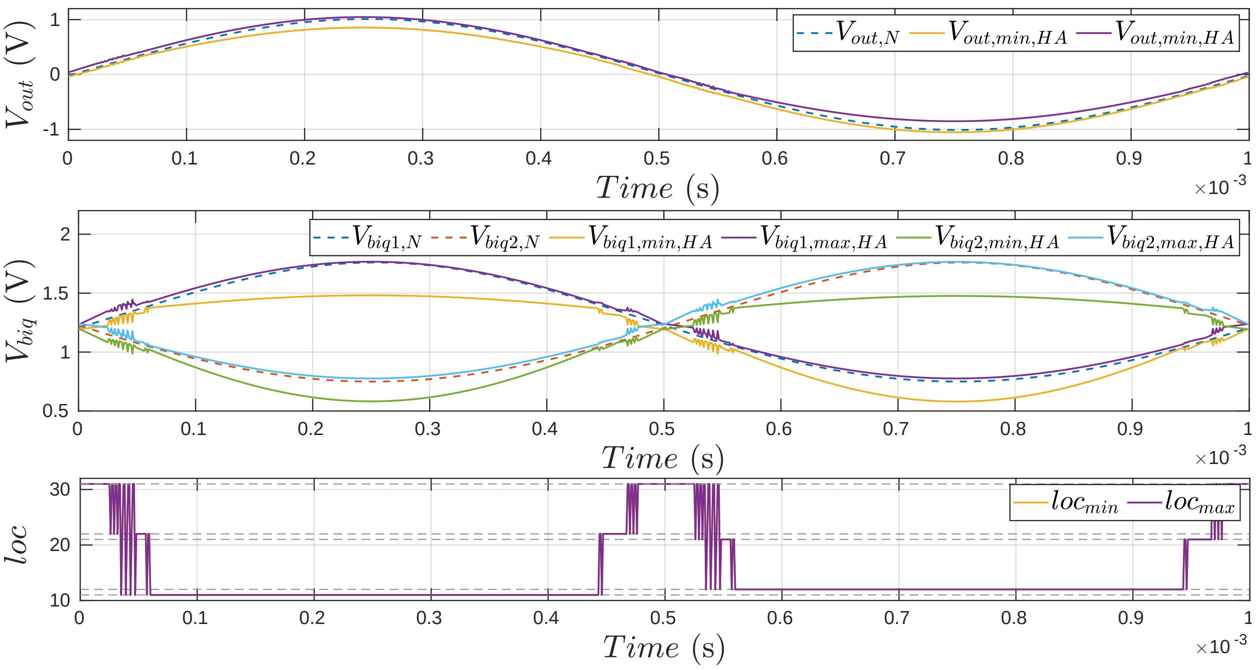


Fig. 7.36. Result of the simulation of the netlist compared to the result of a symbolic simulation conducted on a deterministic HA with affine forms.

Fig. 7.36, the location variable *loc* is illustrated. As observed, fluctuations are presented in the results, especially in the internal signals. To eliminate these fluctuations, the HA is changed from a deterministic HA to a non-deterministic HA, by modeling the location variable *loc* by affine forms and changing the if and else conditions involved with this variable into ifS and elseS conditions. The simulation results are illustrated in Fig. 7.37.

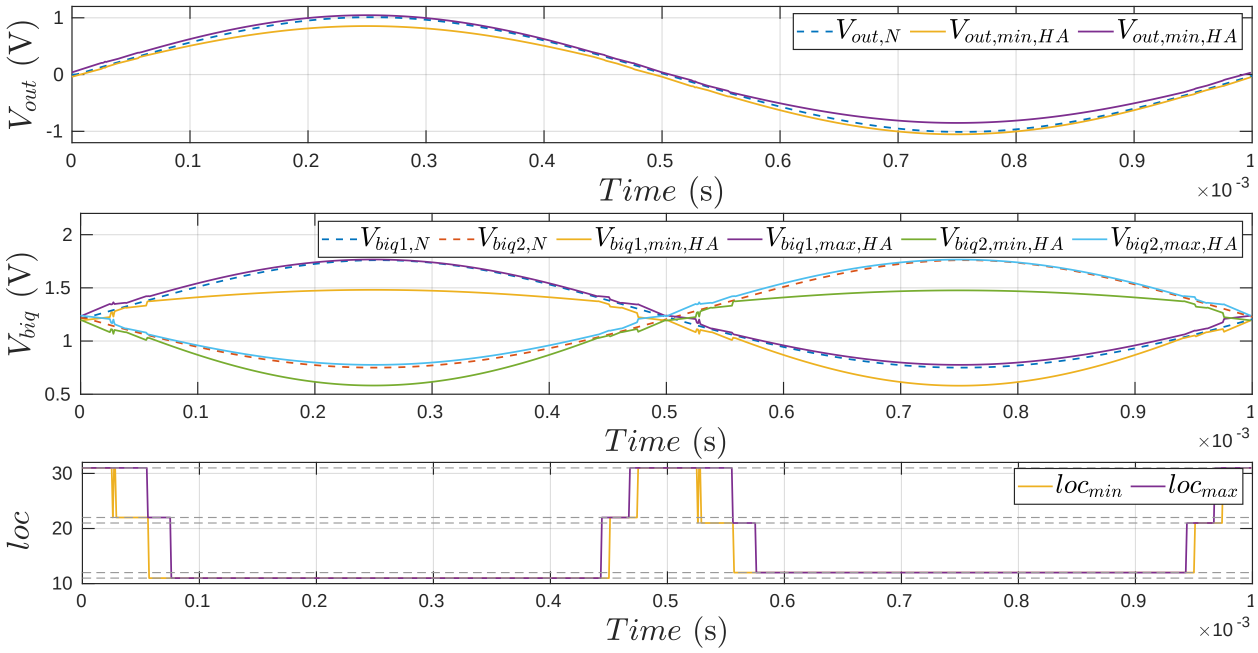


Fig. 7.37. Result of the simulation of the netlist compared to the result of a symbolic simulation conducted on a non-deterministic HA with affine forms. The last row shows the location variable loc attains a range during simulation.

As observed in Fig. 7.37, the location variable loc can attain various values in a single time step, which makes the HA non-deterministic. This eliminates the fluctuations of the signals, and encloses the behavior of the Spice netlist with little over approximations. The over approximations can be reduced by using more than a single symbolic variable to model the matrices of the HAs. This corresponds to a zonotopic description of these matrices, instead of using an interval description as performed here.

7.4 Compositional Abstraction: Control Loop

In this section the extended approach from Section 5.3 is illustrated upon an example from the automotive industry. The control loop from Fig. 7.38 is abstracted by performing a compositional abstraction.

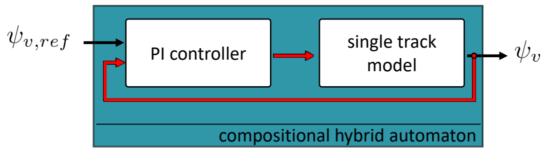


Fig. 7.38. Abstracted control loop consisting of a PI controller and a single track model.

Consequently, the components of the control loop, the PI controller and the single track model, are internally abstracted by *Elsa* to HAs, followed by the generation of the compositional HA (CHA) from these elements. As stated in Section 5.3, this process is completely automated. In the following, the abstraction of the elements of the CHA are handled first separately. The generated CHA is then presented in Section 7.4.3.

7.4.1 Single Track Model

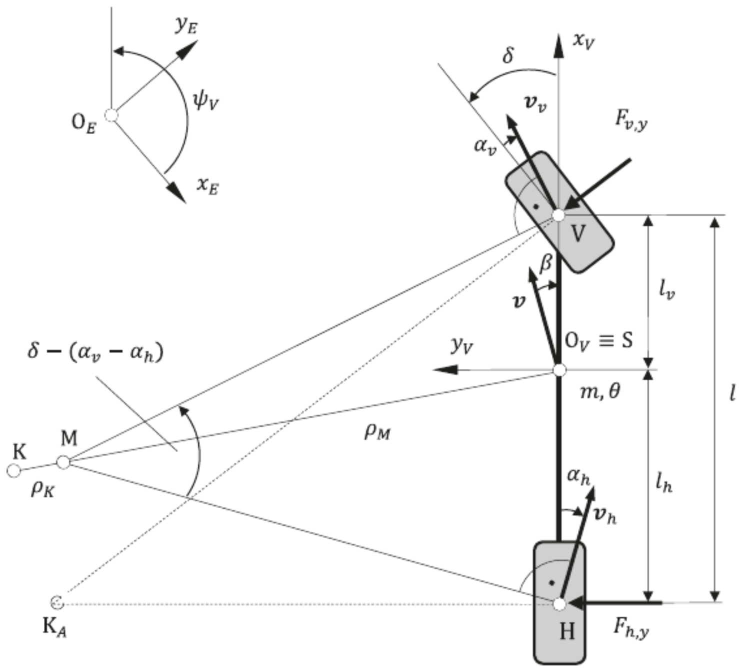


Fig. 7.39. Mathematical description of a single track model [SHB18].

Fig. 7.39 shows the used single track model [SHB18]. The yaw rate ψ and the slip angle β represent the two states of the system. The steering angle δ is the input of the single track model. With C, m, θ and v representing the tire stiffness, mass of the vehicle, moment of inertia around the z-axis, and the constant speed of the car, respectively, the state space equations of the system is described as:

$$\begin{bmatrix} \dot{\psi}_v \\ \dot{\beta} \end{bmatrix} = \begin{bmatrix} -\frac{c_{\alpha,v} l_v^2 + c_{\alpha,h} l_h^2}{v\Theta} & -\frac{c_{\alpha,v} l_v - c_{\alpha,h} l_h}{\Theta} \\ -1 - \frac{c_{\alpha,v} l_v - c_{\alpha,h} l_h}{v^2 m} & -\frac{c_{\alpha,h} + c_{\alpha,h}}{vm} \end{bmatrix} \begin{bmatrix} \psi_v \\ \beta \end{bmatrix} + \begin{bmatrix} \frac{c_{\alpha,v} l_v}{\Theta} \\ \frac{c_{\alpha,v}}{vm} \end{bmatrix} \delta \tag{7.1}$$

As shown in Eq. (7.1), the single track model used is a linear second order system. This model is described in Verilog-A using two equations. Still, as the intention is to use this model as a HA in *Cora*, the model is abstracted by *Elsa* according to Chapter 4. In the first step, *Vera* samples the circuit with 78106 points in 106.28 s. The data is imported into Matlab in 5.07 s. The abstraction performed by *Elsa* consumed 0.39 s. The generated HA has only one location $g1r1$, a single polytopic invariant, and no guards.

Two reachability analysis were performed on the abstracted model, both with a constant step function at the input $\delta = 0.15$ rad. The second reachability additionally considers an input deviation of 0.1 rad, thereby the input becomes $\delta = [0.05, 0.25]$ rad. The results are presented in Fig. 7.40 for the yaw rate ψ_v (first column) and the slip angle β (second column). As observed, the system has a linear behavior.

7.4.2 PI-Controller

The PI controller is abstracted next. The schematic of the controller is illustrated in Fig. 7.41.

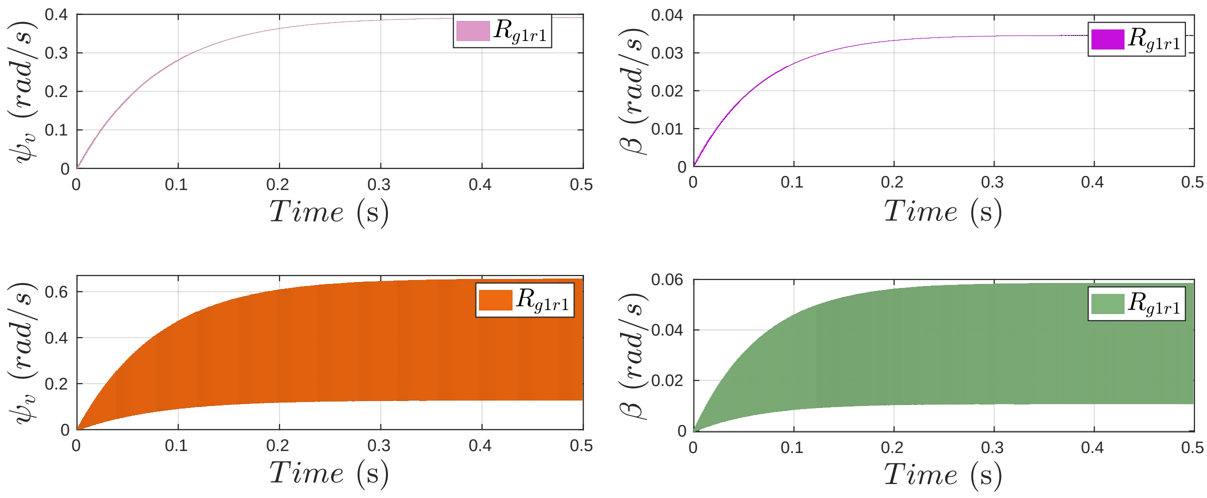


Fig. 7.40. Results of the reachability analysis with $\delta = 0.15$ (1st row) and $\delta = [0.05, 0.25]$ (2nd row).

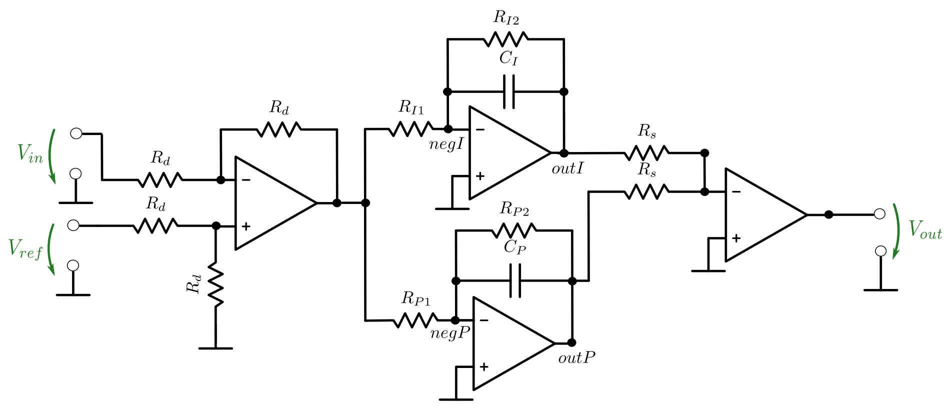


Fig. 7.41. Schematic of the PI controller: V_{in} and V_{ref} are the inputs. For the integral part a finite large parasitic resistor R_{I2} is added to define the maximum output voltage. For the proportional part a small parasitic capacitance C_p is added to model the timing behavior.

The controller has two input voltages, V_{in} and V_{ref}, and only one output voltage V_{out}. Note that the PI controller has a capacitance at both parts: the integration and the proportional parts. At the integration part, a large capacitance is used, while the capacitance at the proportional part is very small. The capacitance at the proportional part is necessary to avoid working with a DAE system with an index of nilpotency $\eta > 1$ (Appendix A.1).

The Spice netlist has $n = 63$ variables and a dynamic order $r = 53$ (Eq. (3.19)), with eigenvalues ranging from -3.3×10^2 till -6.4×10^{11}. This order is reduced to $m = 2$ by *Vera*. During a sampling time of 551.58 s, *Vera* sampled 8000 points. The sampled points are illustrated in the $\mathcal{S}_{virt}$ space plotted against the output of the system V_{out} in Fig. 7.42a and in the $\mathcal{S}_\lambda$ space plotted against the real part of the first eigenvalue in Fig. 7.42b using *Amcvis*.

In 0.39 s, *spaceM* imports the sampled data into Matlab's memory. With the sampled points at hand, *Elsa* abstracts the system into a HA in 3.60 s. During this abstraction, *Elsa* performs the group identification by eigenvalue clustering. To identify the regions of a group, *Elsa* uses the

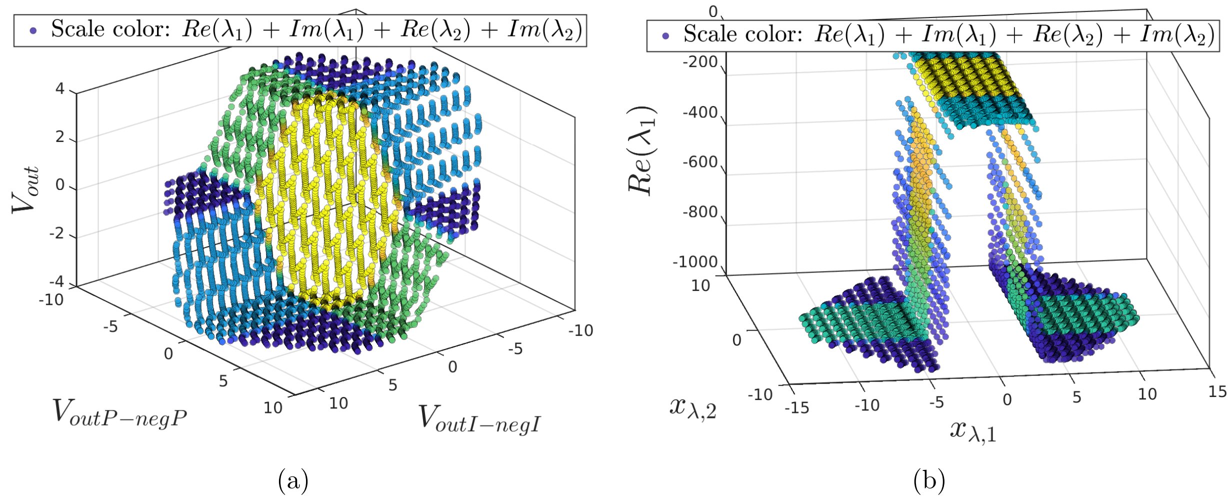


Fig. 7.42. In (a) the $\mathcal{S}_{virt}$ space is illustrated versus the output of the PI controller V_{out}, while in (b) the $\mathcal{S}_\lambda$ space is illustrated against the real part of the first eigenvalue λ_1.

clustering method DBSCAN from Section 4.4.2 with ϵ specified as the minimum state step in the $\mathcal{S}_\lambda$ space. The result of the region identification is presented in Fig. 7.43. Note that before DBSACAN

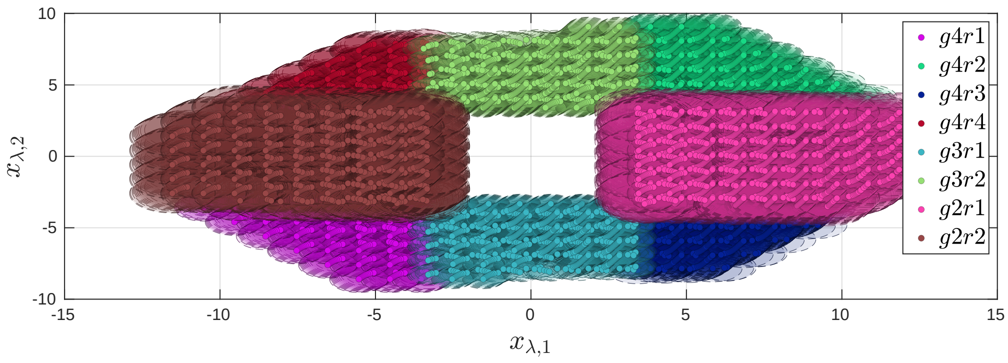


Fig. 7.43. Result of the region identification using the DBSCAN clustering algorithm on each group.

is launched, the necessity of a region identification is analyzed based on distances between the points of a group (Section 4.4.2). For the example, the region identification for the center location $g1r1$ is skipped and therefore missing in Fig. 7.43.

For simplicity, the invariants were specified as intervals, while the guards were modeled as polytopes identified via the *distance method* in the $\mathcal{S}_\lambda$ space. Moreover, the HA is modeled with jump functions. The resulted model is illustrated in Fig. 7.44 before the application of the jump functions. As observed, 9 locations have been identified. Group $g1$ represents the location where the PI controller exhibits a linear behavior. In group $g3$ and $g2$ the I-part and the P-part, respectively, go into saturation, while in group $g4$ both parts become saturated.

7.4.3 Compositional Model

A compositional automaton can be generated for the control loop shown in Fig. 7.38 using the approach from Section 5.3. Additionally to the sampled data of both systems, the single track

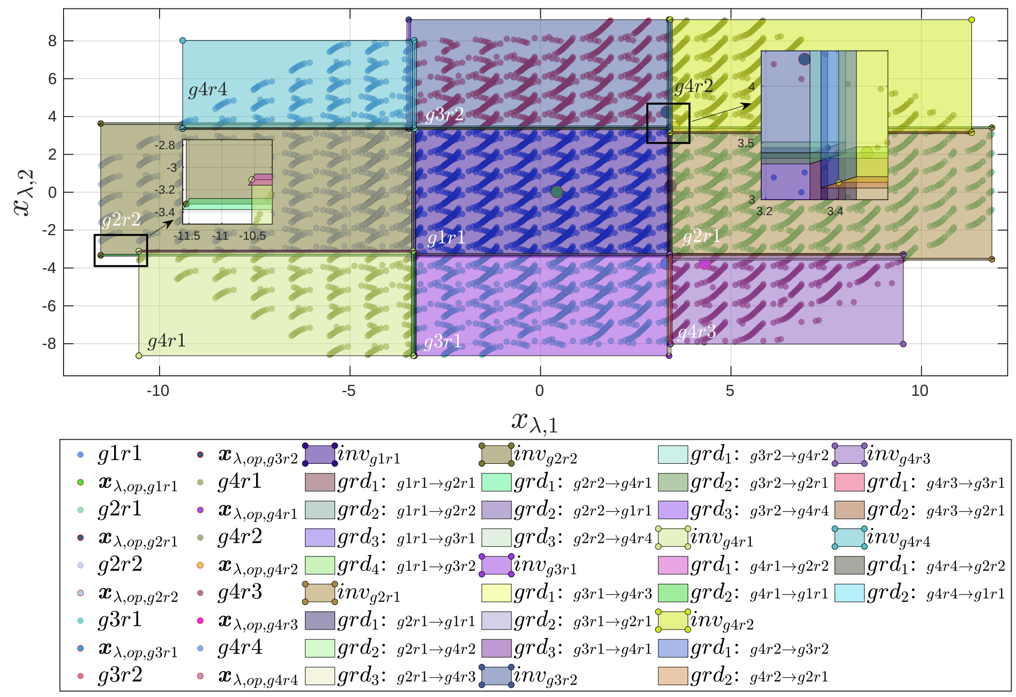


Fig. 7.44. Generated HA in the $\mathcal{S}_\lambda$ space before applying the jump functions. The guards were modeled as polytopes with a specified large thickness (volume) only for demonstration purposes. However, the model used for simulation has considerable slimmer guards.

model and the PI controller, an option file is provided that lists the connections between the elements. For the current example, this file specifies that the first and second input ports of the PI controller, V_{ref} and V_{in}, are from the global input and from the output of the single track model, respectively. The input of the single track model, the steering angle δ, is the output of the PI controller. The output of the PI controller is the voltage at the output (V_{out} in Fig. 7.41), and the output of the single track model (ψ_v from Eq. (7.1)) are from the original state space $\mathcal{S}_o$ of each system. The outputs of both systems are specified using Eq. (5.7). For the PI controller, the output is simply calculated by:

$$\begin{aligned} V_{out} &= \boldsymbol{c}_{tmp,pi}\boldsymbol{x}_{pi} \\ &= \boldsymbol{c}_{loc}(\boldsymbol{x}_{\lambda,pi} - \boldsymbol{x}_{\lambda,op,pi}) + \cancel{\boldsymbol{d}_{loc,pi}(\boldsymbol{u}_{pi} - \boldsymbol{u}_{op,pi})} + k_{loc,pi} \\ &= \boldsymbol{c}_{loc,pi}(\boldsymbol{x}_{\lambda,pi} - \boldsymbol{x}_{\lambda,op,pi}) + V_{out,op} \end{aligned} \tag{7.2}$$

Note that all variables and vector in Eq. (7.2) correspond to the PI controller (pi). Similarly, the output of the single track model (st) is calculated via:

$$\psi_v = \boldsymbol{c}_{loc,st}(\boldsymbol{x}_{\lambda,st} - \boldsymbol{x}_{\lambda,op,st}) + \psi_{v,op}\,, \tag{7.3}$$

which represents the input to the PI controller. Thus, the control loop controls the yaw rate ψ_v according to the reference voltage at V_{ref}.

In 9.17 s, *Elsa* created the CHA. Note that, *spaceM* is integrated in the model call. Thus, internally, *Elsa* first generated the HAs from each sampled data set, followed by the generation of a CHA consisting of these HAs. The model was deployed in the form for the usage with the *parallelHybridAutomaton* class from *Cora*.

With the single track models speed set to 10 km/h, a reachability analysis is performed with a unit step at V_{in}, the input of the PI controller. The result is illustrated in Fig. 7.45. Note that after the solution has been calculated by *Cora*, the internal signals are resolved using the $\boldsymbol{x}_\lambda$ state variables to calculate the inputs of the systems. This is performed by using Eqs. (7.2, 7.3) to compute the output of one system, which represents the input of the next system. Finally, the back-transformation is as usually applied for both systems with the calculated and global inputs, to compute all internal variables of both systems in their original state spaces.

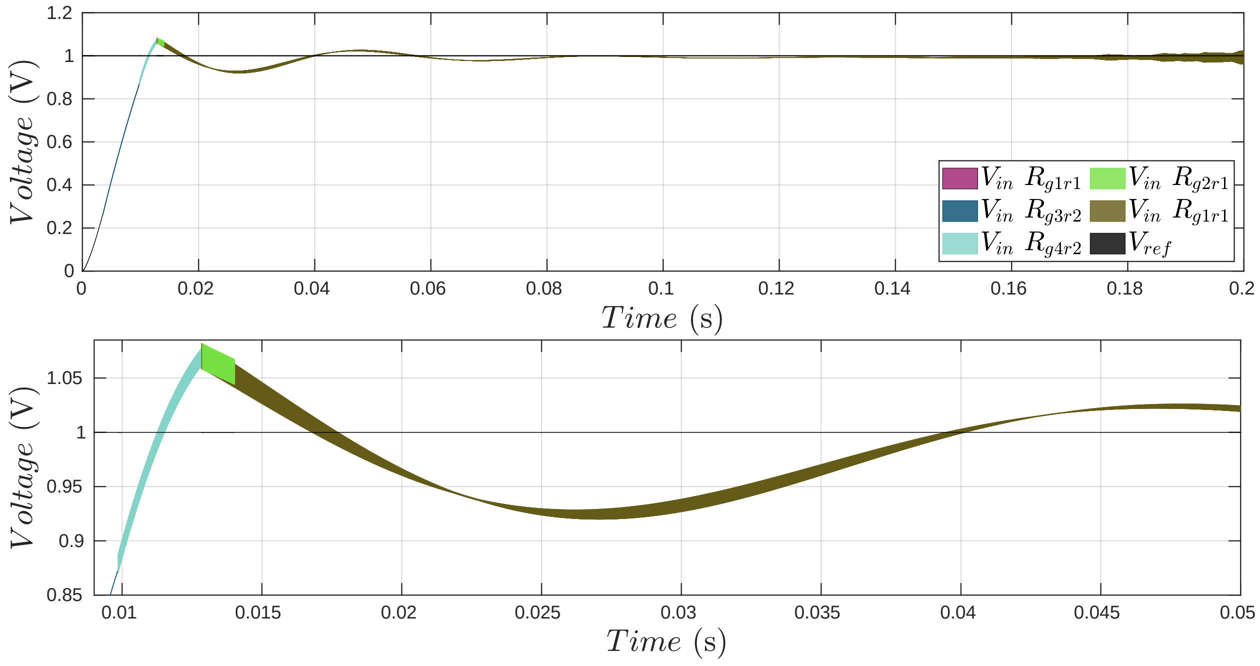


Fig. 7.45. Result of the reachability analysis of the compositional system illustrated upon the inputs of the PI controller. The figure in (a) is enlarged in (b) to show the different dynamic behaviors obtained during the simulation due to the switching of the location of the PI controller. Note that V_{ref} steps from 0 to 1 V at t=0 s.

As shown in Fig. 7.45, the PI controller undergoes 4 transitions, thereby changing 5 times the dynamic behavior. Starting from the unsaturated state ($g1r1$), first the P-part ($g3r2$) then both parts ($g4r2$) go into saturation. After that, the P-part reaches again a linear behavior, while the I-part is still in the saturation mode ($g2r1$). Finally, both parts leave saturation ($g1r1$).

The various examples handled in this chapter and the obtained results show how the introduced abstraction methodology could be used to abstract simple up till mid range circuits easily with an acceptable accuracy. Depending on the settings used, the accuracy of the model as well as the speed-up obtained can vary. For large circuits, a compositional approach could be used that first models the underlying circuits, followed by generating a compositional model from the obtained HAs, thus giving this approach the capability of scalability while preserving an acceptable accuracy.

8 Conclusion and Future Directions

8.1 Conclusion

In this dissertation, a fully automated abstraction approach was proposed that abstracts a Spice netlist to a hybrid automaton (HA) with a finite set of locations. These HAs have a reduced order compared to the original netlist. Moreover, they can be used for simulations with a significant speed up factor compared to the netlist with little deviations. On top of that, the generated HAs can be formally verified against the Spice netlist they abstract, closing thereby the modeling loop by models which are correct by construction. By additionally considering the error between the abstract model and the Spice netlist, the generated HA can be used for the formal verification of the circuits using various formal methods such as reachability analysis and run time verification.

As stated, the approach is fully automated. In the behavioral abstraction process, three tools are used: *Vera*, *spaceM*, and *Elsa*. The latter two are the result of this dissertation, while the first one is a household tool partially extended to fit the needs of the abstraction methodology. The approach first samples a netlist via *Vera*. Second, the sampled data points are imported into Matlab by using the developed tool *spaceM*. Finally, the developed tool *Elsa*, which stands for eigenvalue-based hybrid linear system abstraction, is used to model a HA from the sampled data. The resulted HA exhibits a linear behavior in each of its locations characterized mainly by the sampled eigenvalues.

The tools *Elsa* uses various unsupervised machine learning algorithms, such as clustering the eigenvalues and the sampled data to identify the locations of the HA. Moreover, various algorithms examine and label the sampled data, which are used to identify the invariants and guards of the HA in the specified representations. Additional options, like filtering the sampled points or applying jump functions, can be specified which result in different behaviors. Moreover, the number of locations of the HA can be specified as well. In general, increasing the number of locations usually result in more accurate models.

This dissertation systematically builds a map that links the investigated approach to the properties of the constructed HA. Several examples have been covered demonstrating various advantages of the presented methodology, such as: a significant speed-up, a reduced system order, modeling with parameter variation to enclosing abstraction and technology parameter variations, compositional modeling of the system, formally verify the constructed models, use the abstracted models in various verification routines, and automatically obtaining a model at system level from a circuit level description. On the other hand, the abstraction approach is also coupled with acceptable deviations, which occur especially during the transitions of the location of the HA. Several methodologies were introduced that differently handle the transitions of the locations and the corresponding obtained deviations. Moreover, the order reduction should be handled carefully, as a too aggressive order reduction might result in the incapability of capturing the essential behavior used to describe the model accurately.

The presented approach overcomes the boarders between various programming languages. From a Spice netlist, the models generated could be deployed in three various output languages. The main target of this diversity was the focus on the formal methods, that is, for each of the deployed model formats, a verification method can be used. The models generated in Matlab syntax can be used in a reachability analysis performed by *Cora*. The Verilog-A models can be checked for equivalence against the original Spice netlist. The models generated in System-AMS can be online monitored and thus verified during runtime.

Finally, there are still many difficulties for the formal verification of analog and especially AMS circuits. The stated approach proposed to abstract a netlist by a HA and use it in verification routines. However, the abstracted model still exhibits slight deviations from the original netlist. Even though the approach offered to compensate for the errors performed during the abstraction of the circuit by modeling the system with parameter variations, over approximations are usually obtained. Even though these over approximations were acceptable and well bounded, more effort must be spent in examining and optimizing the generated models to obtain even tighter bounded results. Thus, accurate formal verification is still a goal that future work targets to achieve.

8.2 Future Directions

With the presented methodology, small and mid range circuits could be easily abstracted. However, the user still needs some knowledge about the circuit, to provide the correct settings, with the reduction order being the most significant option.

For the presented methodology, there are some aspects which could be improved. Probably the most significant one is linking the sampling tool *Vera* to the abstraction tool *Elsa*. Considering this point from the sampling aspects, *Vera* could interact with the eigenvalue clustering in *Elsa*. Thus, *Elsa* could examine the need to sample more points in specific regions of the state space. This would drastically reduce the sampling time, as only significant locations are sampled precisely. Considering this point from the modeling aspect, the modeling process of *Elsa* might be changed to a one-the-fly one.

Another point that offers room for improvement, is the system description. At the current time, the HAs are described linearly in their locations. This of course brings the capability of easily scaling, especially when a compositional approach is used. However, more accurate system descriptions might be obtained by using slightly nonlinear functions instead.

One of the major goals of this dissertation was removing the borders between the models on system level and those on the circuit level. The presented approach proved its capability of transitioning a circuit level netlist to a system level description with an acceptable accuracy. Considering the opposite direction, and thereby the synthesis of a netlist at circuit level from a system level description, there might be some promising directions in investigating the usage of the presented methods.

Another major goal was to introduce an approach which is scalable. As stated in the previous chapter, this approach utilizes various methods to attain scalability, of which the most promising one in the compositional abstraction. By dividing the circuit into sub-circuits, followed by abstracting these netlist and building a compositional model from the obtained HAs, the introduced approach could be used to handle large circuits, bringing it once again to *divide and conquer*.

Bibliography

[ABB11] V. Acary, O. Bonnefon, and B. Brogliato, *Nonsmooth Modeling and Simulation for Switched Circuits*, eng, ser. Lecture Notes in Electrical Engineering 69. Dordrecht: Springer, 2011, ISBN: 978-90-481-9681-4.

[ABDM00] E. Asarin, O. Bournez, T. Dang, and O. Maler, “Approximate Reachability Analysis of Piecewise-Linear Dynamical Systems,” *HSCC '00: Hybrid Systems: Computation and Control, LNCS*, pp. 76–90, 2000.

[ABKS] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, “OPTICS: Ordering Points To Identify the Clustering Structure,” en, p. 12,

[Ada09] J. Adamy, *Nichtlineare Regelungen*, ger. Berlin: Springer, 2009, ISBN: 978-3-642-00793-4.

[Ada13] J. Adamy, *Systemdynamik und Regelungstechnik II*, ger, 4. Aufl, ser. Berichte aus der Steuerungs- und Regelungstechnik. Aachen: Shaker, 2013, ISBN: 978-3-8440-1827-1.

[AHP96] R. Alur, T. Henzinger, and Pei-Hsin Ho, “Automatic symbolic verification of embedded systems,” *IEEE Transactions on Software Engineering*, vol. 22, no. 3, pp. 181–201, Mar. 1996, ISSN: 1939-3520. DOI: 10.1109/32.489079.

[AIA14] C. W. Antuvan, M. Ison, and P. Artemiadis, “Embedded human control of robots using myoelectric interfaces,” *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 22, no. 4, pp. 820–827, 2014.

[AK12a] M. Althoff and B. Krogh, “Avoiding geometric intersection operations in reachability analysis of hybrid systems,” ser. HSCC'12 - Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control, Apr. 2012, pp. 45–54. DOI: 10.1145/2185632.2185643.

[AK12b] M. Althoff and B. H. Krogh, “Avoiding geometric intersection operations in reachability analysis of hybrid systems,” en, in *Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control - HSCC '12*, Beijing, China: ACM Press, 2012, p. 45, ISBN: 978-1-4503-1220-2. DOI: 10.1145/2185632.2185643.

[Alt15] M. Althoff, “An Introduction to CORA 2015,” *Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems*, 2015.

[ARIS14] H. Anton, C. Rorres, IT Pro - York University, and Skillsoft Books - York University, *Elementary Linear Algebra: Applications Version, 11th Edition*, English. 2014.

[ARK+13] M. Althoff, A. Rajhans, B. H. Krogh, S. Yaldiz, X. Li, and L. Pileggi, “Formal verification of phase-locked loops using reachability analysis and continuization,” *Communications of the ACM*, vol. 56, no. 10, pp. 97–104, 2013.

[ASB07] M. Althoff, O. Stursberg, and M. Buss, “Reachability analysis of linear systems with uncertain parameters and inputs,” in *Proc. of the 46th IEEE Conference on Decision and Control*, 2007, pp. 726–732.

[ASB10a] M. Althoff, O. Stursberg, and M. Buss, “Computing reachable sets of hybrid systems using a combination of zonotopes and polytopes,” en, *Nonlinear Analysis: Hybrid Systems*, vol. 4, no. 2, pp. 233–249, May 2010, ISSN: 1751570X. DOI: 10.1016/j.nahs.2009.03.009.

[ASB10b] M. Althoff, O. Stursberg, and M. Buss, “Computing reachable sets of hybrid systems using a combination of zonotopes and polytopes,” *Nonlinear analysis: hybrid systems*, vol. 4, no. 2, pp. 233–249, 2010.

[AV07] D. Arthur and S. Vassilvitskii, “K-Means++: The advantages of careful seeding,” in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA ’07, USA: Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035, ISBN: 978-0-89871-624-5.

[AZT07] G. Al Sammane, M. H. Zaki, and S. Tahar, “A Symbolic Methodology for the Verification of Analog and Mixed Signal Designs,” in *2007 Design, Automation & Test in Europe Conference & Exhibition*, Nice, France: IEEE, Apr. 2007, pp. 1–6, ISBN: 978-3-9810801-2-4. DOI: 10.1109/DATE.2007.364599.

[Bai00] Z. Bai, Ed., *Templates for the Solution of Algebraic Eigenvalue Problems*, ser. Software, Environments, Tools. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2000, ISBN: 978-0-89871-471-5.

[Ban14] N. Banagaaya, *Index-Aware Model Order Reduction Methods for DAEs*, English. Technische Universiteit Eindhoven, 2014, ISBN: 978-90-386-3700-6.

[BAS16] N. Banagaaya, G. Alì, and W. H. A. Schilders, *Index-Aware Model Order Reduction Methods: Applications to Differential-Algebraic Equations*, eng, ser. Atlantis Studies in Scientific Computing in Electromagnetics 2. Paris: Atlantis Press, 2016, ISBN: 978-94-6239-188-8.

[BCE+10] M. Barnasconi, G. Christoph, K. Einwich, M. Damm, M.-M. Louëra, T. Maehne, F. Pecheux, and A. Vachoux, *Standard SystemC AMS Language Reference Manual*, 2010.

[BDH96] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, “The Quickhull algorithm for convex hulls,” *ACM TRANSACTIONS ON MATHEMATICAL SOFTWARE*, vol. 22, no. 4, pp. 469–483, 1996.

[BFG+16] E. Barke, A. Fürtig, G. Gläser, C. Grimm, L. Hedrich, S. Heinen, E. Hennig, H.-S. L. Lee, W. Nebel, G. Nitsche, M. Olbrich, C. Radojicic, and F. Speicher, “Embedded tutorial: Analog-/mixed-signal verification methods for AMS coverage analysis,” in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2016, pp. 1102–1111.

[BGG+09] E. Barke, D. Grabowski, H. Graeb, L. Hedrich, S. Heinen, R. Popp, S. Steinhorst, and Y. Wang, “Formal approaches to analog circuit verification,” in *Proc. DATE ’09. Design, Automation & Test in Europe Conference & Exhibition*, Apr. 2009, pp. 724–729.

[BHA95] A. Balivada, Y. Hoskote, and J. Abraham, “Verification of transient response of linear analog circuits,” in *VLSI Test Symposium, 1995. Proceedings., 13th IEEE*, Apr. 1995, pp. 42–47. DOI: 10.1109/VTEST.1995.512615.

[Bor98] C. Borchers, "Symbolic Behavioral Model Generation of Nonlinear Analog Circuits," *IEEE Transactions on Circuits and Systems II: Analog & Digital Signal Processing*, vol. 45, no. 10, pp. 1362–1371, 1998.

[CK03] A. Chutinan and B. Krogh, "Computational Techniques for Hybrid System Verification," *IEEE Transactions on Automatic Control*, vol. 48, no. 1, pp. 64–75, 2003.

[CWL+15] J.-Y. Chen, S.-W. Wang, C.-H. Lin, C.-N. Liu, Y.-J. Lin, M.-J. Lee, Y.-L. Luo, and S.-Y. Kao, "Automatic behavioral model generator for mixed-signal circuits based on structure recognition and auto-calibration," in *2015 International SoC Design Conference (ISOCC)*, Gyungju, South Korea: IEEE, Nov. 2015, pp. 3–4, ISBN: 978-1-4673-9308-9. DOI: 10.1109/ISOCC.2015.7401683.

[Dav03] A. Davis, "An overview of algorithms in Gnucap," in *University/Government/Industry Microelectronics Symp.*, 2003, pp. 360–361.

[DC05] T. R. Dastidar and P. Chakrabarti, "A verification system for transient response of analog circuits using model checking," in *VLSI Design, 2005. 18th International Conference On*, IEEE, 2005, pp. 195–200.

[DDM04] T. Dang, A. Donzé, and O. Maler, "Verification of analog and mixed-signal circuits using hybrid system techniques," in *Formal Methods in Computer-Aided Design*, Springer, 2004, pp. 21–36.

[Doo79] P. V. Dooren, "The Computation of Kronecker's Canonical Form of a Singular Pencil," *Journal on Linear Algebra and its Applications*, vol. 27, pp. 103–140, 1979.

[Dra20] C. Draudt, "CTL-A Monitor Generator für SystemC-AMS mit affinen Formen," B.S. Thesis, Institute of Computer Science, University of Frankfurt, 2020.

[dS04] L. H. de Figueiredo and J. Stolfi, "Affine Arithmetic: Concepts and Applications," *Numerical Algorithms*, vol. 37, no. 1, pp. 147–158, Dec. 2004, ISSN: 1572-9265. DOI: 10.1023/B:NUMA.0000049462.70970.b6.

[EKX] M. Ester, H.-P. Kriegel, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," en, p. 6,

[Est00] D. Estévez-Schwarz, "Consistent Initialization for Index-2 Differential Algebraic Equations and its Application to Circuit Simulation," *Dissertation, Humboldt-Universität Berlin*, 2000.

[FGG+17] A. Fürtig, G. Gläser, C. Grimm, L. Hedrich, S. Heinen, H.-S. L. Lee, G. Nitsche, M. Olbrich, C. Radojicic, and F. Speicher, "Novel metrics for Analog Mixed-Signal coverage," in *Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2017 IEEE 20th International Symposium On*, IEEE, 2017, pp. 97–102.

[FH18] A. Fürtig and L. Hedrich, "Formal Techniques for Verification and Coverage Analysis of Analog Systems," in *Formal System Verification*, Springer, 2018, pp. 1–35.

[FHS+07] M. Fränzle, H. Hungar, C. Schmitt, B. Wirtz, B. Becker, W. Damm, M. Fränzle, E.-R. Olderog, A. Podelski, and R. Wilhelm, "HLang: Compositional Representation of Hybrid Systems via Predicates," SFB/TR 14 AVACS, Reports of SFB/TR 14 AVACS 20, Jul. 2007.

[FKR06] G. Frehse, B. H. Krogh, and R. A. Rutenbar, "Verifying Analog Oscillator Circuits Using Forward/Backward Abstraction Refinement," in *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*, ser. DATE '06, 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 257–262, ISBN: 3-9810801-0-6.

[FLD+11] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable verification of hybrid systems," in *Computer Aided Verification*, Springer, 2011, pp. 379–395.

[GBR04] S. Gupta, K. B.H., and R. Rutenbar, "Towards Formal Verification of Analog Designs," *ICCAD '04: International Conference on Computer Aided Design*, 2004.

[GGKF18] G. Gläser, M. Grabmann, G. Kropp, and A. Fürtig, "There is a limit to everything: Automating AMS operating condition check generation on system-level," *Integration*, vol. 63, pp. 383–391, Sep. 2018. DOI: 10.1016/j.vlsi.2018.02.016.

[Gie05] G. Gielen, "CAD tools for embedded analogue circuits in mixed-signal integrated systems on chip," *Computers and Digital Techniques, IEE Proceedings -*, vol. 152, no. 3, pp. 317–332, May 2005. DOI: 10.1049/ip-cdt:20045116.

[GPHB05] D. Grabowski, D. Platte, L. Hedrich, and E. Barke, "Time Constrained Verification of Analog Circuits using Model-Checking Algorithms," *ENCTS: Workshop on Formal Verification of Analog Circuits*, 2005.

[GT07] Z. J. D. Ghiath Al Sammane Mohamed H. Zaki and S. Tahar, "Towards Assertion Based Verification of Analog and Mixed Signal Designs Using PSL," *Forum on Design Languages (FDL)*, 2007.

[GXGM19] G. Gielen, N. Xama, K. Ganesan, and S. Mitra, "Review of methodologies for pre- and post-silicon analog verification in mixed-signal SOCs," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 1006–1009.

[HB23] L. Hedrich and E. Barke, "A formal approach to verification of linear analog circuits with parameter tolerances," in *Proc. Design, Automation and Test in Europe*, 23, pp. 649–654. DOI: 10.1109/DATE.1998.655927.

[HB5] L. Hedrich and E. Barke, "A formal approach to nonlinear analog circuit verification," in *Proc. IEEE/ACM International Conference on Computer-Aided Design ICCAD-95. Digest of Technical Papers*, 5, pp. 123–127. DOI: 10.1109/ICCAD.1995.480002.

[Hen00] E. Hennig, "Symbolic Approximation and Modeling Techniques for Analysis and Design of Analog Circuits," *Dissertation, Universität Kaiserslautern, Shaker Verlag, Aachen*, 2000.

[HGL04] U. B. Holz, G. H. Golub, and K. H. Law, "A Subspace Approximation Method for the Quadratic Eigenvalue Problem," en, *SIAM Journal on Matrix Analysis and Applications*, vol. 26, no. 2, pp. 498–521, Jan. 2004, ISSN: 0895-4798, 1095-7162. DOI: 10.1137/S0895479803423378.

[HHB02] W. Hartong, L. Hedrich, and E. Barke, "On Discrete Modeling and Model Checking for Nonlinear Analog Systems," *CAV '02: International Conference on Computer-Aided Verification, LNCS*, vol. 2404, pp. 401–413, 2002.

[HKH04] W. Hartong, R. Klausen, and L. Hedrich, "Formal Verification for Nonlinear Analog Systems: Approaches to Model and Equivalence Checking," *Advanced Formal Verification, R. Drechsler, ed., Kluwer Academic Publishers, Boston*, pp. 205–245, 2004.

[HKJM13] M. Herceg, M. Kvasnica, C. Jones, and M. Morari, “Multi-parametric toolbox 3.0,” in *Proc. of the European Control Conference*, Zürich, Switzerland, Jul. 2013, pp. 502–510.

[HRB75] C. Ho, A. Ruehli, and P. Brennan, “The Modified Nodal Approach to Network Analysis,” *IEEE Transactions on Circuits and Systems*, vol. 22, no. 6, pp. 504–509, 1975.

[IR17] A. Ilchmann and T. Reis, Eds., *Surveys in Differential-Algebraic Equations. 4: ...* eng, ser. Differential-Algebraic Equations Forum. Cham: Springer, 2017, ISBN: 978-3-319-46618-7.

[JKN10] K. D. Jones, V. Konrad, and D. Ničković, “Analog property checkers: A DDR2 case study,” en, *Formal Methods in System Design*, vol. 36, no. 2, pp. 114–130, Jun. 2010, ISSN: 0925-9856, 1572-8102. DOI: 10.1007/s10703-009-0085-x.

[KHD17] X. Kong, C. Hu, and Z. Duan, *Principal Component Analysis Networks and Algorithms*, en. Springer, Jan. 2017, ISBN: 978-981-10-2915-8.

[KTR+20] N. Kochdumper, A. Tarraf, M. Rechmal, M. Olbrich, L. Hedrich, and M. Althoff, “Establishing Reachset Conformance for the Formal Analysis of Analog Circuits,” in *ASP-DAC*, 2020.

[LAH+15] H. S. L. Lee, M. Althoff, S. Hoelldampf, M. Olbrich, and E. Barke, “Automated generation of hybrid system models for reachability analysis of nonlinear analog circuits,” in *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, Jan. 2015, pp. 725–730. DOI: 10.1109/ASPDAC.2015.7059096.

[LF09] R. Larson and D. C. Falvo, *Elementary Linear Algebra*, English. Boston: Houghton Mifflin Harcourt Pub. Co., 2009, ISBN: 978-0-547-00481-5.

[Llo82] S. Lloyd, “Least squares quantization in PCM,” *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, Mar. 1982, ISSN: 1557-9654. DOI: 10.1109/TIT.1982.1056489.

[LMH02] L. Lemaitre, C. McAndrew, and S. Hamm, “ADMS-automatic device model synthesizer,” in *Custom Integrated Circuits Conference, 2002. Proceedings of the IEEE 2002*, IEEE, 2002, pp. 27–30.

[Lun10] J. Lunze, *Regelungstechnik. 2: Mehrgrößensysteme, digitale Regelung: mit 55 Beispielen, 101 Übungsaufgaben sowie einer Einführung in das Programmsystem MATLAB*, ger, 6., neu bearb. Aufl, ser. Springer-Lehrbuch. Berlin: Springer, 2010, ISBN: 978-3-642-10197-7.

[Mär91] R. März, “Numerical Methods for Differential Algebraic Equations,” *Acta Numerica*, pp. 141–198, 1991.

[Mey08] C. D. Meyer, *Matrix Analysis and Applied Linear Algebra*, en. Philadelphia: Society for Industrial and Applied Mathematics, 2008, ISBN: 978-0-89871-454-8.

[MN04] O. Maler and D. Nickovic, “Monitoring Temporal Properties of Continuous Signals,” in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques i*, ser. Lecture Notes in Computer Science, vol. 3253, Springer, 2004, pp. 152–166.

[MN13] O. Maler and D. Ničković, “Monitoring properties of analog and mixed-signal circuits,” en, *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 3, pp. 247–268, Jun. 2013, ISSN: 1433-2779, 1433-2787. DOI: 10.1007/s10009-012-0247-9.

[MPDG09] R. Mukhopadhyay, S. K. Panda, P. Dasgupta, and J. Gough, "Instrumenting AMS assertion verification on commercial platforms," en, *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 2, pp. 1–47, Mar. 2009, ISSN: 1084-4309, 1557-7309. DOI: 10.1145/1497561.1497564.

[NM07] D. Nickovic and O. Maler, "AMT: A Property-Based Monitoring Tool for Analog Systems," English, in *Formal Modeling and Analysis of Timed Systems*, ser. Lecture Notes in Computer Science, J.-F. Raskin and P. Thiagarajan, Eds., vol. 4763, Springer Berlin Heidelberg, 2007, pp. 304–319, ISBN: 978-3-540-75453-4. DOI: 10.1007/978-3-540-75454-1_22.

[PAOS03] J. Phillips, J. Afonso, A. Oliveira, and L. M. Silveira, "Analog macromodeling using kernel methods," in *Proceedings of the 2003 IEEE/ACM International Conference on Computer-Aided Design*, IEEE Computer Society, 2003, p. 446.

[Pip19] R. Piper, "Formale Abstraktion und Verifikation von Analog mixed signal Schaltungen in SystemC AMS," M.S. Thesis, Institute of Computer Science, University of Frankfurt, 2019.

[Rad16] C. Radojicic, "Symbolic Simulation of Mixed-Signal Systems with Extended Affine Arithmetic," Doctoral Thesis, Technische Universität Kaiserslautern, 2016.

[RG] C. Radojicic and C. Grimm, *Affine Arithmetic Decition Diagrams (AADD).*

[RG16] C. Radojicic and C. Grimm, "Formal verification of mixed-signal designs using extended affine arithmetic," in *2016 12th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*, Jun. 2016, pp. 1–4. DOI: 10.1109/PRIME.2016.7519482.

[RGJR17] C. Radojicic, C. Grimm, A. Jantsch, and M. Rathmair, "Towards verification of uncertain cyber-physical systems," in *Proceedings 3rd International Workshop on Symbolic and Numerical Methods for Reachability Analysis, SNR@ETAPS 2017, Uppsala, Sweden, 22nd April 2017*, E. Ábrahám and S. Bogomolov, Eds., ser. EPTCS, vol. 247, 2017, pp. 1–17. DOI: 10.4204/EPTCS.247.1.

[Ria08] R. Riaza, *Differential-Algebraic Systems: Analytical Aspects and Circuit Applications*, eng. New Jersey, NJ: World Scientific, 2008, ISBN: 978-981-279-180-1.

[RSRG12] C. Radojicic, F. Schupfer, M. Rathmair, and C. Grimm, "Assertion-based verification of signal processing systems with affine arithmetic," in *Proceeding of the 2012 Forum on Specification and Design Languages*, 2012, pp. 20–26.

[SA01] S. Seshadri and J. A. Abraham, "Frequency response verification of analog circuits using global optimization techniques," *Journal of Electronic Testing*, vol. 17, no. 5, pp. 395–408, 2001.

[Sal02] A. Salem, "Semi-formal verification of VHDL-AMS descriptions," *ISCAS '02: IEEE International Symposium on Circuits and Systems*, vol. 5, pp. 333–336, 2002.

[SH08] S. Steinhorst and L. Hedrich, "Model Checking of Analog Systems using an Analog Specification Language," in *Proc. Design, Automation and Test in Europe DATE '08*, Mar. 2008, pp. 324–329. DOI: 10.1109/DATE.2008.4484700.

[SH10a] S. Steinhorst and L. Hedrich, “Advanced methods for equivalence checking of analog circuits with strong nonlinearities,” *Formal Methods in System Design*, vol. 36, no. 2, pp. 131–147, 2010.

[SH10b] S. Steinhorst and L. Hedrich, “Advanced methods for equivalence checking of analog circuits with strong nonlinearities,” en, *Formal Methods in System Design*, vol. 36, no. 2, pp. 131–147, Jun. 2010, ISSN: 0925-9856, 1572-8102. DOI: 10.1007/s10703-009-0086-9.

[SH12a] S. Steinhorst and L. Hedrich, “Equivalence checking of nonlinear analog circuits for hierarchical AMS System Verification,” in *VLSI and System-on-Chip (VLSI-SoC), 2012 IEEE/IFIP 20th International Conference On*, IEEE, 2012, pp. 135–140.

[SH12b] S. Steinhorst and L. Hedrich, “Trajectory-directed discrete state space modeling for formal verification of nonlinear analog circuits,” in *Proceedings of the International Conference on Computer-Aided Design*, ACM, 2012, pp. 202–209.

[SHB18] D. Schramm, M. Hiller, and R. Bardini, *Modellbildung und Simulation der Dynamik von Kraftfahrzeugen*, ger, 3., aktualisierte und ergänzte Auflage. Berlin: Springer Vieweg, 2018, ISBN: 978-3-662-54480-8.

[SK03] O. Stursberg and B. H. Krogh, “Efficient representation and computation of reachable sets for hybrid systems,” in *Hybrid Systems: Computation and Control*, O. Maler and A. Pnueli, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 482–497, ISBN: 978-3-540-36580-8.

[SL10] A. Singh and P. Li, “On behavioral model equivalence checking for large analog/mixed signal systems,” in *Proceedings of the International Conference on Computer-Aided Design*, IEEE Press, 2010, pp. 55–61.

[SWL+17] L.-Y. Song, C. Wang, C.-N. J. Liu, Y.-J. Lin, M.-J. Lee, Y.-L. Lo, and S.-Y. Kao, “Non-regression approach for the behavioral model generator in mixed-signal system verification,” in *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Abu Dhabi: IEEE, Oct. 2017, pp. 1–5, ISBN: 978-1-5386-2880-5. DOI: 10.1109/VLSI-SoC.2017.8203462.

[TH19a] A. Tarraf and L. Hedrich, “Automatic Modeling of Transistor Level Circuits by Hybrid Systems with Parameter Variable Matrices,” in *,International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, Lausanne, Switzerland, 2019.

[TH19b] A. Tarraf and L. Hedrich, “Behavioral Modeling of Transistor-Level Circuits using Automatic Abstraction to Hybrid Automata,” in *Design, Automation and Test in Europe*, Florence, 2019.

[TH20] A. Tarraf and L. Hedrich, “Modeling Circuits with Parameter Variation by ELSA: Eigenvalue Based Linear Hybrid System Abstraction,” in *17. GMM/ITG-Fachtagung ANALOG*, 2020.

[TKR+20] A. Tarraf, N. Kochdumper, M. Rechmal, L. Hedrich, and M. Olbrich, “Equivalence Checking Methods for Analog Circuits Using Continuous Reachable Sets,” in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Limassol, Cyprus, 2020.

[TM01] F. Tisseur and K. Meerbergen, “The Quadratic Eigenvalue Problem,” en, *SIAM Review*, vol. 43, no. 2, pp. 235–286, Jan. 2001, ISSN: 0036-1445, 1095-7200. DOI: 10.1137/S0036144500381988.

[TR05] P. Thati and G. Roşu, “Monitoring Algorithms for Metric Temporal Logic Specifications,” en, *Electronic Notes in Theoretical Computer Science*, vol. 113, pp. 145–162, Jan. 2005, ISSN: 15710661. DOI: 10.1016/j.entcs.2004.01.029.

[TSKK19] P.-N. Tan, M. Steinbach, A. Karpatne, and V. Kumar, *Introduction to Data Mining*, Second edition. NY NY: Pearson, 2019, ISBN: 978-0-13-312890-1.

[Tsu03] C.-C. Tsui, *Robust Control System Design: Advanced State Space Techniques*, en. CRC Press, Dec. 2003, ISBN: 978-0-8247-5881-3.

[Wag17] J. Wagner, “Synthese von Verilog-A Monitoren aus Spezifikationen,” M.S. Thesis, Institute of Computer Science, University of Frankfurt, 2017.

[WAN+09a] Z. Wang, N. Abbasi, R. Narayanan, M. H. Zaki, G. Al Sammane, and S. Tahar, “Verification of analog and mixed signal designs using online monitoring,” in *2009 IEEE 15th International Mixed-Signals, Sensors, and Systems Test Workshop*, Scottsdale, AZ, USA: IEEE, Jun. 2009, pp. 1–8, ISBN: 978-1-4244-4618-6. DOI: 10.1109/IMS3TW.2009.5158695.

[WAN+09b] Z. Wang, N. Abbasi, R. Narayanan, M. H. Zaki, G. Al Sammane, and S. Tahar, “Verification of analog and mixed signal designs using online monitoring,” in *2009 IEEE 15th International Mixed-Signals, Sensors, and Systems Test Workshop*, Scottsdale, AZ, USA: IEEE, Jun. 2009, pp. 1–8, ISBN: 978-1-4244-4618-6. DOI: 10.1109/IMS3TW.2009.5158695.

[WKZC11] Y.-C. Wang, A. Komuravelli, P. Zuliani, and E. M. Clarke, “Analog circuit verification by statistical model checking,” en, in *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*, Yokohama, Japan: IEEE, Jan. 2011, pp. 1–6, ISBN: 978-1-4244-7515-5. DOI: 10.1109/ASPDAC.2011.5722168.

[WLL+19] Q. Wang, S. Ling, X. Liang, H. Wang, H. Lu, and Y. Zhang, “Self-Healable Multifunctional Electronic Tattoos Based on Silk and Graphene,” en, *Advanced Functional Materials*, vol. 29, no. 16, p. 1 808 695, Apr. 2019, ISSN: 1616-301X, 1616-3028. DOI: 10.1002/adfm.201808695.

[WLM+08] D. Walter, S. Little, C. Myers, N. Seegmiller, and T. Yoneda, “Verification of Analog/Mixed-Signal Circuits Using Symbolic Methods,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 12, pp. 2223–2235, Dec. 2008, ISSN: 0278-0070. DOI: 10.1109/TCAD.2008.2006159.

[YDL12] L. Yin, Y. Deng, and P. Li, “Verifying Dynamic Properties of Nonlinear Mixed-signal Circuits via Efficient SMT-based Techniques,” in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD ’12, New York, NY, USA: ACM, 2012, pp. 436–442, ISBN: 978-1-4503-1573-9. DOI: 10.1145/2429384.2429474.

[Yiz95] Yizong Cheng, “Mean shift, mode seeking, and clustering,” en, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 8, pp. 790–799, 1995, ISSN: 01628828. DOI: 10.1109/34.400568.

[ZATB07] M. Zaki, G. Al-Sammane, S. Tahar, and G. Bois, “Combining Symbolic Simulation and Interval Arithmetic for the Verification of AMS Designs,” *FMCAD*, 2007.

[ZFHM05] W. Zheng, Y. Feng, X. Huang, and H. Mantooth, “Ascend: Automatic bottom-up behavioral modeling tool for analog circuits,” in *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium On*, May 2005, 5186–5189 Vol. 5. DOI: 10.1109/ISCAS.2005.1465803.

[ZG19] C. Zivkovic and C. Grimm, “Nubolic simulation of AMS systems with data flow and discrete event models,” in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 1457–1462.

[ZGO+19] C. Zivkovic, C. Grimm, M. Olbrich, O. Scharf, and E. Barke, “Hierarchical Verification of AMS Systems With Affine Arithmetic Decision Diagrams,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 10, pp. 1785–1798, Oct. 2019, ISSN: 0278-0070, 1937-4151. DOI: 10.1109/TCAD.2018.2864238.

[ZTB06] M. H. Zaki, S. Tahar, and G. Bois, “A Practical Approach for Monitoring Analog Circuits,” in *Proceedings of the 16th ACM Great Lakes Symposium on VLSI*, ser. GLSVLSI ’06, New York, NY, USA: ACM, 2006, pp. 330–335, ISBN: 1-59593-347-6. DOI: 10.1145/1127908.1127984.

[ZTB08] M. H. Zaki, S. Tahar, and G. Bois, “Formal verification of analog and mixed signal designs: A survey,” *Microelectronics Journal*, vol. 39, no. 12, pp. 1395–1404, 2008.

Appendices

A Additional Descriptions

A.1 The Index of Nilpotency

Theorem A.1.1. *A square matrix $\boldsymbol{N}$, is a nilpotent matrix of index η, if:*

$$\boldsymbol{N}^{\eta} = \boldsymbol{0} \tag{A.1}$$

η is called the index of nilpotency. Nilpotent matrices have several special properties. In this dissertation, we are only interested in the following property:

$$\begin{aligned}(\boldsymbol{I}+\boldsymbol{N})^{-1} &= \sum_{i=0}^{\eta-1}(-\boldsymbol{N})^{i} \\ &= \boldsymbol{I}-\boldsymbol{N}+\boldsymbol{N}^{2}+\ldots+(-1)^{\eta-1}\boldsymbol{N}^{\eta-1}\end{aligned} \tag{A.2}$$

For the case that Assumption 2 is not made, Eq. (3.21) becomes:

$$\begin{bmatrix}\boldsymbol{I}_{\Lambda,red} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{N}_{red}\end{bmatrix}\Delta\dot{\boldsymbol{x}}_s = \begin{bmatrix}\boldsymbol{\Lambda} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{I}_{\infty,red}\end{bmatrix}\Delta\boldsymbol{x}_s + \begin{bmatrix}\tilde{\boldsymbol{B}}_{\Lambda,red}\\ \tilde{\boldsymbol{B}}_{\infty,red}\end{bmatrix}\Delta\boldsymbol{u} \tag{A.3a}$$

$$\Delta\boldsymbol{y} = \begin{bmatrix}\tilde{\boldsymbol{C}}_{\Lambda,red} & \tilde{\boldsymbol{C}}_{\infty,red}\end{bmatrix}\Delta\boldsymbol{x}_s \tag{A.3b}$$

By that, Eq. (A.3a) is divided into two parts similarly to Eq. (3.21).
Compared to Eq. (3.21), the unchanged part:

$$I_{\Lambda,red}\Delta\dot{\boldsymbol{x}}_{\lambda} = \boldsymbol{\Lambda}\Delta\boldsymbol{x}_{\lambda} + \tilde{\boldsymbol{B}}_{\Lambda,red}\Delta\boldsymbol{u} \tag{A.4}$$

The algebraic part from Eq. (3.21) becomes as well a dynamic part:

$$\boldsymbol{N}_{red}\Delta\dot{\boldsymbol{x}}_{\infty} = \boldsymbol{I}_{\infty,red}\Delta\boldsymbol{x}_{\infty} + \tilde{\boldsymbol{B}}_{\infty,red}\Delta\boldsymbol{u} \tag{A.5}$$

Eq. (A.5) can be derived over time and multiplied from the right by the nilpotent matrix $\boldsymbol{N}_{red}$ to yield:

$$\begin{aligned}\xRightarrow{\boldsymbol{N}_{red}\frac{d}{dt}} \boldsymbol{N}_{red}^{2}\Delta\ddot{\boldsymbol{x}}_{\infty} &= \boldsymbol{N}_{red}\boldsymbol{I}_{\infty,red}\Delta\dot{\boldsymbol{x}}_{\infty} + \boldsymbol{N}_{red}\tilde{\boldsymbol{B}}_{\infty,red}\Delta\dot{\boldsymbol{u}} \\ &= \boldsymbol{I}_{\infty,red}\Delta\boldsymbol{x}_{\infty} + \tilde{\boldsymbol{B}}_{\infty,red}\Delta\boldsymbol{u} + \boldsymbol{N}_{red}\tilde{\boldsymbol{B}}_{\infty,red}\Delta\dot{\boldsymbol{u}}\end{aligned} \tag{A.6}$$

Note that $\tilde{\boldsymbol{B}}_{\infty,red}$ is considered invariant over time. With η representing the index of nilpotency this process can be repeated η-times to yield:

$$\boldsymbol{N}_{red}^{\eta}\Delta\boldsymbol{x}_{\infty}^{(\eta)} = \Delta\boldsymbol{x}_{\infty} + \sum_{i=0}^{\eta-1}\boldsymbol{N}_{red}^{i}\tilde{\boldsymbol{B}}_{\infty,red}\Delta\boldsymbol{u}^{(\boldsymbol{i})} \tag{A.7}$$

The left side of Eq. (A.7) is obviously equal to zero (see Eq. (A.1)). Thus, $\boldsymbol{x}_\infty$ is uniquely determined by:

$$\Delta \boldsymbol{x}_\infty = -\sum_{i=0}^{\eta-1} \boldsymbol{N}_{red}^{i} \tilde{\boldsymbol{B}}_{\infty,red} \Delta \boldsymbol{u}^{(\boldsymbol{i})} \tag{A.8}$$

Notice the difference between Eq. (A.2) and Eq. (A.8).

The remaining algorithm remains unchanged. Only the back-transformation in *Elsa* is affected by this change. For that, first consider Eq. (3.16) which states the relationship between the $\mathcal{S}_o$ and $\mathcal{S}_s$ space. In Eq. (3.27), this equation was extended to yield a relationship between $\mathcal{S}_o$, $\mathcal{S}_\lambda$ and $\mathcal{S}_\infty$ which represents the origin of the back-transformation. Instead of changing Eq. (3.27) to Eq. (4.9), Eq. (3.27) is changed to:

$$\begin{aligned} \Delta \boldsymbol{x} &= \boldsymbol{F}_\lambda \Delta \boldsymbol{x}_\lambda + \boldsymbol{F}_\infty \Delta x_\infty \\ &= \boldsymbol{F}_\lambda \Delta \boldsymbol{x}_\lambda - \boldsymbol{F}_\infty \sum_{i=0}^{\eta-1} \boldsymbol{N}_{red}^{i} \tilde{\boldsymbol{B}}_{\infty,red} \Delta \boldsymbol{u}^{(\boldsymbol{i})} \end{aligned} \tag{A.9}$$

By replacing Eq. (4.9) by Eq. (A.9), the approach stated in Chapter 4 can be used.

A.2 Similarity Transformations With Complex Eigenvalues

Sometimes it is desired to describe a system without complex values. Consider a diagonal matrix $\boldsymbol{A}$ with the complex eigenvalue $\lambda_1 = \sigma_1 + j\omega_1$ and its conjugate $\lambda_2 = \sigma_1 - j\omega_1$ on its diagonals:

$$\boldsymbol{A} = \begin{bmatrix} \sigma_1 + j\omega_1 & 0 \\ 0 & \sigma_1 - j\omega_1 \end{bmatrix} \tag{A.10}$$

Using the transformation matrix $\boldsymbol{T}$ and its inverse $\boldsymbol{T}^{-1}$:

$$\boldsymbol{T} = \begin{bmatrix} 1 & -j \\ -1 & -j \end{bmatrix} \qquad \boldsymbol{T}^{-1} = \begin{bmatrix} \frac{1}{2} & -\frac{1}{2} \\ \frac{j}{2} & \frac{j}{2} \end{bmatrix} \tag{A.11}$$

$\boldsymbol{A}$ can be transformed to a matrix with only real values:

$$\boldsymbol{T}^{-1}\boldsymbol{A}\boldsymbol{T} = \begin{bmatrix} \sigma_1 & \omega_1 \\ -\omega_1 & \sigma_1 \end{bmatrix} \tag{A.12}$$

B Core of SpaceM

```
// Author: Ahmad Tarraf
// Date: 28.01.2019
// last review 03.06.2020

//syntax spaceM: [G,El,Er,C,D,Dim,INames,DC,Reach,ReachSlew,inputMatrix,outMatrix
    ,Odim,Idim,Xdim,Xsdim,Boxid]=spaceM('rcc.gc.acv.Test');
//or call space=spaceM('rcc.gc.acv'); for structure mode
#include <iostream>
#include <stdlib.h>
#include "statespace.h"
#include "acv.h"
#include "preprocessor.h"
#include "msl.h"
#include <chrono>

int verbose; /* in msmct global definiert */
double predmindeltat;

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
        //Init
        auto start = chrono::steady_clock::now(); // measure time
        matlabData *spaceP = new matlabData;
        StateSpace *stsp = new StateSpace(0);

        // pass all data as a struc == 1 | or  elementwise == 0
        int strucMode = 0;

        // if only one output is specified, create structure
        if (nlhs == 1)
                strucMode=1;

        int argc = nrhs;
        int preprocess_count = 0;
        int argi = 1;
        char **argv;
        bool full = true;
        bool onlyacv = true;
        bool changed = true;
        verbose = 0;
        predmindeltat = 0;

        // allocate mem:
        argv = (char **)mxCalloc(argc, sizeof(char **));
        argv[0] = (char *)mxCalloc(200, sizeof(char *));

        // pass input arguments
        for (int i = 0; i < argc; i++)
        {
                argv[i] = mxArrayToString(prhs[i]);
        }
        // fprintf(stdout, "Running spaceM on %s \n", argv[0]);
        std::cout <<"Running spaceM on " << argv[0]<< std::endl;
        // core function -> get data
        std::cout<<"Reading file "<<endl;
        try
        { readACVFile(argv[0], stsp, spaceP);}
        catch (std::exception &e)
        {
                std::cerr << "An error occured" << e.what() << '\n';
        }
        std::cout<<"Reading finished "<<endl;
        auto pause1 = chrono::steady_clock::now(); // measure time
        std::cout << "Elapsed time: " << chrono::duration_cast<chrono::seconds>(
            pause1 - start).count() << " sec" <<std::endl;;
        // pass data to matalb
        //*********************

        std::cout<<"Copying data to matlab "<<endl;
        if (strucMode == 0)
        {
                // 1) matricies
                plhs[0] = spaceP->Copyvalue("Graph");
                plhs[1] = spaceP->Copyvalue("Eigvleft");
                plhs[2] = spaceP->Copyvalue("Eigvright");
                plhs[3] = spaceP->Copyvalue("center");
                plhs[4] = spaceP->Copyvalue("d");

                // 2) Names
                plhs[5] = spaceP->CopyStrArray("dimensions");
                plhs[6] = spaceP->CopyStrArray("InputNames");
                //plhs[5]=spaceP->CopyStrArray("XStateNames");
                //plhs[6]=spaceP->CopyStrArray("X_sStateNames");

                //3) vectors
                plhs[7] = spaceP->Copyvalue("DC");
                plhs[8] = spaceP->Copyvalue("Reach");
                plhs[9] = spaceP->Copyvalue("ReachSlew");
                plhs[10] = spaceP->Copyvalue("inputMatrix");
                plhs[11] = spaceP->Copyvalue("outputMatrix");

                // 4)Scaler
                plhs[12] = spaceP->Copyvalue("Outputdim");
                plhs[13] = spaceP->Copyvalue("Inputdim");
                plhs[14] = spaceP->Copyvalue("XStatesdim");
                plhs[15] = spaceP->Copyvalue("X_sStatesdim");
                plhs[16] = spaceP->Copyvalue("BoxId");
                plhs[17] = spaceP->Copyvalue("p2p");
                free(spaceP);
        }
        // pass all at once
        // create Strucutre array:
        else
        {
                mxArray *G, *El, *Er, *C, *D, *Dim, *INames, *DC, *Reach, *
                    ReachSlew, *inputMatrix, *outputMatrix,*Odim,*Idim, *Xdim, *
                    Xsdim, *BoxId, *p2p;
                const char *fieldnames[] = {"Graph", "Eigvleft", "Eigvright",
                 "center", "d", "dimensions", "inputNames", "DC", "Reach",
                 "ReachSlew", "inputMatrix","outputMatrix", "Outputdim",
                 "Inputdim", "XStatesdim", "X_sStatesdim","BoxId","p2p"};

                G = spaceP->Copyvalue("Graph");
                El = spaceP->Copyvalue("Eigvleft");
                Er = spaceP->Copyvalue("Eigvright");
                C = spaceP->Copyvalue("center");
                D = spaceP->Copyvalue("d");
                Dim = spaceP->CopyStrArray("dimensions");
                INames = spaceP->CopyStrArray("InputNames");
                DC = spaceP->Copyvalue("DC");
                Reach = spaceP->Copyvalue("Reach");
                ReachSlew = spaceP->Copyvalue("ReachSlew");
                inputMatrix = spaceP->Copyvalue("inputMatrix");
                outputMatrix = spaceP->Copyvalue("outputMatrix");
                Odim = spaceP->Copyvalue("Outputdim");
                Idim = spaceP->Copyvalue("Inputdim");
                Xdim = spaceP->Copyvalue("XStatesdim");
                Xsdim = spaceP->Copyvalue("X_sStatesdim");
                BoxId = spaceP->Copyvalue("BoxId");
                p2p = spaceP->Copyvalue("p2p");
                std::cout<<"Creating structure "<< endl;
                plhs[0] = mxCreateStructMatrix(1, 1, 18, fieldnames);

                mxSetFieldByNumber(plhs[0], 0, 0,  G);
                mxSetFieldByNumber(plhs[0], 0, 1,  El);
                mxSetFieldByNumber(plhs[0], 0, 2,  Er);
                mxSetFieldByNumber(plhs[0], 0, 3,  C);
                mxSetFieldByNumber(plhs[0], 0, 4,  D);
                mxSetFieldByNumber(plhs[0], 0, 5,  Dim);
                mxSetFieldByNumber(plhs[0], 0, 6,  INames);
                mxSetFieldByNumber(plhs[0], 0, 7,  DC);
                mxSetFieldByNumber(plhs[0], 0, 8,  Reach);
                mxSetFieldByNumber(plhs[0], 0, 9,  ReachSlew);
                mxSetFieldByNumber(plhs[0], 0, 10, inputMatrix);
                mxSetFieldByNumber(plhs[0], 0, 11, outputMatrix);
                mxSetFieldByNumber(plhs[0], 0, 12, Odim);
                mxSetFieldByNumber(plhs[0], 0, 13, Idim);
                mxSetFieldByNumber(plhs[0], 0, 14, Xdim);
                mxSetFieldByNumber(plhs[0], 0, 15, Xsdim);
                mxSetFieldByNumber(plhs[0], 0, 16, BoxId);
                mxSetFieldByNumber(plhs[0], 0, 17, p2p);
                // mxDestroyArray(G);  mxDestroyArray(El);
                // mxDestroyArray(Er); mxDestroyArray(C);
```

```
                    // mxDestroyArray(D);  mxDestroyArray(Dim);
                    // mxDestroyArray(INames); mxDestroyArray(DC);
                    // mxDestroyArray(Reach);  mxDestroyArray(ReachSlew);
                    // mxDestroyArray(inputMatrix); mxDestroyArray(Idim);
                    // mxDestroyArray(Xdim); mxDestroyArray(Xsdim);
                    // mxDestroyArray(BoxId);
                    }

          // finalise
          auto end = chrono::steady_clock::now(); // measure time
          std::cout << "Elapsed time: " << chrono::duration_cast<chrono::seconds>(
              end - start).count() << " sec" <<std::endl;;

          std::cout << "Successful terminated" << endl;
}
```

C Spice Netlist of the Operation Amplifier

The schematic of the operation amplifier used in Section 4.1 is illustrated below:

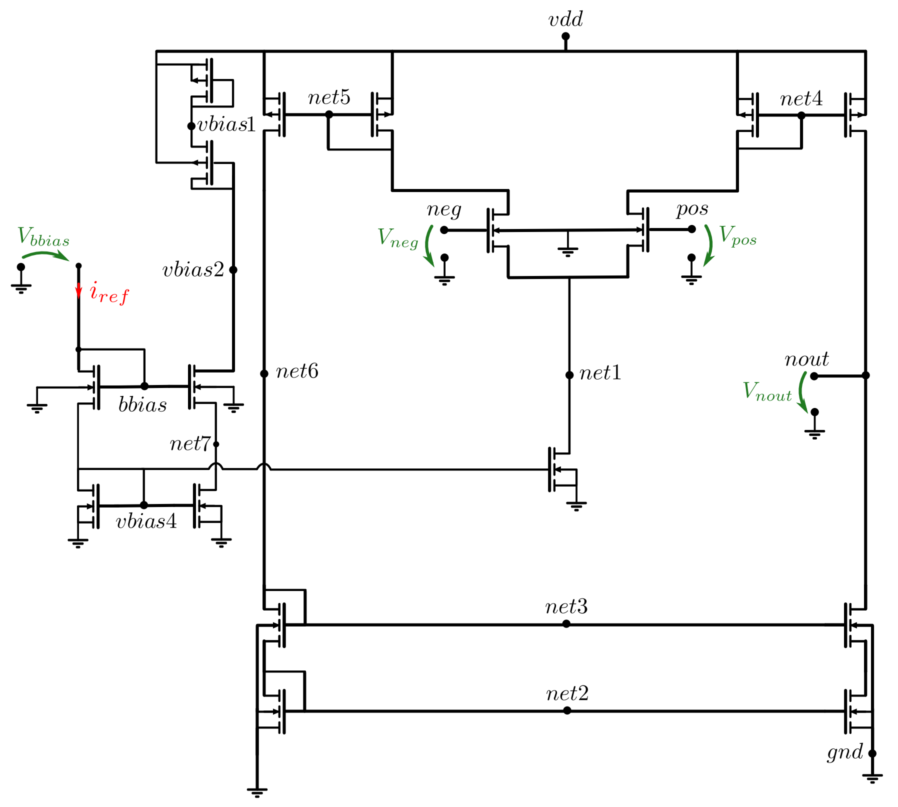


Fig. C.1. Schematic of the operation amplifier used in Fig. 4.2 from Section 4.1.

The corresponding test bench which can be used for *Vera* as well as for *Gnucap* is stated bellow:

Listing C.1: Test bench

```
spice

.param vdd 1.7
* .verilog
* load ./rc.so
* paramset myrc rc; .c=3n; .r=200k; endparamset
* myrc rc1 (nin,nout1,0);
* spice

* for Vera
V1  nin 0 0

R1  nin nin2 100k
C1  nin2 0  0.01u
R2 nin2 neg 1000k
R3 nout neg 1000k
C2 nout neg 0.1u

vvdd  vdd  0     dc='3.3/2'
vvref vref 0     dc='0'
vvdd  0    vss   dc='3.3/2'
iref  vdd  bbias 5u

.include opIN_offset2e6_67MV.gc
* gnd inneg inpos out bias vdd
xI2 vss neg 0 nout bbias vdd opIN_offset2e6_67MV

* .options noccons
*.print tran v(nodes) hidden(0)

** verification cut here **
* replace V1 to see something happen.
*.print dc v(nodes) v(nout)
V1 nin 0 sin(frequency=1, amplitude=18, offset=0)
*V1 nin 0 pulse iv=0 pv=9  rise=200u delay=2000u

.print tran v(nout) V(nin) v(nodes)

.op
.tran .5m 3000m > orginal.gc.acs
.end
```

D Generated Models in Different Output Languages

D.1 Matlab (Cora) Model of the Running Example

For the running example from Section 4.1, several models were created in various programming languages. An example of a Matlab model is given in Listing D.1.

Listing D.1: Abstracted Matlab (*Cora*) model of the running example

```
1           %System @ group 1 region 1
2           A1 = [ .. ..  ;  .. .. ] ;
3           B1 = [ ..  ; .. ] ;
4           Sys = linearSys('linearSys',A1 ,B1);
5           inv = interval([ ..; .. ],[ .. ; .. ]) ;
6           guard1  = halfspace([ .. ; .. ], ..) ;
7           reset.A = eye(2,2); reset.b = [ .. ; .. ] ;
8           tran{1} = transition( guard1, reset,2) ;
9           guard2  = halfspace( [ .. ;.. ], ..) ;
10          reset.A = eye(2,2); reset.b = [ .. ; .. ] ;
11          tran{2} = transition( guard2, reset,3) ;
12          locs{1} =  location('1 -> g1r1',inv,tran,Sys);
128         HA = hybridAutomaton(locs);
185         HA = reach(HA,options);
191         trans.state{1}.F = [ .. ..; ...
365         X_space.X=backtransformation(trans,HA,..);
```

As observed, the system and input matrices are initialized at lines 2 and 3, respectively. On line 5, the invariant is initialized, in this case as an interval hull. Since $g1r1$ has 2 guards, two transitions are defined on lines 8 and 11. Each transition hosts the guards, in this case defined as halfspaces, and their corresponding jump functions which are defined by the *reset* structure. Moreover, the transitions also contain the target location. For example the first transition on line 8 targets $g2r1$ ($locs\{2\}$). Note that the target locations on lines 8 and 11 are specified as doubles. The model abstraction approach makes sure that these numbers are unique and distinct. The name of the location, invariant, transitions, and system description are passed to the location cell array *locs* at line 12. For the reaming two locations, a similar initialization is performed. The cell array *locs* is passed to the hybrid automaton class at line 128. At line 185, a reachability analysis is performed with the previously specified HA in *Cora*. After the analysis finishes, the results are finally transformed at line 365 from the $\mathcal{S}_\lambda$ space to the $\mathcal{S}_o$ space using the transformations matrices and operating points stored in *trans*.

D.2 Verilog-A Model of the Running Example

An example of a Verilog-A model for the running example is given in Listing D.2 using the *grdV method*. The explanation of the model is stated in Section 4.6.2.

Listing D.2: Abstracted Verilog-A model of the running example

```
// Creation options ------------------------
//
// Name of source file: rccTransistors_gc
//------------------------------------------
`include "discipline.h"
`ifdef insideADMS
`define P(x) x
`define INITIAL_MODEL          @(initial_model)
`else
`define P(p)
`define INITIAL_MODEL          @(initial_step)
`endif
module rcc(nin , nout);
inout      nin , nout ;
electrical nin , nout ;
//Xs system variables
real  A11             `P(ask="yes") ;
real  A21             `P(ask="yes") ;
electrical  x_lam1 ;
electrical  x_lam2 ;
real X_xI2_net1       `P(ask="yes") ;
real loc              `P(ask="yes") ;
analog begin
  xShift1  = -4.793030e-03 ;
  xShift2  = -4.793020e-04 ;
  if( (1.139373e+01)*(V(x_lam1) - xShift1) + (0.000000e+0)*(V(x_lam2) - xShift2) < -1.934276e+01  )
  begin
    loc = 22 ;
    xShift1  = -1.719380e+00 ;
    xShift2  = 1.828060e+00  ;
    if( (-9.727910e+00)*(V(x_lam1) - xShift1) + (-0.000000e+00)*(V(x_lam2) - xShift2) < -9.810597e-01
        )
      loc = 11 ;
  end
  if(debug == 1 && loc == 11)
  begin
     if(((1.0e+00)*(V(x_lam1) - xShift1) + (0.0e+00)*(V(x_lam2) - xShift2) > 1.728103e+00) ||
        ((0.0e+00)*(V(x_lam1) - xShift1) + (1.0e+00)*(V(x_lam2) - xShift2) > 5.608539e+00) ||
        ((-1.0e+00)*(V(x_lam1) - xShift1)+ (0.0e+00)*(V(x_lam2) - xShift2) > 1.697667e+00) ||
        ((0.0e+00)*(V(x_lam1) - xShift1) + (-1.0e+00)*(V(x_lam2) - xShift2) > 5.785191e+00) )
     $strobe("out_of_loc_%f",loc);
  end
  if(loc == 11)
  begin
    A11         = -1.039866e+01 ;
    A21         =  0.000000e+00 ;
    A12         =  0.000000e+00 ;
    A22         = -1.099663e+03 ;
    B11         = -9.145890e+00 ;
    B21         =  9.999631e+02 ;
    xShift1     = -4.793030e-03 ;
    xShift2     = -4.793020e-04 ;
    uOp         =  0.000000e+00 ;
  end
  if(loc == 22)
  begin
    A11         = -1.867446e+01 ;
    A21         =  0.000000e+00 ;
  I(x_lam1) <+ -1*scale*ddt(V(x_lam1))  ;
  I(x_lam1) <+ scale*(A11*(V(x_lam1) - xShift1) + A12*(V(x_lam2) - xShift2) + B11*(V(nin) - uOp));
  I(x_lam2) <+  -1*scale*ddt(V(x_lam2))  ;
  I(x_lam2) <+ scale*(  A21*(V(x_lam1 ) - xShift1 ) + A22*(V(x_lam2 ) - xShift2 ) +   B21*(V(nin) -
      uOp) );
  X_xI2_net1 = xOp1 + F11*(V(x_lam1) - xShift1) + F12*(V(x_lam2) - xShift2) + FooEoob11*(V(nin) -
      uOp);
  X_neg = xOp22 + F221*(V(x_lam1) - xShift1) + F222*(V(x_lam2) - xShift2) + FooEoob221*(V(nin) - uOp);
end
endmodule
```

D.3 SystemC-AMS Model of the Running Example

An example of a SystemC-AMS model for the running example is given in Listing D.3 using the *disC method*. The explanation of the model is stated in Section 4.6.3.

Listing D.3: Abstracted SystemC-AMS model of the running example

```
// Creation options --------------------------
//
// Name of source file: rccTransistors_gc
//---------------------------------------------
#pragma once
#include <systemc>
#include <systemc-ams>

SCA_TDF_MODULE( HA_TDF_rcc )
{
  sca_tdf::sca_in<double> in_nin;
  sca_tdf::sca_out<double> out_nout;

  HA_TDF_rcc( sc_core::sc_module_name nm ) :
    in_nin("in_nin"),
    out_nout("out_nout")
    {}

  void initialize()
   {
    tf = sca_util::sca_create_tabular_trace_file("trace_HA_TDF_rcc.dat");
    sca_util::sca_trace(tf, V1_br ,"V1_br");
    loc     = 21;
    xLam(0) = 0.0;
    xLam(1) = 0.0;
   }
  void set_attributes()
   {
        set_timestep(sca_core::sca_time(1.000000e-04, sc_core::SC_SEC));
   }
  void processing()
   {
    if(loc == 11)
     {a(0,0) = -10.398661;
      a(1,0) = 0.000000;
      b(1,0) = 999.963073;
      xShift(0) = -0.004793;
      xShift(1) = -0.000479;
      fooeoob(23,0) = 1.000000;
      if( (1.139373e+01)*(xLam(0) - xShift(0)) + (0.000000e+00)*(xLam(1) - xShift(1))  <
          -1.907022e+01 )
       {loc = 22;}
      if( (-1.139373e+01)*(xLam(0) - xShift(0)) + (0.000000e+00)*(xLam(1) - xShift(1))  <
          -1.968965e+01 )
       {loc = 21;}
     }
    if(loc == 22)
     {
     }
    u(0)    = in_nin.read() - uOp(0);
    xLam(0) = a(0,0)*(xLam(0)-xShift(0)) + a(0,1)*(xLam(1)-xShift(1)) + b(0,0)*x(0);
    xLam(1) = a(1,0)*(xLam(0)-xShift(0)) + a(1,1)*(xLam(1)-xShift(1)) + b(1,0)*x(0);
    xLam(0) = xLam(0) + xShift(0);
    xLam(1) = xLam(1) + xShift(1);
    xI2_net1.write( xOp(0)  + f(0,0)*(xLam(0) - xShift(0)) + f(0,1)*(xLam(1) - xShift(1)) +
        fooeoob(0,0)*(in_nin.read() - uOp(0)) );
    neg.write( xOp(21)  + f(21,0)*(xLam(0) - xShift(0)) + f(21,1)*(xLam(1) - xShift(1)) +
        fooeoob(21,0)*(in_nin.read() - uOp(0)) );
   }
  public:
  sca_util::sca_matrix<double> a, b, c, d, f, fooeoob;
  sca_util::sca_vector<double> xOp, uOp, xShift;
  int loc;
  sca_util::sca_vector<double> xLam, u, y;
  sca_tdf::sca_trace_variable<double> nin;
};
```