DISSERTATION

Symbolic Simulation of Mixed-Signal Systems with Extended Affine Arithmetic

Thesis approved by the Department of Computer Science of the University of Kaiserslautern for the award of the Doctoral Degree Doctor of Engineering (Dr.-Ing.)

to

Čarna Radojičić, M.Sc.

Reviewers:

Univ. Prof. Dr. Christoph Grimm
Design of Cyber Physical Systems
University of Kaiserslautern

Univ. Prof. Dr. Lars Hedrich
Institute of Computer Science
Johann Wolfgang Goethe-Universität, Frankfurt

Prof. Dr. Zainalabedin Navabi
ECE Department
Worcester Polytechnic Institute, Massachusetts

Dean:

Univ. Prof. Dr. Klaus Schneider
Embedded Systems Group
University of Kaiserslautern

Date of Defense: 8th September, 2016

D 386

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Acknowledgment

I would like to express my sincere gratitude to my supervisor Prof. Christoph Grimm for his guidance, remarks and engagement throughout my work. Thank You dear professor for supporting and encouraging me in every step of my difficulties during this thesis. I am also grateful to my second and third supervisor Prof. Lars Hedrich and Prof. Zainalabedin Navabi for the careful reading of my work and giving useful comments and suggestions.

I give special thanks to my colleagues from the Institute of Computer Technology TU Vienna, Florian Schupfer and Michael Rathmair. Thank you Florian for the opportunity to be the part of your project and research team. Many thanks to all my colleagues at the Department for Design of Cyber-Physical Systems TU Kaiserslautern Xiao Pan, Frank Wawrzik, Ralf Gruenwald, Javier Moreno, Christopher Heinz and Thiyagarajan Purusothaman for cooperation and a pleasant working ambiance.

I am very grateful to my sister and brother for their unconditional support and motivation throughout my life. They are the most important people in my life and I dedicate this thesis to them. Special thanks I want to give to my parents on their boundless love they gave to me. My dear father and mother, I know you are now on more beautiful place, happy and proud. I will love you forever.

Last but not least, many thanks I give to all my friends, especially Branka and Zorica for all their love, support and patience. Thank you.

Kurzfassung

Analog-Digitale Systeme verbinden analoge Schaltungen mit digitalen Hardware und Software Systemen. Eine konkrete Herausforderung ist die empfindliche Reaktion von analogen Teilen auf bereits kleine Änderungen der Parameter bzw. Eingangssignale. Die genauen Werte von Schaltungs- und und Systemparametern wie z.B. Prozess, Spannung und Temperatur sind oft unbekannt; wir modellieren sie daher wie unbekannte Werte (‘Unsicherheiten’). Unbekannte Werte von Parametern und Eingangssignalen können das Systemverhalten beeinflussen und dazu führen, dass Eigenschaften des Systems nicht mehr in dem vorgegebenen Rahmen liegen. Zur Verifikation Analog-Digitaler Systeme ist der Einfluss von Unsicherheiten auf das Systemverhalten von zentraler Bedeutung.

Verifikation von Analog-Digitalen Systemen wird normalerweise durch numerische Simulation durchgeführt. Ein einziger Simulationslauf ermöglicht es Entwicklern einzelne Parameterwerten aus oftmals Bereichen von unsicheren Werten zu verifizieren. Numerische Simulationsmethoden wie die Monte Carlo Simulation, die Corner Case Simulation und erweiterte Methode wie Importance Sampling oder Design-of-Experiments ermöglichen die Verifikation von Bereichen – auf Kosten einer höheren Anzahl an Simulationsläufen und dem Risiko, dass mögliche Fehler nicht entdeckt werden. Formale und symbolische Methoden sind interessante Alternativen. Diese Methoden bieten eine umfassende Verifikation. Aber formale Methoden skalieren nicht gut bezogen auf Heterogenität und Komplexität. Diese Ansätze sind auch nicht kompatibel zu bestehenden und etablierten Modellierungssprachen. Das erschwert ihre Integration in den industriellen Entwurfsfluss. In frühere Arbeiten zur Verifikation von Analog-Digitalen Systemen wurde Affine Arithmetik für die symbolische Simulation benutzt. Dies ermöglicht die hohe Abdeckung durch formale Methoden mit der einfachen Anwendbarkeit der Simulation zu kombinieren. Affine Arithmetik berechnet die Ausbreitung von Unsicherheiten durch meist lineare, analoge Schaltungen und DSP Applikationen mit exakten Werten. Aber, sie ist aktuell nur in der Lage mit angrenzenden Bereichen zu rechnen, ermöglicht jedoch nicht die Darstellung von und das Rechnen mit diskreten Werten, wie sie beispielsweise durch Software entstehen. Dies ist eine gravierende Einschränkung: Unsicherheiten in Analog-Digitalen Systemen werden oft durch eingebet-

tete Software kompensiert; Verifikation von Systemeigenschaften muss daher beide Teile, analoge Schaltungen und eingebettete Software erfassen.

Das Ziel dieser Arbeit ist die Erweiterung von Affiner Arithmetik, die auch symbolische Simulation von digitaler Hardware und Software ermöglicht, und schließlich die Demonstration ihrer Anwendbarkeit und Skalierbarkeit. Verglichen mit anverwandten Arbeiten und State-of-the-art, bietet Dissertation die folgende Leistungen:

1. Die Arbeit stellt Erweiterte Affine Arithmetik Formen (XAAF) zur Repräsentation von Branch- und Mergeoperationen vor.
2. Die Arbeit beschreibt arithmetische und Vergleichsoperationen mit XAAF, und verringert die Überapproximation mit Hilfe von LP Solvern.
3. Die Dissertation zeigt und diskutiert Möglichkeiten die XAAF in bestehende Modellierungssprachen, insbesondere SystemC, zu integrieren. Auf diese Weise können Brüche im Entwurfsfluss vermieden werden.

Die Anwendung und Skalierbarkeit dieser Methodik wird mit der symbolischen Simulation eines Δ-Σ Wandlers dritter Ordnung und einer PLL-Schaltung eines IEEE 802.15.4 Transceivers demonstriert.

Abstract

Mixed-signal systems combine analog circuits with digital hardware and software systems. A particular challenge is the sensitivity of analog parts to even small deviations in parameters, or inputs. Parameters of circuits and systems such as process, voltage, and temperature are never accurate; we hence model them as uncertain values ('uncertainties'). Uncertain parameters and inputs can modify the dynamic behavior and lead to properties of the system that are not in specified ranges. For verification of mixed-signal systems, the analysis of the impact of uncertainties on the dynamical behavior plays a central role.

Verification of mixed-signal systems is usually done by numerical simulation. A single numerical simulation run allows designers to verify single parameter values out of often ranges of uncertain values. Multi-run simulation techniques such as Monte Carlo Simulation, Corner Case simulation, and enhanced techniques such as Importance Sampling or Design-of-Experiments allow to verify ranges – at the cost of a high number of simulation runs, and with the risk of not finding potential errors. Formal and symbolic approaches are an interesting alternative. Such methods allow a comprehensive verification. However, formal methods do not scale well with heterogeneity and complexity. Also, formal methods do not support existing and established modeling languages. This fact complicates its integration in industrial design flows.

In previous work on verification of Mixed-Signal systems, Affine Arithmetic is used for symbolic simulation. This allows combining the high coverage of formal methods with the ease-of use and applicability of simulation. Affine Arithmetic computes the propagation of uncertainties through mostly linear analog circuits and DSP methods in an accurate way. However, Affine Arithmetic is currently only able to compute with contiguous regions, but does not permit the representation of and computation with discrete behavior, e.g. introduced by software. This is a serious limitation: in mixed-signal systems, uncertainties in the analog part are often compensated by embedded software; hence, verification of system properties must consider both analog circuits and embedded software.

The objective of this work is to provide an extension to Affine Arithmetic that allows symbolic computation also for digital hardware and software sys-

tems, and to demonstrate its applicability and scalability. Compared with related work and state of the art, this thesis provides the following achievements:

1. The thesis introduces extended Affine Arithmetic Forms (XAAF) for the representation of branch and merge operations.
2. The thesis describes arithmetic and relational operations on XAAF, and reduces over-approximation by using an LP solver.
3. The thesis shows and discusses ways to integrate this XAAF into existing modeling languages, in particular SystemC. This way, breaks in the design flow can be avoided.

The applicability and scalability of the approach is demonstrated by symbolic simulation of a Δ-Σ Modulator and a PLL circuit of an IEEE 802.15.4 transceiver system.

Contents

Chapter 1

Introduction

1.1 Motivation

Mixed-signal systems are systems composed of analog and digital circuits usually controlled by additional software. Analog circuits are by nature highly sensitive to deviations in their operating conditions/environments. These deviations cause variations in the circuit parameters, which modify the original circuit behavior. The modified behavior can change the properties of the original behavior and lead to their violation. Hence it is of crucial importance to take them into account during system analysis and verification.

Usually, the exact values of these variations are **unknown** and in this work they are defined and modeled as *uncertain* values. In the rest of the thesis the term *uncertainties* will be used to describe them. There are various sources of these *uncertainties*, such as:

- Variations in the manufacturing process
- Variations during life time operation – temperature drift, components aging, etc.
- Uncertainties introduced by abstraction of reality. Real systems are often abstracted with behavioral models, which are easier to simulate and verify
- Uncertainties introduced by analog to digital conversion and the computation in the digital domain with finite-precision arithmetic
- Noise in single blocks, power supply, system environment, etc.

Figure 1.1 shows an example of an IEEE 802.15.4 RF (Radio Frequency) transceiver with some possible uncertainties. Noise in power supplies can have a high impact on the system behavior. It can cause variations in decision thresholds of a phase/frequency detector of the transceiver PLL (Phased-Locked Loop) circuit. This can further lead to inability of the PLL to lock and generate the required carrier frequency brought to the input of a mixer circuit. The frequency mismatch in the mixer causes introduction of new frequencies in the band of interest. This undesired effect might lead to wrong interpreted data on the receiver side.

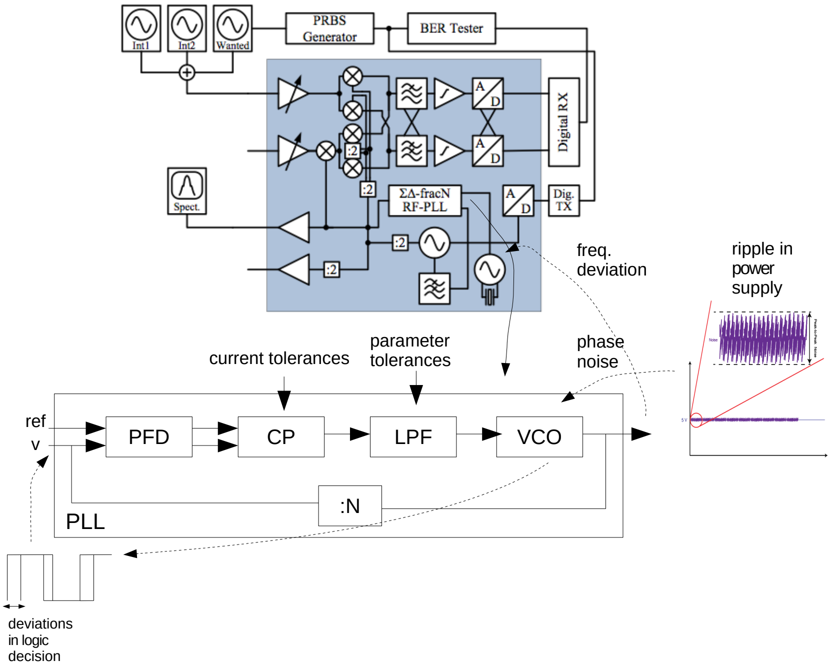


Figure 1.1: Block diagram of RF transceiver with possible uncertainties

A general approach to analyze a system behavior under uncertainties is a numeric simulation. In the numeric simulation one simulation run is able to cover only one particular operating condition. To include the wide range of uncertain values, a high number of runs is required. If one run finds a trajectory that violates the properties, one can guarantee the presence of errors. However, if this trajectory is not found, one cannot guarantee the absence of these errors.

In contrast to simulation, formal methods are able to verify a system over a wide range of operating conditions within one execution. However, these methods are not able to cope with a complexity and heterogeneity of AMS (Analog/Mixed-Signal) systems, such as one shown in Fig. 1.1.

In addition, neither simulation nor formal methods are able to trace the specification violations to their sources. Formal methods are able to generate counter examples which lead to specification violation. However, formal verification is not able to give the answer to this question: "Which part of a system causes a specification to fail?"

1.2 Goals and scope of the work

The goal of this work is to combine the high coverage of formal methods with the general applicability of simulation. The first step towards this goal is to capture the sets of uncertain values in a formal way using, for example ranges. Section 3.5 provides possible classes of uncertainties and the way to formally model them. The next step defines operations on ranges required to capture dynamics in a system behavior. The overall idea of the proposed approach is shown in Fig. 1.2.

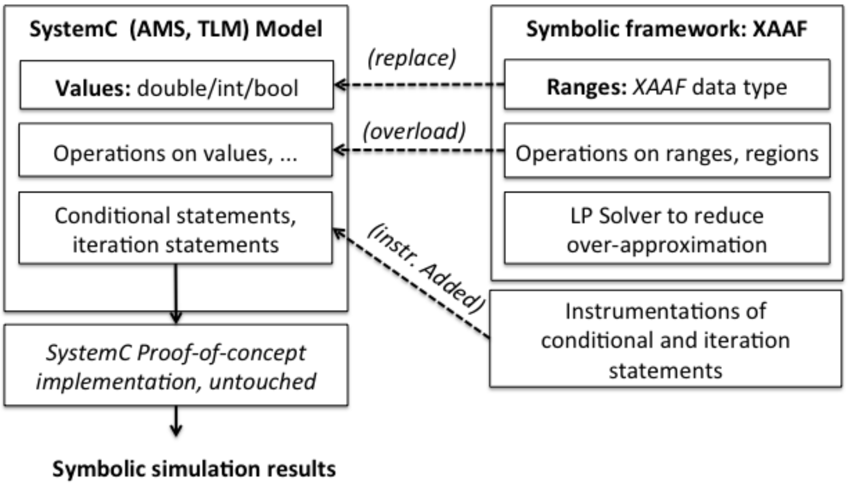


Figure 1.2: The idea of the methodology

The proposed methodology is applied on AMS systems modeled by a block diagram. This way of modeling is a common industrial practice used in traditional hardware description languages (HDLs) such as SystemC-AMS, Matlab-Simulink, VHDL-AMS, Verilog, etc. Uncertainties are captured as symbolic ranges. Operations applied on ranges compute a symbolic output in one simulation run. The symbolic response covers the whole set of trajectories over the considered range of uncertain values.

The scope of the work is currently reduced to SystemC AMS simulator. However, the same concept can be implemented within any simulator, which supports the use of abstract data types. As simulator, this work uses the proof-of-concept implementations available from Accellera and Fraunhofer (AMS extensions). This simulator has been developed for verification and validation by numerical, non-symbolic simulation.

In order to permit symbolic simulation of existing models, we give the double/int/bool values of a numeric simulator the semantics of symbolic simulation. Objective is to allow us the use of the existing, numeric simulator for symbolic simulation. This permits us to keep the changes and modifications of the models small while ensuring compatibility with existing tools and flows. For integration of symbolic simulation into the existing simulator, we use the following approaches (see Fig. 1.2):

- Operator overloading, which allows us to define operations on ranges. In this way, arithmetic and relational operators get symbolic semantics. In the model, the signal and variables values that shall be simulated symbolically must be the abstract data type XAAF.
- Instrumentation of control flow statements that is necessary for symbolic simulation of multiple control flow paths on the software side.

1.3 Hypotheses

Combining symbolic approach with simulation, the new methodology should successfully deal and solve the problems of state of the art methods. To show this, I define the requirements that the method should meet. In the rest of the thesis I will call them hypothesis whose validation will be presented at the end of this work.

Hypothesis 1 - Applicability on Complex Mixed-Signal Systems.

The proposed methodology is based on simulation that allows the application of the method on general systems. This hypothesis is validated applying the new verification methodology on a complex industrial case study: a dual-path charge-pump PLL circuit of one IEEE 802.15.4 transceiver. The hypothesis is accepted if the method is able to show that the system meets/does not meet the desired property in the presence of different sources of uncertainties.

Hypothesis 2 - Comprehensive Verification Coverage.

The new methodology captures uncertain values as ranges giving them symbolic values. Computation with symbolic values results in a symbolic system response. The hypothesis is accepted if the symbolic output encloses the whole set of outputs computed for the particular parameter values inside the ranges. This is shown finding the worst-case bounds and comparing their values with the ones obtained by numeric techniques such as Monte Carlo and Design of Experiments.

1.4 Contributions

The starting point of this work is Affine Arithmetic proposed by [1]. Affine Arithmetic (AA) is a powerful technique to compute with intervals in a symbolic way. It was introduced as an improved alternative of Interval Arithmetic (IA) [2]. Symbolic representation of ranges with AA overcomes a well-known dependency problem of IA. AA proved its efficiency in a vast number of applications.

The first application of AA was in computer graphics [3]: computing octrees, quad-trees and ray tracing. This work reports more accurate results compared with Interval Arithmetic applied on the same examples.

[4, 5] use AA In Digital Signal Processing (DSP) to analyze finite-precision effects caused by rounding errors in floating-point arithmetic. An upper bound for the error is estimated using a general applicable static error analysis based on AA. The results of the proposed error estimation are compared with techniques based on detailed simulation strategy [6]. It is shown that using AA a comparable estimation can be obtained with a significantly lower computational effort.

AA also found its applicability in the verification and analysis of analog circuits. [7] proposes semi-symbolic analysis of analog circuits based on AA. Using this approach AA is used in sizing of analog circuits taking the changes of operating conditions and manufacturing tolerances into account. Affine Arithmetic computes the bounds of the worst-case circuit behavior and the global minimum of sizing problem is determined due to inclusion isotonicity.

[8] and [9] use AA for symbolic (forward) simulation of mixed-signal systems and analog circuits, respectively. The goal of these works was to increase coverage of simulation-based techniques towards formal methods.

However, Affine Arithmetic currently supports only computations in the continuous domain. This is not enough for symbolic simulation of mixed-signal systems. Beside continuous parts, they also contain digital or software parts, which require computations in the discontinuous domain. The discrete operations cannot be handled with the standard form of Affine Arithmetic.

This thesis extends Affine Arithmetic towards crossing continuous/discontinuous borders with symbolic values. The extended Affine Arithmetic will be able to handle:

- Discrete parts of Mixed-Signal systems such as comparators, switches, etc.
- Control flow statements in the software code with conditional variables as ranges

More details about the extension of Affine Arithmetic and computation with extended affine forms will be given in Chapter 4.

1.5 Outline

The rest of the thesis is organized as following. Chapter 2 briefly discusses state of the art methods used in verification of analog/mixed signal systems. The methods are compared with respect to run-time complexity, scalability and coverage obtained during verification process.

Chapter 3 is reserved for a brief overview of symbolic simulation based on Affine Arithmetic (AA), which is the basis for this thesis. The mathematical form of AA and the basic mathematical operations with affine forms can be found in this chapter. Nonlinear operations with AA terms require the use of approximation schemes that are in this chapter briefly described. To keep the form of AA, these operations add a new symbol per each execution. To avoid symbol explosion, the method acting as a garbage collection or Hansen's forms are used. The short description of approaches and the improved scalability of AA is given.

Finally, this chapter shows the practical use of Affine Arithmetic, listing sources of system uncertainties and describing the way to model them using AA. Symbolic simulation based on AA results in symbolic outputs, which encloses the set of simulation traces, obtained in one (symbolic) simulation run. The way to perform the symbolic simulation on system, as well on circuit level is briefly explained.

Chapter 4 briefly introduces the extension of Affine Arithmetic proposed in this work. It allows computation with uncertain values on the digital and software side of a system. Beside standard mathematical operations this chapter introduces new operations to pass uncertainties through the discontinuous domain. The idea is illustrated through two small examples: a simple control flow in the software code and a water level control system modeled as automata with two discrete states. The chapter ends with a brief explanation of the method implementation.

Chapter 5 introduces the assertion-based approach proposed to automatically compare symbolic outputs with the properties a system should fulfill. The following points are covered:

- the assertion language with the appropriate set of operators
- the list of typical system properties and the way to describe them using these operators
- illustration using two small examples: a 2nd order IIR (Infinite Impulse Response) filter and a closed loop control system. The properties, that were verified, are: the overflow of filter output due to finite precision in DSP and the stability property of a control system.
- the implementation of approach as a separate library in SystemC AMS simulation environment

Chapter 6 proves the efficiency and the applicability of Extended Affine Arithmetic on two more complex non-trivial case studies: a 3rd Order Delta-Sigma Modulator and a Charge-Pump PLL circuit. The considered PLL circuit is used for generation of high frequency carriers in a multiband IEEE 802.15.4 RF transceiver shown in Fig. 1.1. The proposed assertion approach is used to describe typical properties of the case studies and checked them automatically. The chapter ends with the discussion that highlights the advantages of the proposed method over state of the art methods. Chapter 7 concludes the thesis and points to future directions.

Chapter 2

State of the art

Figure 2.1 gives an overview of state of the art methods for mixed-signal verification. These can be classified into two major classes: simulation and formal methods (see Fig. 2.1).

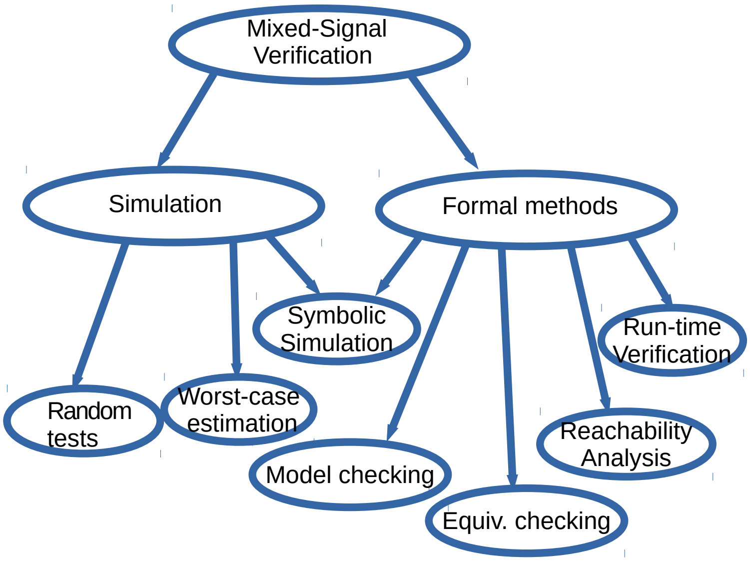


Figure 2.1: Methods for verification of Mixed-Signal Systems

2.1 Simulation-based approaches

Simulation is a common method used to analyze analog/mixed-signal behavior over a wide set of operating conditions. This approach can be categorized into two groups:

1. Statistical methods based on generation of random test cases

2. Methods based on estimation of worst-case behavior

Statistical methods use probability density functions to generate random values for system parameters. These are known as Monte-Carlo simulation. For a thorough system analysis this approach is a very expensive where too many tests are repeated to obtain sufficient problem coverage. To reduce this number, the technique called Importance sampling [10, 11] is proposed. In contrast to Monte-Carlo this approach does not use the original probability density function, but the samples are chosen from the other function called importance density function. Using this function the values are sampled from regions, which have the highest impact on the system performance. Although the number of runs is reduced, random tests cannot guarantee that worst-case scenarios are covered.

The focus of the second class of simulations is to investigate the effects of parameters and find the values, which can lead to worst-case scenarios. [12] considers corner values of parameters to estimate the worst-case behavior. However, this can lead to false estimation, since corners are not by default worst-case parameter values. The methods in [13] and [14] deliver more accurate results but a specification of a minimum yield requirement is required. [15] proposes Design of Experiments (DOE) which uses different classes of metamodels to find parameters with the highest impact. It estimates “worst-case” system behavior using significantly lower number of simulation runs. However, in general all methods based on numeric simulations require multi runs which still cannot guarantee dependability. Table 2.1 compares multi-run methods looking at run-time complexity.

Table 2.1: Comparison of run-time complexity.

method	complexity	comment
Monte-Carlo	O(1)	probabilistic approach; many runs needed
Importance sampling	O(1)	monte-carlo runs reduced, but still many runs
Corner-Case	$O(2^k)$	runs increase exponentially with k uncert.
Design of Experiments		number of runs dependable on various factors: number of uncertainties, Euclidean distance between pairs of samples in the parameter space, used meta-model

Numeric simulations are able to prove the existence of errors but cannot prove the absence of them. Counter-examples provide 100% guarantee that system properties are violated. If the counter-examples are not found, the correctness of system behavior cannot be proved due to finite numbers of simulated scenarios. To increase the coverage of numeric simulations, formal verification methods were proposed.

2.2 Formal methods

The high applicability and efficiency of formal methods were proved in the verification of digital systems [16]. Clarke in [16] proposes model checking methods. These are based on state-graph models of system behavior and allow exhaustive exploration of all possible states and transitions in the graph model. However, due to infinite continuous state space, formal verification of analog/mixed-signal systems is still a challenge. Literature survey provides the following techniques for formal verification of analog/mixed-signal designs.

2.2.1 Equivalence checking

One approach to verify analog/mixed-signal designs in more formal way is equivalence checking. It is based on proving the equivalence of two models. In [17] verification of analog circuits is done by checking equivalence between two transfer functions. One transfer function models the circuit implementation and the other the specification. The transfer functions are transformed to the discrete Z-domain, which allows the canonical encoding into Ordered Binary Decision Diagrams (OBDDs). However, this method considers circuit parameters only in nominal conditions; no variations are taken into account. Later, [18] proposes the method, which explores the effect of parameter tolerances to the circuit behavior. The circuit transfer function is extracted from the circuit netlist. Both methods are restricted to linear analog circuits.

[19, 20] proposes an equivalence checking approach for nonlinear analog circuits. The idea of the method is to compare functional behaviors of two circuits represented in the form of differential algebraic equations (DAE). Since levels of abstractions for modeling two circuits might be different, the internal state space variables might also be different. Thus, the direct comparison of the state space would lead to wrong verification results. Transforming the state space of both circuits to canonical representations, which are then directly compared, solves this. However, this is not a trivial task and for complex systems the transformation could not be automatically performed.

Beside circuit-level, equivalence checking can also be performed at system-level as proposed in [21]. There, equivalence is checked between two VHDL-AMS system models. Each VHDL-AMS model contains digital part, analog part and convertor components for converting analog to digital and via versa. Digital part of two designs is verified using classical equivalence checking methods like SAT/BDD method. Analog parts were simplified using rewriting engine and compared using AMS simulator. However, it is not easy to find appropriate rewriting rules to arbitrary classes of analog circuits and this method is often restricted to simple and specific designs. In general, all methods based on equivalence checking suffer from the complexity problem as it is surveyed in [22].

Table 2.2: Methods for equivalence checking [22]

	[17]	[19]	[18]	[21]
Type of systems	Linear	Nonlinear	Linear	Nonlinear AMS
Models	Transfer function	ODE-DAE	Transfer function	ODE-DAE
Analysis Regions	Transient response	Near operating point transient response	Near operating point	Functional analysis
Analysis domain	Frequency	Time	Frequency	Time
Techniques and analysis	OBDDs comparisons	Qualitative analysis	Interval analysis	Rewriting, SAT simulation
Tools	N/A	MAPLE	MAPLE	M-CHECK
Case studies	Low pass filter	CMOS inverter, Opamp	Band pass filter	D/A converter

2.2.2 Model checking

The other approach to formal verification is based on algorithmic system verification like model checking. Depending on the system complexity the researchers propose two classes of methods: direct and indirect methods.

The first class of methods is applied on the original systems where a generated state model is the exact representation of the system behavior [23, 24]. These methods are restricted to linear systems with a simple continuous dynamics. For nonlinear systems with more complex dynamic behavior the computation time to generate a discrete model significantly increases. Therefore, the indirect methods must be applied. They convert original systems to abstract models easier to analyze and verify.

The work in [25, 26] proposes timed automata to approximate continuous system parts. In [27] the time automata is refined by adding clocks allowing us to reduce over-approximation of abstract models; the number of spurious behaviors that do not correspond to concrete ones. The applicability of this method is restricted to systems with a monotonic behavior.

[28] proposes Linear Hybrid Automata (LHA) to abstract continuous system dynamics. To deal with system nonlinearities, [29, 30] extend LHA with a piecewise linear model. The approach is based on linear phase-portrait approximation where the state space of the nonlinear system is divided into linear regions. Since there is no standard method for partition of state space, strongly nonlinear models can hardly be handled.

[31, 32] proposes the discretization of the infinite continuous state space of nonlinear analog systems. The continuous space is discretized into regions called hypercube, which represent the discrete states of the finite state model. Once a model is created, model-checking algorithms are applied to verify typical system properties. The system properties are described with Computation Tree Logic (CTL) formulas. [33, 34] extend CTL formu-

las [35, 36] to cover analog behavior. The main weakness of these approaches is the state explosion problem and the total runtime, which grows exponentially with the number of state variables. Only systems with few state variables (maximum five) can be handled.

In [37] the authors propose a new model for the state representation of AMS designs, Timed Hybrid Petri Nets (THPN). The reachable states of THPN are computed by Difference Bound Matrix (DBM)-based algorithm, which uses convex polygons (zones) to represent them. The desired system properties are described with ACTL (Analog Computation Tree Logic) and automatically verified. This approach requires specification of AMS designs in the form of differential equations (DEs) which are then transformed into THPN.

One more approach towards formal verification of AMS designs is a a Bounded Model Checking (BMC) implemented in the Property-Checker tool [38]. This tool is intended to verify the AMS behaviors, which should reach steady states. As all previous methods, this tool also requires the specification of AMS design in the intermediate language. The design must be described using the semantics of XML language.

The first step towards the integration of formal methods in a standard design flow is proposed by [39]. The authors improve the previous work [37] to allow specification of AMS circuits using a familiar language (in this case VHDL-AMS). A new model called Labeled Hybrid Petri Net (LHPN) is introduced, which is automatically generated from VHDL-AMS designs. To enable the application of the methodology in general (not only on VHDL-AMS designs), [40, 41] propose the generation of LHPNs from simulation traces got from an arbitrary simulator. System properties are then verified using one of the following model checkers: binary decision diagrams (BDD)-based model checker [42, 43], (difference bound matrices) DBM-based model checker [44] and (satisfiability modulo theories) SMT-based model checker [42, 43]. Since LHPN is generated from simulation traces, the quality of the model depends on the quality of delivered simulation results. To increase the quality of verification results the authors propose the use of coverage metrics. They should give information about the regions, which are not explored by simulation traces but where the undesired behavior can appear. The lower quality of simulation results, the more information coverage metrics should deliver.

2.2.3 Reachability analysis

To deal with the state explosion problem, one line of research performs state exploration directly on system dynamics. This approach is in literature known as reachability analysis. In general, this approach cannot find the exact representation of all reachable states. This is due to complex system dynamics and unknown exact values of initial conditions. Therefore, reach-

Table 2.3: Methods for model checking [22]

	[32, 34]	[37]	[38]
Type of systems	Nonlinear	Nonlinear	AMS
Models	ODE, DAE	THPN/ODE	piecewise linear automation
Analysis Regions	No restriction	No restriction	Steady State
Techniques and analysis	Numerical anal.	Numerical approx.	Bounded MC
State space partitions	HyperCubes	Convex polygons	
Temporal Logic	CTL-AT	ACTL	FOL
Tools	Amcheck	ATACS	Property Checker
Case studies	Smitt trigger, Opamp, VCO	Tunnel diode, PLL	Sequential circuit

able sets are often approximated using different geometrical representations.

[45] proposes the partition of the continuous state space using hypercubes of fixed size. However, the computation time of this approach is too high. To reduce this time the authors in [46] propose the reduction of the state space dimension by projections of the state space with polygons.

In **d/dt** [47, 48] and **Checkmate** [49, 50, 51, 52] tools, the set of reachable states is approximated using polyhedra. **d/dt** uses orthogonal polyhedra, while **Checkmate** is based on abstractions using the sequence of convex polyhedra, so called *flow pipe* approximations. In [53] the authors extend **d/dt** to support verification of nonlinear analog circuits whose behavior is described with differential algebraic equations (DAEs). The extended approach is illustrated on a second order biquad low pass filter. However, **d/dt** misses an automatic translation from a circuit description to a hybrid automata (HA) and a formal language to specify analog properties. In [54] the **Checkmate** tool is used to verify properties of two analog circuits, a tunnel diode oscillator circuit and a delta-sigma modulator. Systems are converted to Approximate QTSs (Quotient Transition Systems), which represent conservative approximation of the infinite-state representation of the original system. Since AQTSs represent approximations of the considered hybrid system, the tool verifies only universal properties (properties which must hold for all paths of QTS). These properties are described by ACTL class of formulas [55].

One more approach on reachability analysis not considered by [22] is the work presented recently in [56]. This work proposes the use zonotopes for the approximation of the reachable set. It provides efficient results on verification of a locking property of dual-path charge-pump PLL circuit. However, there are some open questions: “Can this approach handle circuit nonlinearities (e.g. in Voltage-Controlled Oscillator)?” or more important “Can this approach be applied in general?”

Table 2.4: Methods for reachability analysis [22]

	[45]	[46]	[54]	[53]
Types of Systems	Nonlinear	Nonlinear	Nonlinear	Nonlinear
Models	ODE	ODE	HA/ODE-DAE	HA/ODE-DAE
Analysis Regions	No restriction	No restriction	No restriction	No restriction
Techniques and analysis	Simulation	Projection numerical approx.	Numerical approx.	Numerical approx., MILP
State space partitions	Fixed size hyperCubes	Polygons	Convex polyhedra	Orthogonal polyhedra
Tools	COSPAN	Matlab/Coho	Checkmate	d/dt
Case studies	Interlock Circuits	Van der Pool Oscillator	Tunnel Diode $\Delta - \Sigma$ mod	Low Pass Filter $\Delta - \Sigma$ mod

2.2.4 Run-time verification

Run-time verification is the other set of methods dealing with the complexity problem of formal methods. The basic idea is to use monitors, that monitor signal traces generated from numeric simulations. Monitors check properties automatically and can work in *offline* and *online* mode. The properties to be verified are described with assertions whose semantic follow the syntax of the appropriate specification language. This idea was firstly adopted for verification of digital designs [57]. In [58, 59] the authors propose formal specification languages like Property Specification Language (PSL) and System Verilog Assertion (SVA). For specification of analog behavior these languages require extensions and the certain number of modifications.

[60] presents Signal Temporal Logic (STL), as extension of PSL. The language is used to specify analog properties, which are verified *offline*. [61] also proposes the *offline* tool, but for verification of mixed-signal properties. This tool requires modeling AMS system with System of Recurrence Equations (SRE) and it does not support mixed behaviors for continuous time (only discrete).

In general, the weakness of offline methods is that properties can be checked only after simulation. For complex systems memory usage to save the signal traces can increase significantly.

To face with this problem the authors in [62] extend the work [60] towards *online* monitoring. The approach proposes an incremental monitoring method, which combines the simplicity of the offline method with the early failure detection of online monitoring. Concretely, the offline method is not applied on the entire set of simulation traces but incrementally on each simulation trace.

Online monitoring can be also found in [63] where SVAs are extended to cover continuous time. In [64] an assertion library is integrated in the environment of MLDesigner which uses multi domain approach to model systems: CTDE (Continuous Time Discrete Event) MOCs (Model of Computations) for analog modeling and Discrete Event MOCs for modeling digital system parts.

Monitoring timed automata (MTA) [65], based on the approach [66], also allows the verification of system properties during simulation. In [67] the authors integrate the MTA in the automatic stimulus generation framework which is interfaced with the VHDL-AMS simulator.

Above described assertion-based methods are limited to verification of time behavior; frequency behavior is not covered. To allow description of properties in the frequency domain, [68, 69] introduce the language of Mixed Signal Assertions (MSA). They combine continuous time, discrete time and frequencies with temporal logic.

In general run-time verification is a very fast and hence an attractive approach. However, there is no guarantee that a system meets specifications due to the finite number of tested simulation traces.

2.2.5 Symbolic simulation

The goal of symbolic simulation is to combine the high coverage of formal methods and interactive way of analysis as simulation. Symbolic simulation is also a part of formal verification, e.g. to compute reachability. The idea is based on symbolic approach, which replaces concrete numeric values with symbols. Symbolic simulation can be seen as a methodology, which removes tradeoff in conventional simulations between required simulation runs and a coverage. With symbolic representation, a comprehensive coverage of formal methods is obtained in one or very few simulation runs.

In early 70's [70] symbolic simulation was applied for software testing and debugging. Since program variables are symbols, control flows in software code are executed for all possible branches/path conditions. For large programs, a comprehensive simulation is problematic, because the number of paths increases exponentially (path explosion problem). To cope with the path explosion problem, Satisfiability Modulo Theories (SMT) solvers[71] are used to more accurately determine the reachable paths. The control flow is pruned by heuristics to focus on "relevant" paths [72, 73]. Recent tools allow symbolic execution at the level of C code [74], Low Level Virtual Machine (LLVM) [75] or Dalvik [76] intermediate code and are able to handle larger software systems. While for C code a specific symbolic executor is used, abstract virtual machines like LLVM or Dalvik do not need any specific high-level language at all.

Design languages such as VHDL, Verilog and SystemC have, together with their simulation methods, reached a complexity where it would be desirable to use an existing compiler and simulator for symbolic simulation or model checking. The question "Can a Simulator Verify a Circuit?" was first asked by Bryant [77] who proposed the use of the ternary logic (0,1,X) of logic simulators for this purpose. Unfortunately, this will quickly lead to simulations with "X" states only.

In [78] Bryant and Seger propose Symbolic Trajectory Evaluation where symbolic simulator is seen as model checker. The properties are described with limited class of formulas written in Linear Temporal Logic (LTL) logic. This work was later extended to support verification of safety properties [79, 80, 81]. [82] proposes Generalized Symbolic Trajectory Evaluation (GSTE) which enables adding fairness to system properties. To avoid simulations with "X" states only, GSTE does not encode the entire system state space symbolically. It looks at the complexity of specification and replaces only required inputs and initial states with symbols. This methodology found place in industry applications; Motorola used STE for verification of memory subsystems while Intel performed block-level verification based on this methodology. However, only limited class of properties can be described and verified; existential properties cannot be expressed. CTL model checking is still more expressive.

In [83] Chris Wilson overcomes the problem of "too many X" combining the ternary logic with more powerful BDD structure. In Wilson's simulator the symbolic values are represented in two ways: as BDD variable or symbol "X" tagged with the corresponding index. When the output values depend on inputs with multiple tags, its result is "X". The idea is to start symbolic simulation using only tagged values for input variables. If outputs result in "X", then input symbols are replaced with BDD variables and symbolic test is performed again. This process is repeated until the output value is either 0 or 1. This simulator is restricted to simple data paths, since every time tagged "X" is replaced with BDD variable the entire simulation must be repeated. Recent tools rely on more powerful BDD techniques or even SAT/SMT solvers. For this purpose, HDLs are translated into automata or other intermediate representations. For formal verification in complex modeling and design languages, languages such as SystemC are translated to intermediate languages, SISSI [84].

Recent works [85, 86] brought the idea of symbolic approach to analog and mixed-signal circuits. Both methods combine symbolic representation of system behavior with Interval Arithmetic (IA)[2] to compute over the ranges of system parameters. In [85] mixed-signal designs are modeled symbolically in the form of recurrence equations and computed over the intervals of system parameters. However, due to divergence problem associated with IA, symbolic simplifications need to be applied at each time step This leads to high computational costs.

[86] proposes symbolic representation of mixed-signal behavior in the form of SAT constraints by using NL-SMT solver. The constraints over variables are represented as intervals using Interval Arithmetic. Similar, the over approximation problem of IA is solved by symbolic simplifications with ICP(Interval Constraint Propagation) which is applied at each computation step. Since this leads to significant increase of computational costs, the combination with numeric simulations was required.

[8, 87, 9] overcomes the problems of IA using the improved alternative called Affine Arithmetic [1, 88]. One more advantage of these works over

[85, 86] is that the symbolic methodology is used directly in an existing simulator without translation a system model to a formal one. However, these approaches were restricted to continuous state space due to inability of Affine Arithmetic to compute in the discrete domain. This work extends Affine Arithmetic towards discrete computations with symbolically represented ranges.

Chapter 3

Symbolic Simulation based on Affine Arithmetic (AA)

3.1 Symbolic simulation

Recent works [8, 89] use Affine Arithmetic [1, 88, 90] for symbolic simulation of analog/mixed-signal systems. A system is simulated over a set of variable inputs, initial conditions and parameter values. All system variations are symbolically modeled as ranges using affine forms. Using symbolic representation Affine Arithmetic brings the simulation the following advantages:

1. High dependability of verification results using one (symbolic) simulation run
2. Measuring the impact of each variation to the total system performance and the trace ability of variations back to their sources

In [8] simulation is performed at system level using an existing SystemC AMS simulator [91], intended for numeric simulation. Beside system-level simulation, symbolic simulation is also performed at circuit level [89]. In [89] Affine Arithmetic is integrated in an equation solver to compute a circuit response over a set of circuit parameter values.

3.1.1 System-level simulation using SystemC (AMS)

In [8] AMS system is modeled in a usual way by a block diagram as shown by Figure 3.1. The functions of blocks are described by mathematical functions in time-domain (multiplication, integration, differential equations, etc.), by transfer functions $H(s)$, or by linear circuits with switches, or by C++ code.

The functional blocks communicate via directed signals (*(timed) dataflow, signal-flow*). For simulation, the outputs of the blocks are computed in discrete time steps, following the data flow's direction by computation of the block's functions, or by solving the equation systems of the blocks. This way of modeling is the common industrial practice and is the underlying

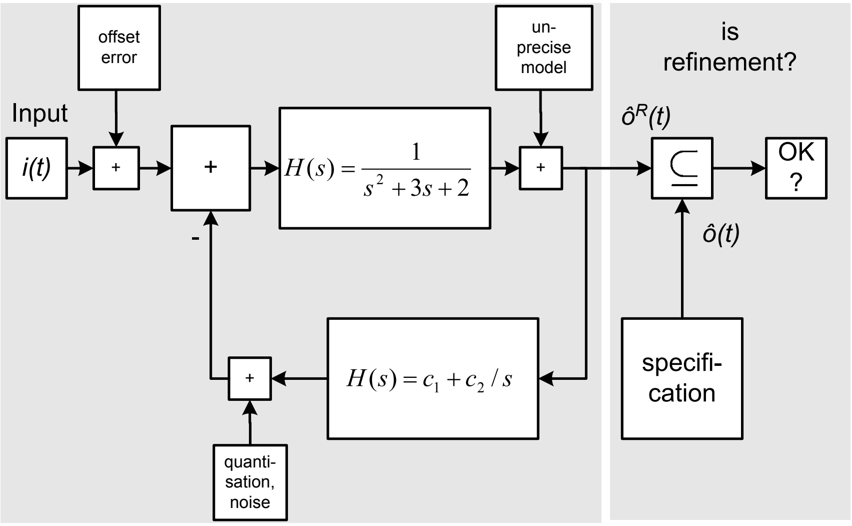


Figure 3.1: Modeling AMS system by a block diagram [8]

model of computation of modeling languages such as Matlab/Simulink and SystemC AMS. [8] uses SystemC AMS.

SystemC AMS [91] is introduced as an extension to SystemC modeling language [92], which enables modeling, simulation and verification of analog and mixed signal systems in the same simulation environment. SystemC is a C++ class library intended for simulation of cycle-accurate system behavior using discrete-event simulation kernel in the background. As an extension, SystemC AMS keeps all properties of SystemC while adding new features to handle mixed-signal behavior. New execution semantics include modeling AMS systems on different levels of abstractions: discrete-time, continuous-time, non-conservative and conservative modeling. They are supported by three models of computation shown in Figure 3.2:

- Timed Data Flow (TDF) - TDF supports the abstraction of signals and system quantities as discrete-time values available at discrete time points. The distance between the points is defined by the sampling period T_s. All standard types supported by C++ can be used to specify the type of discrete-time values. TDF also supports the use of templates to specify value types. Therefore, one can implement its own data type and use it to represent quantities and signal values.
- Linear Signal Flow (LSF) - LSF and ELN support continuous-time modeling which is more natural way of modeling the physical world. The system behavior is specified as a set of differential algebraic equations (DAEs) solved by a suitable equation solver. LSF supports sys-

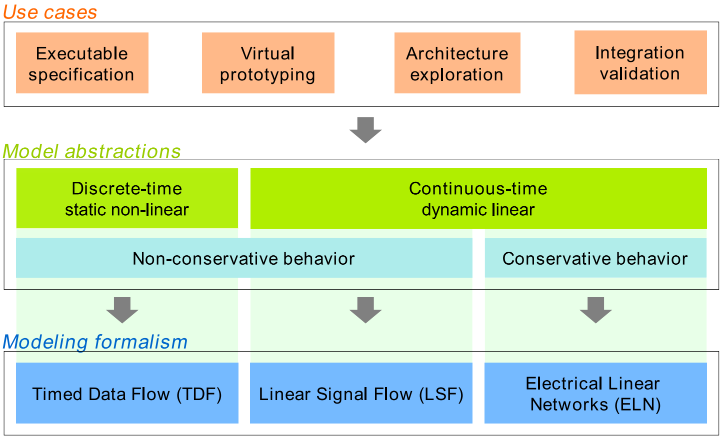


Figure 3.2: Modeling formalisms in SystemC AMS [93]

tem modeling only with the set of available modules called primitive modules. The response of such a system is computed using a linear DAE solver.

- Electrical Linear Network (ELN) - Unlike LSF, ELN supports modeling a conservative continuous-time behavior. These models are more complex, hence they require that interactions between system quantities satisfy Kirchhoff's laws.

In [8] the authors use TDF as model of computation. In order to turn the existing SystemC AMS numeric simulator to a symbolic one, C++ values `double/int/bool` are given the symbolic semantics using Affine Arithmetic (AA). Replacement of standard C++ data types with `AAF` is possible through operator overloading.

One symbolic simulation run results in the safe set of all possible system behaviors over the considered range of operating conditions. The system response for the system shown in Figure 3.1 is shown in Figure 3.3. The errors introduced in the control loop fall towards 0 and the output signal converges to the steady state value. This behavior is exactly what a designer expects from a designed PI controller (Figure 3.1 bottom). In contrast to Interval Arithmetic, Affine Arithmetic is able to track correlations between same quantities in a system loop. This further avoids overapproximation of output signal values that could lead to false negatives.

3.1.2 Circuit-level simulation

Beside at system level, symbolic simulation can also be applied at circuit level [89, 94, 95].

[89] implements a symbolic SPICE-like circuit simulator. It was used

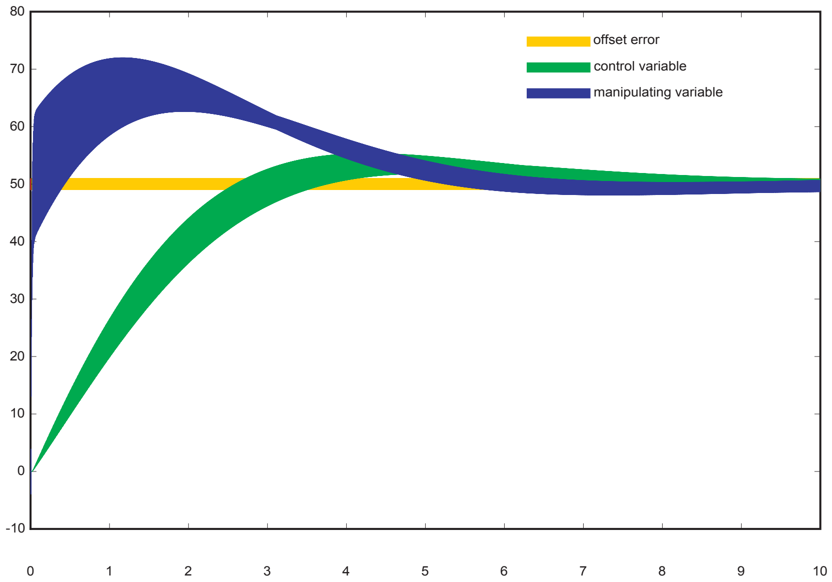


Figure 3.3: Visualization of simulation results [8]

for symbolic (forward) simulation of analog circuits which are modeled by nonlinear differential equations in the implicit form:

$$\mathbf{F}(\mathbf{x},\mathbf{p}) = \mathbf{0}.$$

So far this seems to be the only approach that is able to handle nonlinear differential equations in the implicit form of high complexity as needed for symbolic (forward) simulation. Circuit simulation is performed in two steps. In the first step, a net list of the circuit is transformed into the corresponding system of differential equations using the Modified-Node-Analysis (MNA). Then in the second step the equation system is solved in the numerical solver, which uses one of the methods of numerical integration like forward Euler, backward Euler or trapezoidal method to solve the given equations. The solver provides DC, AC and transient circuit analysis. To deal with affine terms, the circuit simulator performs the algorithm divided into three steps:

1. The nominal solution of the system is computed using a well-known Newton-Raphson method.
2. The equation is linearized at the nominal point with respect to parameter variations and variable inputs. For linear systems the simulator solves linearized equations providing an exact affine solution.
3. To deal with nonlinear systems, the algorithm requires the third step in which the affine solution is overapproximated to deliver the conser-

vative solution. For this purpose, the new deviation symbol (modeling overapproximation) is added.

Figure 3.4 shows the flow of affine circuit simulation.

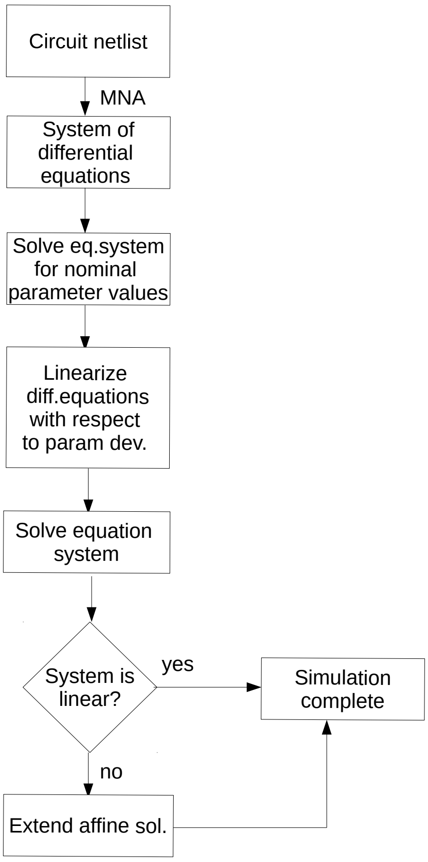


Figure 3.4: The flow of affine circuit simulation

3.2 Affine Arithmetic

Affine Arithmetic (AA) is a powerful technique to compute with ranges. It is proposed as an improved alternative for classical Interval Arithmetic (IA) [2] to handle its dependency problem. The form, which AA uses to represent an arbitrary quantity $\tilde{x}$, is as follows:

$$\tilde{x} = x_0 + \sum_{i=1}^{m} x_i \varepsilon_i \qquad \varepsilon_i \in [-1, 1] \,. \tag{3.1}$$

Each deviation symbol ε_i is a symbolic variable whose exact value is *unknown*, but lies in interval $[-1, 1]$. The deviation symbols are scaled by partial deviations x_i, whose number is equal to m.

The affine form given by Eq. 3.1 is an informal representation of a range $[x_{lo}, x_{up}]$ with lower and upper bounds calculated as:

$$x_{lo} = x_0 - \sum_{i=1}^{m} |x_i|$$
$$x_{up} = x_0 + \sum_{i=1}^{m} |x_i|.$$

The range is converted back to affine form according to the following:

$$[x_{lo}, x_{up}] = \frac{x_{lo} + x_{up}}{2} + \frac{x_{up} - x_{lo}}{2}\varepsilon; \varepsilon \in [-1, 1].$$

With symbolic representation Affine Arithmetic identifies the correlation between affine forms and hence delivers more tight and realistic bounds than Interval Arithmetic. Figure 3.5 shows the geometrical representation of joint range of two correlated quantities represented in Affine and Interval Arithmetic. In Interval Arithmetic joint range of x and y is a box, while in Affine Arithmetic the joint range represent a zonotope, a center-symmetric convex polytope. AA delivers much more accurate bounds than IA, as shown by Figure 3.5.

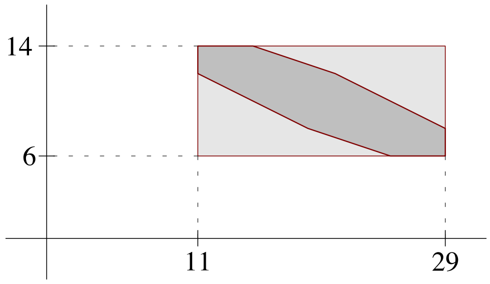


Figure 3.5: Joint range of dependent quantities $\tilde{x} = 20 - 4\varepsilon_1 + 2\varepsilon_3 + 3\varepsilon_4$ and $\tilde{y} = 10 - 2\varepsilon_1 + \varepsilon_2 - \varepsilon_4$ in Affine Arithmetic (zonotope) and Interval Arithmetic (box)[88]

3.2.1 Computation with AA forms

Arithmetic operations on affine forms can be divided into two groups: linear and nonlinear. These operations are defined through the following definitions. The acronym AAF in the following definitions refers to the set of affine forms defined by Eq. 3.1.

Definition 3.1. *If* $\tilde{x}, \tilde{y} \in AAF$ *and* $c \in \mathbb{R}$ *is a real constant then*

$$\tilde{x} + \tilde{y} = x_0 + y_0 + \sum_{i=1}^{m} (x_i + y_i)\varepsilon_i$$
$$\tilde{x} - \tilde{y} = x_0 - y_0 + \sum_{i=1}^{m} (x_i - y_i)\varepsilon_i$$
$$c\tilde{x} = cx_0 + \sum_{i=1}^{m} cx_i\varepsilon_i.$$

The above operations satisfy the closure property; they result in affine forms with exact lower and upper bounds. For nonlinear operations the final result must be approximated to be represented as an affine form. However, the safe inclusion is guaranteed; the final result is always over-approximation and never underestimation of the exact result.

Definition 3.2. *If* $\tilde{x}, \tilde{y} \in AAF$*, nonlinear function* $f(\tilde{x}, \tilde{y})$ *is then computed as:*

$$f(\tilde{x}, \tilde{y}) \subseteq f^a(\tilde{x}, \tilde{y}) + \delta\varepsilon_{m+1}$$

where $\varepsilon_{m+1} \in [-1, 1]$ is a new deviation symbol. δ is the upper bound of the approximation error between the nonlinear function f and the linear approximated function f^a:

$$|f(\tilde{x}, \tilde{y}) - f^a(\tilde{x}, \tilde{y})| \leq \delta \;\; \forall \varepsilon_i \in [-1, 1].$$

The linear approximated function f^a is computed using one of the approximation schemes described in the next section (Section 3.2.2). Nonlinear operations such as multiplication and division of two affine terms $\tilde{x}$ and $\tilde{y}$ are computed as follows:

$$\begin{aligned} \tilde{x} * \tilde{y} \;\; &:= \;\; x_0 * y_0 + \sum_{i=1}^{m} (x_0 y_i + x_i y_0)\varepsilon_i \\ &+ \;\; (\sum_{i=1}^{m} |x_i| \sum_{i=1}^{m} |y_i|)\varepsilon_{m+1} \end{aligned} \tag{3.2}$$

The multiplication result can be seen as the Taylor approximation of $\tilde{x} * \tilde{y}$ with the first order polynomial around the nominal values x_0 and y_0:

$$\begin{aligned} f^a(\tilde{x}, \tilde{y}) \;\; &= \;\; f(x_0, y_0) + \frac{\partial f}{\partial x}(x_0, y_0) * (\tilde{x} - x_0) + \frac{\partial f}{\partial y}(x_0, y_0) * (\tilde{y} - y_0) \\ &= \;\; x_0 * y_0 + \sum_{i=1}^{m} (x_0 y_i + x_i y_0)\varepsilon_i \end{aligned}$$

with the maximum approximation error:

$$\delta = \sum_{i=1}^{m} |x_i| \sum_{i=1}^{m} |y_i|.$$

The symbol ε_{n+1} is a new deviation symbol which together with z_m encloses the exact result. Division of two affine terms is computed as:

$$\frac{\tilde{x}}{\tilde{y}} = \tilde{x} * (\frac{1}{\tilde{y}})$$

with condition that the range modeled with $\tilde{y}$ does not contain zero. $\frac{1}{\tilde{y}}$ can be computed using one of approximation schemes described in the next section.

3.2.2 Approximation schemes of nonlinear operations

In contrast to linear, nonlinear operations do not satisfy the closure property. To keep affine form of a final result, the nonlinear part is replaced with the linear one using one of the approximation schemes. Without loss of generality, it is assumed that the computation is performed in one dimension. Let $\tilde{x}$ be the affine variable $\tilde{x} = x_0 + \sum_{i=1}^{m} x_i \varepsilon_i$ and f the nonlinear function applied on $\tilde{x}$. To get the affine form of z the nonlinear function is approximated with the linear one:

$$z = f(x_0 + \sum_{i=1}^{m} x_i \varepsilon_i) \subseteq f^a(x_0 + \sum_{i=1}^{m} x_i \varepsilon_i) + \delta \varepsilon_{m+1}. \tag{3.3}$$

The approximated function f^a is a linear function:

$$f^a(x_0 + \sum_{i=1}^{m} x_i \varepsilon_i) = k * (x_0 + \sum_{i=1}^{m} x_i \varepsilon_i) + n$$

where $k, n \in \mathbb{R}$. The values of k, n and the approximation error δ depends on the applied approximation scheme. Depending on the application the following approximation schemes are used:

1. *Minimum range approximation.* This approximation delivers the minimized range of the final result. This scheme is very useful for formal verification methods where a system response in the worst case is of high interest. The computed bounds which enclose all possible system behaviors should be as tight as possible since too high over approximation could lead to false negatives.
2. *Chebyshev approximation.* This approximation is very useful in the applications for which sensitivity analysis is of crucial importance. Although *minimum range approximation* provides the minimum over approximation on the computed bounds, the approximation error over the whole range is quite high which could lead to wrong interpreted effects of single sources of uncertainties to the total performance.

The Chebyshev approximation is computed according to the Lemma 1 that can be found in [96]. According to this lemma the coefficients of n-degree polynomial approximation of nonlinear function f over a range $[a, b]$ can be computed using $n + 2$ points:

$$a \leq x_1 < x_2 < ... < x_{n+2} \leq b.$$

for which it holds that

$$\delta = r(x_i) = -r(x_{i+1}) \tag{3.4}$$

where δ represents the maximum approximation error found as:

$$\delta = max_{x \in [a,b]} \left| f(x) - f^a(x) \right|$$

In Affine Arithmetic nonlinear functions are approximated with the first order polynomials:

$$f^a(\tilde{x}) = k * \tilde{x} + n$$

where $\tilde{x}$ is an affine term. Thus, the number of points used to compute the polynomial coefficients are three. The coefficient k is computed assuming function f is convex or concave over the interval $[a, b]$ $(a \neq b)$:

$$k = \frac{(f(b) - f(a))}{b - a}.$$

In that case two points x_1 and x_3 are the lower and upper bound of the interval $[a, b]$. The third point x_2 lies in the range $[a, b]$ and at this point the approximation error reaches the maximum value. So, the first derivative of the approximation error function $r^{'}(x_2)$ is equal to zero:

$$\begin{aligned} r^{'}(x_2) = f^{'}(x_2) - k = 0 \\ f^{'}(x_2) = k \end{aligned}$$

and the value x_2 is computed as:

$$x_2 = (f^{'})^{-1}(k).$$

When this value is found, Eq. 3.4 is used to find the value n of approximated function and the approximation error δ:

$$\begin{aligned} \delta = r(x_i) &= f(x_1) - k * x_1 - n \\ \delta = -r(x_{i+1}) &= k * x_2 + n - f(x_2). \end{aligned}$$

Now, n and δ are calculated as:

$$\begin{aligned} \delta &= \frac{f(x_1) - k * x_1 + k * x_2 - f(x_2)}{2} \\ n &= \frac{f(x_1) - k * x_1 + f(x_2) - k * x_2}{2}. \end{aligned}$$

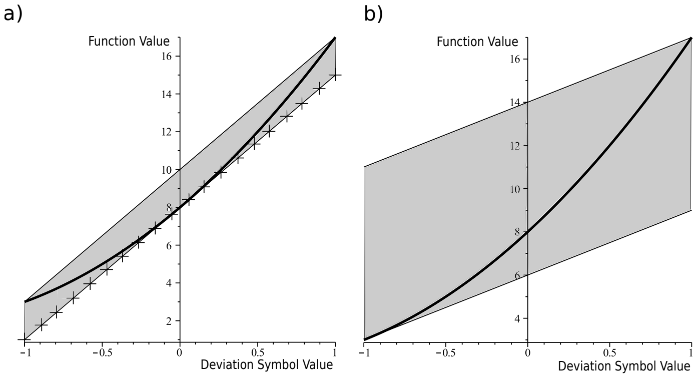


Figure 3.6: Linearization of nonlinear function using: a) Chebyshev b) Min-range approximation [97]

In contrast to Chebyshev approximation, the min-range approximation results in the function with the exact values on the bounds of the range $\tilde{x}$. However, inside the range the approximation error is higher than using Chebyshev approximation. The approximation function is computed as the tangent of function f at one end point. Hence, the approximation error at that point is equal to zero. If $f^{'}(x), f^{''}(x) \geq 0$, the tangent is computed at the lower bound $k = f^{'}(a)$, otherwise at the upper bound $k = f^{'}(b)$. The coefficient n and the approximation error δ are computed as:

$$\delta = \frac{f(a) - k * a + k * b - f(b)}{2}$$
$$n = \frac{f(a) - k * a + f(b) - k * b}{2}.$$

Figure 3.6 illustrates Chebyshev and min-range approximation of a nonlinear function (plotted by bold curve) over interval [-1, 1].

As given by 3.3 the approximation of nonlinear functions with linear ones introduces a new deviation symbol to ensure the safe inclusion. The new deviation symbol introduces overapproximation which contains exact computations. However, the performance of nonlinear operations in a loop or over a long simulation time horizon can lead to symbol explosion and loss of effectiveness of the approach. To attack this problem, literature survey proposes the following techniques:

1. *Cleanup* method proposed by [98] which servers as a garbage-like collection with the difference that new added deviation symbols are not removed from memory but replaced with only tow terms. More details are given in the following part of the section.
2. The other approach [99] combines Affine Arithmetic with Interval Arithmetic [2] where an approximation error is enclosed simply with

an interval. This form of Affine Arithmetic is known as Hansen's form Affine Arithmetic Form. The work in the thesis uses this approach. The advantage of Hansen's form over (1) is described in Section 3.4.

3.3 Implementation of *cleanup* method

As mentioned above, nonlinear functions must be approximated to satisfy the closure property of Affine Arithmetic. In order to ensure the safe inclusion, approximations add additional deviation terms. One deviation term is added every time step the nonlinear operation is performed. Thus, the maximum required memory for each affine variable depends linearly on the number of time steps:

$$u_d * n + u_s \subseteq O(n)$$

where u_d is the number of uncertainties added by nonlinear operations, n is the number of time steps and u_s is the number of static uncertainties.

Even worse, the run time complexity grows quadratic with the number of time steps n:

$$\begin{aligned} \sum_{i=1}^{n}(u_d i + u_s) &= u_d\frac{n(n+1)}{2} + u_s n \\ u_d\frac{n^2}{2} &+ (\frac{u_d}{2} + u_s)n \subseteq O(n^2). \end{aligned}$$

To deal with this problem, Heupke [98] proposes the *cleanup* method to dynamically manage a high number of terms. The idea is similar to one of the garbage collection with the difference that added deviation terms are not deleted form memory but replaced with only two terms. All deviation terms whose values lie below a user-defined level are summed up. One deviation term sums up all terms with positive scaling factors and the other with negative factor signs. In this way a safe inclusion is kept and the number of terms is significantly reduced. The *cleanup* method is called every m time steps. Since m is constant during simulation, the memory complexity becomes the same as the computation with pure numeric terms [98]:

$$u_d * m + u_s \subseteq O(1).$$

The run time complexity is also improved growing linear with the number of time steps [98]:

$$\begin{gathered} (u_d * m + u_s)n + 2n/m(u_d * m + u_s) = \\ (u_d * m + u_s + 2(u_d * m + u_s)/m)n \subseteq O(n). \end{gathered}$$

However, this method faces the following weak points:

1. It can be only applied on robust systems in a closed loop in which the impact of variations converges to values close to zero. If this is not the case, a threshold value for which the method should be called can hardly be specified.
2. The second weak point is that the moment in which a method is called is determined by a user. Hence, too often calls could lead to additional overapproximation and lead to incorrect behavior of a feedback loop. On the other hand, rare calls would lead to a high number of symbols that need to be handled by the method and the simulation process, itself. The computational cost would increase and the efficiency of the *cleanup* method would be lost.

3.4 Hansen's form of Affine Arithmetic

In contrast to the *cleanup* method, Hansen's form [99] applied on AA terms enables a constant time and space complexity of nonlinear operations in general. Each affine form given by Eq. 3.1 can be re-written using Hansens form as follows:

$$\tilde{x} = x_0 + \sum_{i=1}^{m} x_i \varepsilon_i + [0,0] \qquad \varepsilon_i \in [-1,1]$$

where Interval $[0,0]$ is used to represent a nonlinear part and its default value is *zero* interval in the case there are no nonafffine terms. Linear operations given by Definition 3.2.1 applied on Hansen forms of Affine Arithmetic result again in the Hansens forms. Also, in contrast to usual Affine Arithmetic, nonlinear operations of Hansen's affine forms result again in Hansen's form. The operations only update the bounds of interval enclosing the exact results. Thus, multiplication operation can be re-written:

$$\begin{aligned}
\tilde{x} * \tilde{y} &= (x_0 + \sum_{i=1}^{m} x_i \varepsilon_i + [0,0]) * (y_0 + \sum_{i=1}^{m} y_i \varepsilon_i + [0,0]) \\
&= x_0 * y_0 + \sum_{i=1}^{m} (x_0 * y_i + x_i * y_0)\varepsilon_i + \sum_{i=1}^{m} []_i
\end{aligned}$$

where $[]_i$ $i \in \{1, ..., m\}$ are computed as follows:

$$[]_i = \begin{cases} [\text{-}|x_i||y_i|,|x_i||y_i|] & i \neq j \\ [0,\ x_i y_i] & i = j, x_i * y_i > 0 \\ [x_i y_i, 0] & i = j, x_i * y_i < 0 \end{cases}$$

Example. Let $\tilde{x}$ has value $1 + \varepsilon_1 + [0,0]$ and $\tilde{y} = 1 - \varepsilon_1 + \varepsilon_2 + [0,0]$. Multiplication of $\tilde{x}$ and $\tilde{y}$ results in the following:

$$\tilde{x} * \tilde{y} = 1 + \varepsilon_1 * 0 + \varepsilon_2 + [-2,1]$$

where $[-2, 1]$ replaces quadratic terms and has value equal to sum of $[-1, 0]$ and $[-1, 1]$. The range $[-1, 0]$ encloses the values $-\varepsilon_1 * \varepsilon_1$ and $[-1, 1]$ the quadratic term $\varepsilon_1 * \varepsilon_2$. Note that in contrast to Eq. 3.2 no new deviation symbols were added. One more advantage of this approach that it also reduces overapproximation given by Eq. 3.2. The additional term is overapproximated with $[-2, 1]$ and not $[-2, 2]$.

Since no additional symbols are added, the memory consumed for Hansen's forms stays constant with increasing the number of simulation time steps:

$$u_s + 2 \subseteq O(1)$$

where u_s is the number of variations explicitly modeling uncertainties in a system and 2 places are reserved for lower and upper bound of the interval in Hansen's AA form. Time complexity as in numeric simulations changes linear with the number of simulation time steps n. In each time step $c*(u_s+2)$ need to be touched where c is the maximum number of operations applied on AA forms. Since c and u_s are constant the time complexity is linear:

$$\sum_{i=1}^{n} c * (u_s + 2) = c * (u_s + 2) * n \subseteq O(n).$$

3.5 Modeling parameter uncertainties with Affine Arithmetic

In [100] the term *uncertainty* is defined as "any deviation from the unachievable ideal of completely deterministic knowledge of the relevant system". Relying on this definition, the term *uncertainty* is used in this thesis to notify all kind of faults, unforeseen changes and variations that cause a circuit or a system to deviate from its ideal behavior. We classify *uncertainties* according to three major criteria: its *location*, its *modeling approach* and whether it is *static or dynamic.*

Table 3.1 gives three possible locations of uncertainties in mixed-signal designs: uncertainties on system inputs and stimuli, parameter uncertainties, and uncertainties in the system model itself. The uncertainties on inputs are typically caused by the lack of knowledge in initial operating conditions, or interactive scenarios.

Table 3.1: Classification by location

Location	**Examples**
input uncertainties	uncertain initial values or stimuli
parameter uncertainties	tolerances, component aging
modeling uncertainties	abstraction of accurate models

Parameter values often deviate from their values defined by design of a system. Parameter uncertainties are due to variations in a manufacturing process, which add certain tolerances to ideal parameter values. The other source of parameter uncertainties may be aging of system components causing parameter values to vary from its ideal values.

Modeling uncertainties are the result of abstraction of accurate “real” system behavior, which is often too complex to be analyzed.

Table 3.2 gives the classification of uncertainties according to modeling approaches. This chapter is reserved only for uncertainties with continuous values with uniform probability equal to one. This is because Affine Arithmetic in the form given by Eq. 3.1 can handle only these types of uncertainties. Uncertainties with discontinuous values or probabilities different than one require extension of AA to be modeled correctly. The extension of AAF towards modeling discontinuous uncertainties is the work proposed and implemented in this thesis. Its brief explanation is given in the next chapter.

Table 3.2: Classification by modeling approach

Modeling approach	**Examples**
continuous non-deterministic	tolerances, drift, aging
continuous probabilistic	white, colored noise
discrete non-deterministic	possible failure
discrete probabilistic	sporadic failure

The above classes of uncertainties can take two types of values: static and dynamic. This fact brings the third classification of uncertain values. Static uncertainties are uncertainties whose values are constant during simulation. Dynamic uncertainties add uncertainties that are not constant but change their values during simulation run. One way to model the dynamic nature of uncertain values is to add a new deviation symbol at each simulation time step.

In the following it is shown how these uncertainties can be modeled with Affine Arithmetic. For each uncertainty model a corresponding block diagram representation is given. This is a natural way of system modeling in hardware-description languages such as SystemC-AMS.

3.5.1 Static uncertainties

Static uncertainties add a constant deviation to an ideal system behavior. The set of static uncertainties includes:

- Tolerances of parameter values in system components due to variations in manufacturing process

- Uncertainties added due to lack of capturing accurate models of “real” behavior; accurate models are abstracted with a more general model in which the accurate behavior is included

Examples of static uncertainties and the way to model them using Affine Arithmetic is given below.

Gain uncertainty. A frequency domain behavior of e.g. an amplifier can be described with the following transfer function:

$$P(s) = K\frac{N(s)}{D(s)}, \tag{3.5}$$

where K is a gain whose value is defined by amplifier components. For amplifiers with an operational amplifier these components are among an input resistor and a resistor in a feedback loop. Due to tolerance values in system components, a gain value deviates from its nominal value. This can be modeled as an affine term:

$$K_{nom} + \varepsilon K_{dev} \qquad \varepsilon \in [-1, 1]$$

where K_{nom} and K_{dev} are a nominal gain value, and a maximum absolute variation from the nominal value, respectively. Deviation symbol $\varepsilon \in [-1, 1]$ is used to model all gain values whose variation is smaller than K_{dev}. Substituting gain variation in Eq. 3.5, $P(s)$ can be re-written as:

$$P(s) = P_{nom}(s)(1 + \varepsilon P_{dev}(s)) \qquad \varepsilon \in [-1, 1]$$

where $P_{nom}(s) = K_{nom}\ N(s)/D(s)$ is the transfer function of the ideal system block with the gain K_{nom} and $P_{dev}(s)$ the constant function modeling variation from the nominal behavior $P_{dev}(s) = K_{dev}/K_{nom}$. The block diagram of a system model including gain variation is shown in Figure 3.7.

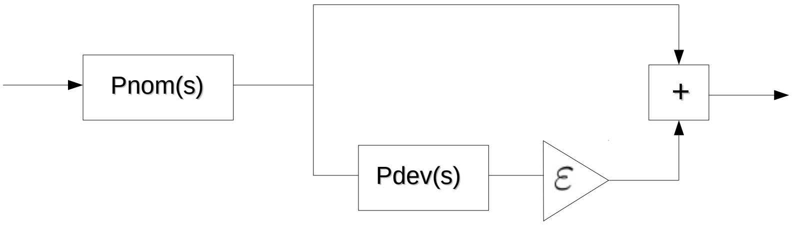


Figure 3.7: Block-level representation of gain uncertainty

Modeling (parameter) uncertainties. Beside gain, parameter values in a system transfer function are also the subject of variations. As a simple example, the following transfer function of a system block is assumed:

$$P(s) = \frac{1}{s^2 + as + 1}$$

The variation of parameter a due to variations such as tolerances of system components can be modeled as an affine form:

$$a_{nom} + \varepsilon a_{dev}, \qquad \varepsilon \in [-1, 1]$$

where a_{nom} represents the nominal value and a_{dev} the maximum absolute variation from the nominal value. Modeling parameter uncertainties on block level can be done transforming $P(s)$ into the form where deviated behavior is super-imposed to the nominal behavior:

$$P(s) = \frac{P_{nom}(s)}{1 + \varepsilon P_{dev}(s) P_{nom}(s)}, \qquad \varepsilon \in [-1, 1]$$

$P_{nom}(s)$ represents the nominal model with the parameter value a_{nom}:

$$P_{nom}(s) = \frac{1}{s^2 + a_{nom}s + 1}$$

and $P_{dev}(s)$ is the deviated function modeled as $P_{dev}(s) = a_{dev}s$. The representation on block level is shown in Figure 3.8.

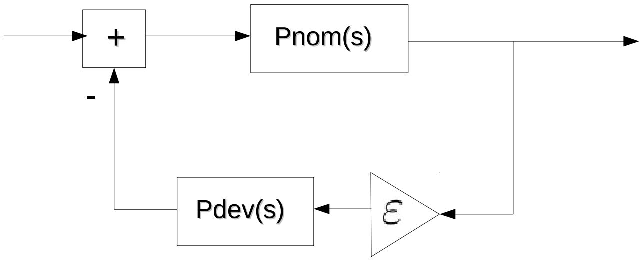


Figure 3.8: Block-level representation of parameter uncertainty

Abstraction of accurate model with Affine Arithmetic. Accurate models are often due to their nonlinearity and complexity very hard to simulate and verify. Thus, accurate behaviors are often abstracted with more simple models. These models include accurate behaviors and hence each specification met by the abstracted model will also be satisfied in the accurate one. Abstraction using Affine Arithmetic is demonstrated on a diode, a commonly used nonlinear element in many circuits. As an illustration, the abstraction will be performed only in the forward region, when a diode conducts electric current. The pair (I_D, V_D) represents DC operating point of a diode. A voltage range over which a diode operates is assumed to be symmetric around DC voltage V_D. This range is modeled using Affine Arithmetic as follows:

$$v_d = V_D + \varepsilon_1 \Delta v_d \qquad \varepsilon_1 \in [-1, 1] \, .$$

where Δv_d is the maximum absolute distance of operating voltage from DC operating point. The accurate diode behavior can be described with the following equation:

$$i_d = I_s(e^{\frac{v_d}{\eta V_T}} - 1).$$

The most simple way to get an abstract diode model is to linearize the accurate model with the first order Taylor polynomial around DC operating point (I_D, V_D):

$$\begin{aligned} i_d &= I_D + \frac{\partial i_d}{\partial v_d}(V_D)(v_d - V_D) + lin\_error \\ &= I_D + I_s e^{\frac{V_D}{\eta V_T}} \frac{1}{\eta V_T} \varepsilon \Delta v_d + lin\_error \qquad \varepsilon \in [-1, 1] \end{aligned}$$

The symbol *lin_error* assigns linearization error which is added to enclose the accurate model in the abstracted one. The absolute value of linearity error is equal to the maximum absolute value of the Lagrange remainder

$$\begin{aligned} |lin\_error| &= max(\frac{1}{2}\left|\frac{\partial^2 i_d}{\partial v_d^2}(\xi)\right|(v_d - V_D)^2) \\ &= max(\frac{1}{2}\left|\frac{\partial^2 i_d}{\partial v_d^2}(\xi)\right|)(\Delta v_d^2) \end{aligned}$$

where ξ can take any value from $\xi \in [V_D - \Delta v_d, V_D + \Delta v_d]$. To include the accurate model, linearization error is represented as:

$$lin\_error = \varepsilon_2 \, |lin\_error| \qquad \varepsilon_2 \in [-1, 1]$$

Figure 3.9 right shows the approximation of nonlinear diode (Figure 3.9 left) at the operating point.

A way of modeling time delay. Uncertainty caused by time delay can be static if a system block adds a constant time delay, or dynamic if a time delay changes its value during simulation, e.g., a jitter. A time delay added by an analog component such as a filter can be modeled in the frequency domain as $e^{-s*delay}$. If the *delay* depends on parameters whose variations are static (due to e.g. tolerances of parameter values), its value is also static. Let $[0, del\_max]$ be the range of possible *delay* values. The value 0 assumes there is no delay, while *del_max* is the maximum delay caused by variations in

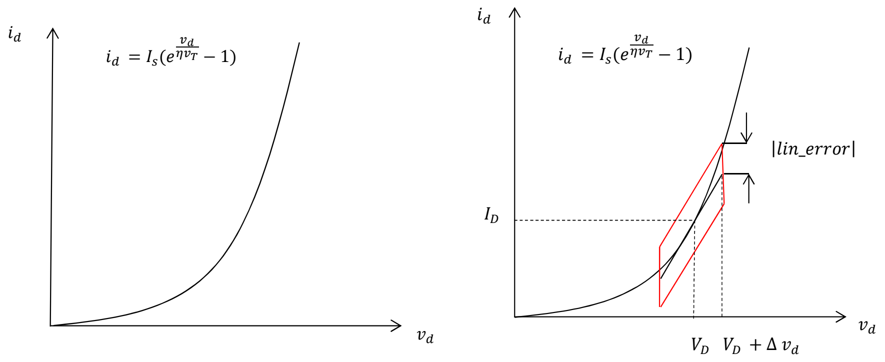


Figure 3.9: Forward diode characteristic; left: Accurate diode model right: Abstracted model at DC operating point

component parameters. An analog block with a time delay has the following transfer function:

$$P(s) = e^{-s*delay} P_{nom}(s) \tag{3.6}$$

where $P_{nom}(s)$ models the ideal behavior of the block (without time delay) and *delay* represents a time delay whose exact value lies in the interval $[0, del\_max]$. This range can be modeled using Affine Arithmetic. Delay values are usually small, maximum order of μs. For these small values $e^{-\tau s}$ can be approximated using the first-order Taylor polynomial around the nominal value 0:

$$\begin{aligned} e^{-s*delay} &:= 1 + \frac{(e^{-s*delay})'}{1!}|_{delay=0}\Delta del \\ &= 1 + (-s) * \Delta del \end{aligned} \tag{3.7}$$

Δdel represents variation from the nominal value, which lies in the interval $[0, del\_max]$. Substituting the approximation of $e^{-s*delay}$ (Eq. 3.7) in Eq. 3.6 $P(s)$ can be re-written as:

$$\begin{aligned} P(s) &:= P_{nom}(s)(1 + (-s) * (\frac{del\_max}{2} + \varepsilon\frac{del\_max}{2})) \\ &= P_{nom}(s)(1 + P_{dev}(s)) \end{aligned}$$

where $P_{nom}(s)$ models the ideal behavior and $P_{dev}(s)$ the deviation from the ideal behavior. The block-level representation of delay faults in mixed-signal systems is shown in Figure 3.10.

3.5.2 Dynamic uncertainties

Unlike static, dynamic uncertainties change their values during simulation. Typical examples for dynamic uncertainties are:

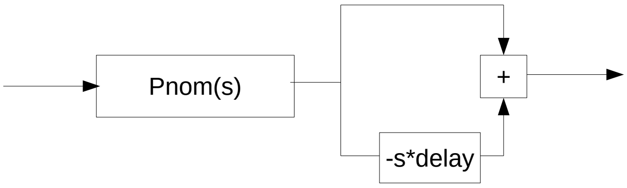


Figure 3.10: A system block with constant time delay

1. Round off errors in Digital Signal Processing due to limited number of bits used to represent numbers
2. Quantization errors caused by finite resolution in Analog/Digital (A/D) converters

Real numbers in DSPs can be represented in fixed or floating-point arithmetic. Due to a finite number of bits used to represents numbers, quantization errors are inevitable. In a DSP system there are two kinds of quantization errors: rounding and truncation error. Rounding is a process that rounds a number to the nearest level. Rounding to an integer value, the number gets the next higher integer value if the fractional part is ≥ 0.5 or the next lower integer if it is < 0.5. On the other hand, truncation is a process that assigns a value to the next lower value eliminating a certain number of least significant digits.

Fixed-point arithmetic. In fixed-point arithmetic a commonly used number representation has one bit for sign, the finite number of bits for the integer part m and the finite number of bits for the fractional part n. Rounding in fixed-point representation with n fractional bits adds the maximum quantization error 2^{-n-1} where 2^{-n} is the minimal representable number. Thus, the quantization error can range in $[-2^{-n-1}, 2^{-n-1}]$. Figure 3.11 illustrates rounding to a fixed-point number with 2 fractional bits.

Using Affine Arithmetic, the interval of rounding error can be modeled as:

$$\varepsilon[m] * 2^{-n-1} \quad \varepsilon[m] \in [-1, 1] \tag{3.8}$$

where n is the number of fractional bits and 2^{-n-1} the maximal value of rounding error. The dynamic nature of the rounding error is modeled by adding a new deviation symbol $\varepsilon[m]$ every time point m the rounding operation is performed.

Truncation adds the maximum error equal to the minimal representable number. In fixed-point arithmetic with n fraction bits the truncation error can be computed by subtracting the value 2^{-n-1} from the interval of the rounding error (Eq. 3.8). Hence, the truncation error ranges in $[-2^{-n}, 0]$ and its affine representation is:

$$-2^{-n-1} + \varepsilon[m] * 2^{-n-1} \quad \varepsilon[m] \in [-1, 1]$$

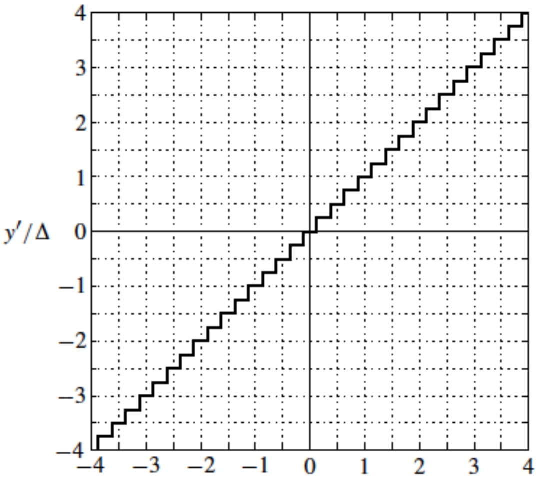


Figure 3.11: Rounding of a fixed-point representation with 2 fractional bits [101]

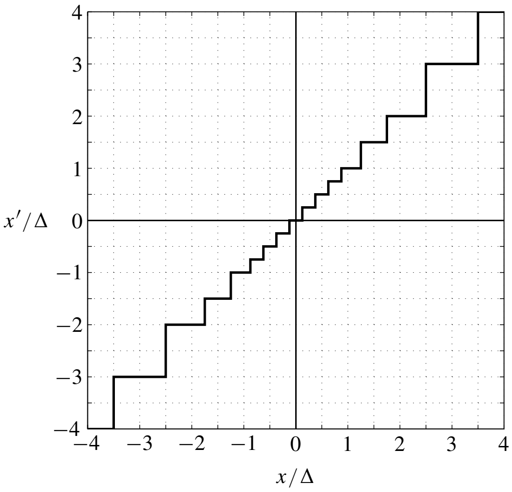


Figure 3.12: Rounding of a floating-point representation with 2 fractional bits [101]

Floating-point arithmetic. In contrast to fixed-point arithmetic the errors in the floating-point arithmetic depend on the value of the number to be represented. Figure 3.12 illustrates rounding of floating-point numbers with two fractional bits. The quantization errors are defined by the quantization step δ whose value depends on the number to be quantized. This value can be calculated as $2^n * q$ where n is the number of bits required to represent

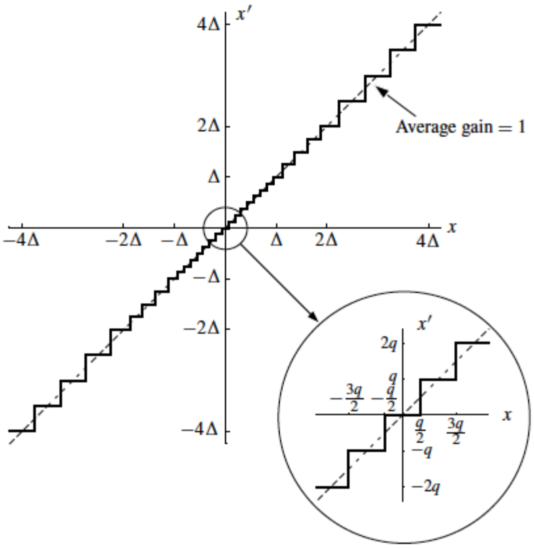


Figure 3.13: Quantization step and quantization error [101]

mantissa. The value q represents the minimal representable number in the corresponding floating-point representation.

Quantization. During conversion from analog to digital values, A/D converters introduce the certain value of quantization error. This is due to the converter finite resolution defined by the number of bits N used to represent the analog value v_{analog}. N-bit A/D converter adds the maximum quantization error $\frac{1}{2}LSB$ where LSB is $\frac{1}{2^N}$. The quantization error is uniformly distributed within the interval $[-\frac{1}{2}LSB, \frac{1}{2}LSB]$. Using Affine Arithmetic, this error can be easily represented as:

$$quant(v_{analog}, LSB) = v_{analog} + \varepsilon[m] * 2^{N-1} \quad \varepsilon[m] \in [-1, 1].$$

where $\varepsilon[m]$ defines a deviation symbol added at m time step.

Chapter 4

Extended Affine Arithmetic-XAA

The current form of Affine Arithmetic is limited to continuous operations with moderate nonlinearities. However, for Mixed-Signal Systems this is not sufficient. Beside continuous parts, discontinuous operations must also be handled. They are introduced by:

- discrete system parts such as comparator, switches, limiter, etc.
- software control flow statements (branches, loops)

For symbolic simulation, both are treated in a similar way: they are seen as a comparison of variables called *conditional variables.* For more clear explanation we will refer to the motivation example given in Section 1.1. Uncertainties in a PLL circuit lead to uncertainties on decisions in a phase frequency detector. Detection of the positive edge can be seen as comparison of the (Voltage Controlled Oscillator) VCO phase with 2π. If the phase is higher or equal to 2π the signal value is one, otherwise zero. Since uncertain values are enclosed with ranges, the computed system quantities will also be ranges. Thus, the comparison of phase values with 2π can result in `true`, or `false`, but also both $\{\texttt{false}, \texttt{true}\}$.

Figure 4.1 shows switching of the charge pump of the PLL circuit. PFD is activated on the phases of the reference signal ref and the VCO signal (divided by N) v. Here, it is assumed that the reference signal leads. In this case the reference signal reaches first 2π and the charge pump is on. When the phase of v signal reaches 2π the pump is switched off till the next cycle.

As shown in Figure 4.1, the comparison of the range of the v phase with 2π may lead to uncertainty in the switching of the charge pump. To cover both cases in one execution, this work proposes the extension of Affine Arithmetic-XAA. The relational operators are overloaded to support the comparison of ranges with specified thresholds. What follows in this chapter defines the XAA form (XAAF). We will also define arithmetic and relational operations which allow computation with XAAFs.

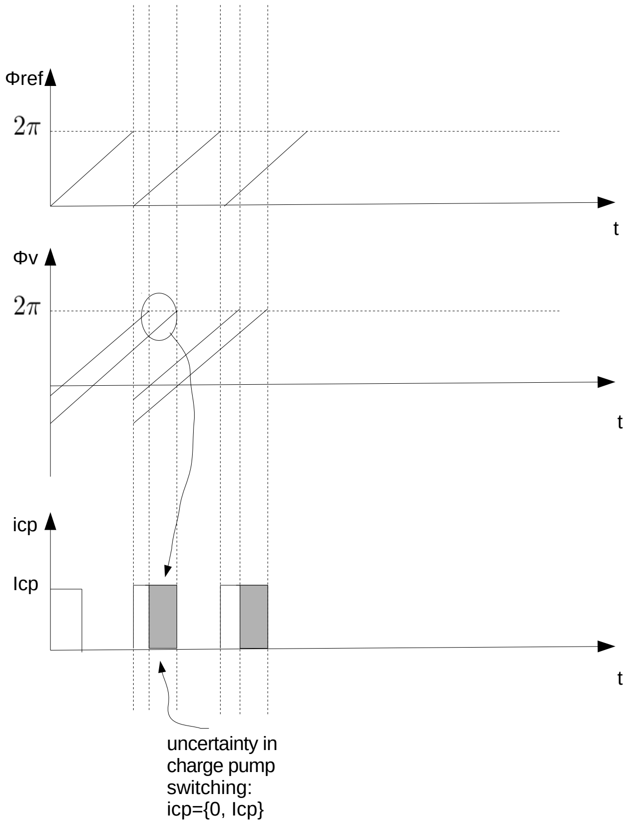


Figure 4.1: Charge pump activity

4.1 Definition and computation

With XAAFs uncertain results of relational operators are represented by symbols ω. As in Affine Arithmetic, where symbol ε captures unknown values lying in the range of $[-1, 1]$, ω represents unknown values from set $\{-1, 1\}$. The value $\omega = -1$ assigns `false` result of a relational operator, while $\omega = 1$ assigns `true`. An XAA form $\hat{x}$ is defined as follows:

$$\begin{aligned}\hat{x} &= \underbrace{x_0 + \sum_{i=1}^{m} x_i \varepsilon_i}_{\tilde{x}_0} + \sum_{k=1}^{n} \omega_k \underbrace{(x_{0k} + \sum_{i=1}^{m} x_{ik} \varepsilon_i)}_{\tilde{x}_k} \\ &= \tilde{x}_0 + \sum_{k=1}^{n} \omega_k \tilde{x}_k.\end{aligned}$$

where n represents the number of discrete modes covered by ω_k symbols and m the number of continuous uncertainties modeled with ε_i symbols. In the rest of the thesis ω_k symbols will be called *mode* symbols. $\tilde{x}_0$ represents the central range/polytope of the set around which mode symbols $\omega_k \in \{-1, 1\}$ merges ranges/polytopes each represented by AAF. The range $\tilde{x}_k$ represents the distance of each range/polytope from the central range $\tilde{x}_0$; x_{0k} represents the center of the range $\tilde{x}_k$ and x_{ik} the variations from the center of the range introduced by continuous uncertainties.

Computation with XAAFs is performed through overloaded arithmetic operators.

Definition 4.1. *If $c \in \mathbb{R}$ is a real constant and $\hat{x}, \hat{y}$ are two $XAAFs$ then*

$$\begin{aligned}\hat{x} \pm \hat{y} &= x_0 \pm y_0 + \sum_{i=1}^{m}(x_i \pm y_i)\varepsilon_i \\ &+ \sum_{k=1}^{n}\omega_k(x_{0k} \pm y_{0k} + \sum_{i=1}^{m}(x_{ik} \pm y_{ik})\varepsilon_i).\end{aligned}$$

$$c\hat{x} = c(x_0 + \sum_{i=1}^{m} x_i\varepsilon_i) + \sum_{k=1}^{n}\omega_k c(x_{0k} + \sum_{i=1}^{m} x_{ik}\varepsilon_i) \tag{4.1}$$

These operations are linear resulting again in $XAAFs$. As in Affine Arithmetic, non-affine operations require approximations. Analogous to Hansen's form, where non-affine terms are replaced with intervals, nonlinear terms in extended affine forms are modeled as sets. For instance, the multiplication operation is defined as follows.

Definition 4.2. *If $\hat{x}, \hat{y}$ are two $XAAFs$, then:*

$$\begin{aligned}\hat{x} * \hat{y} &= (\tilde{x}_0 + \sum_{k=1}^{n}\tilde{x}_k\omega_k) * (\tilde{y}_0 + \sum_{k=1}^{n}\tilde{y}_k\omega_k) \\ &= \tilde{x}_0 * \tilde{y}_0 + \sum_{k=1}^{n}(\tilde{x}_0 * \tilde{y}_k + \tilde{x}_k * \tilde{y}_0)\omega_k + \sum_{k=1}^{n}\{\}_k\end{aligned}$$

where $\{\}_k$ $k \in \{1, ..., n\}$ are computed as following:

$$\{\}_k = \begin{cases} \{\text{-}\tilde{x}_k\tilde{y}_j,\ \tilde{x}_k\tilde{y}_j\} & k \neq j \\ \tilde{x}_k\tilde{y}_j & k = j, (\omega_k\omega_j = 1) \end{cases}$$

However, only possible combinations of mode symbols are used for computation of approximation errors. Possible combinations are determined with LP (Linear Programming) solver used to eliminate over approximations, as explained in the following.

Relational operators are overloaded to support comparison of ranges which can result in `true`, `false` but also both $\{$`true`, `false`$\}$. For example, the operator $<$ is evaluated such that:

$$\hat{x} < \hat{y} = \begin{cases} true & : ub(\hat{x}) < lb(\hat{y}) \\ false & : lb(\hat{x}) \geq ub(\hat{y}) \\ \{false, true\} & : otherwise \end{cases}$$

$ub(\hat{x})$ and $lb(\hat{y})$ are total upper and lower bounds of $\hat{x}$ and $\hat{y}$, respectively. Using XAAF, `true` is represented with 1, `false` with 0, and unknown resp. $\{$`false`, `true`$\}$ with:

$$0.5 + \omega 0.5; \omega \in \{-1, 1\}.$$

The current implementation in this thesis is limited to the comparison with thresholds that are single values ($lb(\hat{y})) == ub(\hat{y})))$. In the rest of the work, the thresholds will be denoted with y.

The expression $\hat{x} < y$ is `true` only in the case that the whole set of values covered by $\hat{x}$ lies below the threshold given by y; $\forall x \in \hat{x}, x < y$.

Table 4.1 lists the overloaded relational operators. The only operators, that do not return XAAF, are operators $==$ and $!=$. The operator $==$ returns `true` only if $\hat{x}$ is equal to y, otherwise `false`. Since y is a single value, both bounds of $\hat{x}$ should be equal to y. The operator $!=$ is implemented as a simple negation of $==$.

Table 4.1: Evaluation of relational operators

Relational operator	Result
$\hat{x} < y$	**if** $(ub(\hat{x}) < y)$ **true** **else if** $(lb(\hat{x}) \geq y)$ **false else** $0.5 + \omega 0.5$
$\hat{x} \leq y$	**if** $(ub(\hat{x}) \leq y)$ **true** **else if** $(lb(\hat{x}) > y)$ **false else** $0.5 + \omega 0.5$
$\hat{x} > y$	**if** $(lb(\hat{x}) > y)$ **true** **else if** $(ub(\hat{x}) \leq y)$ **false else** $0.5 + \omega 0.5$
$\hat{x} \geq y$	**if** $(lb(\hat{x}) \geq y)$ **true** **else if** $(ub(\hat{x}) < y)$ **false else** $0.5 + \omega 0.5$
$\hat{x} == y$	**if** $(lb(\hat{x}) == y \; and \; ub(\hat{x}) == y)$ **true** **else false**
$\hat{x} \; != y$	negation of $==$ operator $!(\hat{x} == y)$

It is important here to mention that in the case that the condition results in $0.5 + \omega 0.5$, the set of values for which $\hat{x}$ lies below/above threshold are saved. They are saved in the form of constraints which are used for the next comparisons. In this way, lower and upper bounds of $\hat{x}$ are significantly improved.

Otherwise, the computed bounds would be over approximated and would lead to execution of cases, which can never happen in reality. The bound computation is defined as an LP problem subject to saved constraints and solved using an LP solver. For this purpose, the open source package GLPK (GNU Linear Programming Kit-glpk-4.45 [102]) is used.

4.2 Modeling uncertainties with Extended Affine Arithmetic

Section 3.5 gives a brief classification of uncertainties which can be modeled with Affine Arithmetic (AA). AA form models uncertainties in the continuous domain. However, propagation of continuous uncertainties through digital and software system parts can also cause uncertainties in the discontinuous domain. The following model covers both kinds of uncertainties with one single form:

$$\hat{x} = x_0 + \sum_{i \in ranges} x_i \varepsilon_i + \sum_{k \in modes} \tilde{x}_k \omega_k$$

- ε symbols are used to model uncertainties in the continuous domain
- ω symbols are used to model uncertainties in the discontinuous domain, e.g., non-determinism in choice between operating modes.

The probabilistic uncertainties mentioned in Section 3.5 are out of scope of this work.

4.3 Implementation of XAAF approach

The XAAF approach is implemented as an independent C++ library that provides the abstract type XAAF. Standard C++ arithmetic and relational operations are overloaded to allow computation with XAAFs. The implementation of *arithmetic* operators follow Definition 4.1. and 4.2. given in Section 4.1. Relational operations $<, \leq, >, \geq, ==, !=$ are overloaded according to Table 4.1.

The implementation of these operators consists of calling a private method, *compare*. The *compare* method is called with the following input parameters:

$$x.compare(y, out\_values)$$

where x is the left operand, y the right operand of the corresponding relational operator (e.g., $x < y$). The argument out_values specifies the vector of values that are returned by the operator. The possible return values for all relational operators are `false` and `true`. Thus, the specified out_values are 0 and 1. Which value is first specified depends on the relational operator.

The *compare* method compares x with the threshold y. If x values lie below the threshold y the compare method returns the first specified value from out_values. If x values are higher or equal than the threshold y the compare method returns the second specified output value. However, if x values cross the threshold y it returns both output values (0 and 1).

For example, the $<$ operator is implemented calling the compare method with the following arguments:

$$x.compare(y, \{1, 0\}).$$

The corresponding relational operator (in this case $<$) returns the result of the compare method. The value 1 is returned only in the case x values are smaller than y, ($x < y$). The value 0 is returned in the case $x \geq y$, otherwise $\{0, 1\}$. Table 4.2 summarizes the implementation of all relational operators.

Table 4.2: Implementation of relational operators

Relational operator	Implementation
$x < y$	return $x.compare(y, \{1, 0\})$
$x \leq y$	return $x.compare(y + 1e - 40, \{1, 0\})$
$x > y$	return $x.compare(y + 1e - 40, \{0, 1\})$
$x \geq y$	return $x.compare(y, \{0, 1\})$
$x == y$	if $(lb(x) == y \ \ and \ \ ub(x) == y)$ *true* else *false*
$x \mathrel{!}= y$	return $!(x == y)$

As shown in Table 4.2, the operators $\leq$ and $>$ call the compare method with slight higher values than the threshold y. The operator $\leq$ should return 1 (the first specified output value) if $x \leq y$. However, as previously described, the compare method returns the first output value only in the case $x < y$. To include y value in this case the compare method is called with slightly higher value than y ($y + 1e - 40$). A similar situation holds for $>$. In this case the compare method returns the second specified value (1) if it holds that $x \geq y$. To exclude the value y, the threshold value is slightly higher than y ($y + 1e - 40$). The implementation of the *compare* method is shown by Algorithm 4.1.

In the first step, the compare method checks the length of the XAAF x. The length of x is equal to the number of mode symbols. If the length is zero, the value x is a contiguous range given by Eq. 3.1. To compare the value x with the specified threshold, the lower and upper bounds are computed. This is done by setting ε symbols in Eq. 3.1 to -1 and 1. If the threshold value does not intersect the range x there is no need for merge; only one value should be returned (lines 6-9). However, if the range of x contains the threshold, the merge operation is called (lines 10-12). The set of satisfied conditions are saved and used for comparisons in the next steps. The satisfied conditions are used to overcome over approximation and compute the exact bounds for next comparisons.

Algorithm 4.1: Implementation of *compare* method

```
Input: XAAF value x, threshold $th$, corresponding output values $y = \{0, 1\}$ ($\{\texttt{false}, \texttt{true}\}$)
Output: false(0), true(1) or both
if length(x)==0 then
    Compute lower and upper bounds of $x$ $lb(x)$ and $ub(x)$
    Compare $x$ with the threshold th
    if $(ub(x) < th)$ then
        Return the first specified value in y
    else if $(lb(x) \geq th)$ then
        Return the second specified value in y
    else
        Save satisfied conditions $S$ $(x < th$ and $x \geq th)$
        Call merge
    end if
else
    Split x into $x_i$ based on previous satisfied conditions
    for each $x_i$ of x do
        repeat
        Compute $min\,\{max\}\,(x_i)$
        subject to previous satisfied conditions $S$
    end for
    Find total minimum and maximum of $x$ $lb(x)$ and $ub(x)$
    if $(lb(x) \geq th)$ then
        Return the first specified value in $y$
    else if $ub(x) < th)$ then
        Return the second specified value in $y$
    else
        for each $x_i$ of x do
            Compare $min\,\{max\}\,(x_i)$ with the threshold th
            if $(th > min(x_i)$ and $th < max(x_i))$ then
                Update the set $S$ with $x_i < th$ and $x_i \geq th$
            end if
            Save comparison result (required for merge operation)
        end for
        Call merge
    end if
end if

merge:
return result $0.5 + \omega_k 0.5$ with S and corresponding $\omega_k$
```

Hence, if the value x contains mode symbols, the bounds are computed subject to satisfied conditions (line 15-19). The bound computation is formulated as an optimization problem. The problem is solved subject to the set of conditions satisfied in the previous computation step. For this purpose, the open source package GLPK (GNU Linear Programming Kit-glpk-4.45 [102]) is used.

To check if the whole range lies below/above th, the total minimum and maximum are found. If the total minimum is higher than th or total maximum is below th, the threshold does not intersect x and the corresponding output is returned (lines 21-24). If this is not the case, each of the computed bounds are compared with the specified threshold th (lines 26-34). If x_i crosses th both satisfied conditions are saved ($x_i < th$ and $x_i \geq th$) (lines 29-30). In the last step the merge is called (line 34).

The merge operation merges both possible output values using a mode symbol ω. The index of ω depends on the comparison results performed in each mode. Three cases are possible:

1. Condition values for each mode are same as in the previous time step; there are no further splittings and the number of non-contiguous regions stays the same. In this case the constraints for the next computations also stay the same, and the same ω_k symbol is used to represent the set $\{\texttt{false}, \texttt{true}\}$.
2. Condition values for each mode are inversion of the condition values in the previous time step; there are no further splittings and the number of non-contiguous regions stays the same. The set of constraints also stays the same and the same ω_k symbol but with negative sign ($-\omega_k$) is used.
3. Condition values result again in $\{\texttt{false}, \texttt{true}\}$ and a new mode symbol must be used to cover further splittings. The set of constraints is updated.

4.4 Code Modification with XAAF

In the semantic of standard programming languages a conditional statement has two branches: one for *true* (if …), and for $false$ (else …). However, in symbolic simulation with XAAF we have to consider a third case: $X = \{\texttt{false}, \texttt{true}\}$. Then, both possible branches of a control flow must be handled. In the following we give two simple re-writing rules to solve this problem.

Rule 1 Conditions in control flow statements are Boolean type (`true`, `false`). Expressions under conditions will execute for `true` condition values. However, comparison of ranges represented with XAAFs can also result in X (unknown). To cover the last case, the condition value must explicitly be compared with `false`:

$$\texttt{cond} \;\rightarrow\; \texttt{cond!} = \texttt{false}.$$

Rule 2 If we have applied Rule 1, the *true* branch must cover the case where the condition is X and *true*. If the condition is X the expression must have both values, one for $\texttt{cond} = \texttt{true}$ and $\texttt{cond} = \texttt{false}$. This can be obtained by modifying the expression with Shannon's expansion:

$$\texttt{lvalue} = \texttt{expr} \;\rightarrow\; \texttt{lvalue} = \texttt{cond} * \texttt{expr}+!\texttt{cond} * \texttt{lvalue}$$

where the first part computes the expression value for $\texttt{cond} = \texttt{true}$ and the second for $\texttt{cond} = \texttt{false}$.

In the XAAF library the set of condition values $X =$`{false, true}` is represented with XAAF:

$$0.5 + \omega 0.5; \omega \in \{-1, 1\}$$

where $\omega = -1$ represents `false`, and $\omega = 1$ represents the `true` condition value. Due to the symmetric property of XAAF, negation of the condition value $0.5 + \omega 0.5$ only changes the sign of the value which is multiplied with ω. Thus, the following holds:

$$!(0.5 + \omega 0.5) = 0.5 - \omega 0.5; \omega \in \{-1, 1\}\,.$$

This explains why in this work ω is chosen to be $\{-1, 1\}$ instead of $\{0, 1\}$. Hence, using XAAF library `lvalue` gets the value:

$$\texttt{lvalue} = (\mathsf{0.5} + \omega * \mathsf{0.5}) * \texttt{expr} + (\mathsf{0.5} - \omega * \mathsf{0.5}) * \texttt{lvalue}$$

In this way for `false` condition value ($\omega = -1$) the expression keeps its value while for `true` ($\omega = 1$) its value is updated to *expr*.

In the following we show how these rules are used to modify conditional and iteration statements written in C/C++.

4.4.1 Conditional statements

Figure 4.2-4.4 show how `if`, `if-else` and `if-else if` are modified to handle ranges as conditional variables.

```
if (cond)
 lvalue=expr;
```

```
if (cond!=false)
 lvalue=cond*expr+!cond*lvalue;
```

Figure 4.2: IF conditional statement

```
if (cond)
 lvalue=expr1;
else
 lvalue=expr2;
```

```
if (cond!=false)
 lvalue=cond*expr1+!cond*expr2;
else
  lvalue=expr2;
```

Figure 4.3: IF-ELSE conditional statement

```
if (cond1)
 lvalue=expr1;
else if (cond2)
 lvalue=expr2;
```

```
if (cond1!=false)
 lvalue=cond1*expr1+!cond1*lvalue;
if (cond2!=false)
 lvalue=cond2*expr2+!cond2*lvalue;
```

Figure 4.4: IF-ELSE IF conditional statement

Figure 4.2 shows the conditional statement with only one `if` branch. Since `if` branch should be executed only if the condition `cond` is `true`, we simply need to apply Rule 1 checking if the condition is not equal to `false`. Following Rule 2 the expression should be modified as:

$$\texttt{lvalue} = \texttt{cond} * \texttt{expr}+!\texttt{cond} * \texttt{lvalue}.$$

For this example the expression value gets the new value only if the condition is `true`; for `false` value of the condition the expression value must stay the same (there is no `else` branch). Hence, negation of the condition !`cond` is multiplied with the old expression value (before `if` statement).

Note that for the `true` condition value the `if` condition is the same as on the left side (!$\texttt{cond} * \texttt{lvalue} = \texttt{0}$).

The similar modification can be done for `if-else` statement shown in Figure 4.3. The only difference is that the expression value is also updated for `false` condition value. To check if the condition results in X =`{false, true}`, we apply Rule 1 and compare if `cond` is different than `false`. According to Rule 2 the expression is computed according to:

$$\texttt{lvalue} = \texttt{cond} * \texttt{expr1}+!\texttt{cond} * \texttt{expr2}.$$

As already explained, the first part of the expression executes the `if` branch of the statement, and the second part the `else` branch of the statement. Since `else` branch will execute only if `cond` is `false`, the rest of the conditional statement stays the same (no rules are required).

Figure 4.4 demonstrates the modification of a conditional statement, which contains more branches. As in the first `if` condition each branch should be executed only if the corresponding condition is `true`. Therefore, here we also check if the conditions are not equal to `false` applying Rule 1. E.g., in the first branch, the expression should update its value only if `cond1` results in `true`. The expression is then computed in the same way as in Figure 4.2 with the corresponding condition:

$$\texttt{lvalue} = \texttt{cond1} * \texttt{expr1} + !\texttt{cond1} * \texttt{lvalue}.$$

The same holds also for the second branch. Here also, if the conditions result in `true` the conditional statement is the same as on the left side.

4.4.2 Iteration statements

There are various variants of iterations; we will only discuss here a `while` statement; other iteration types (`do while`, `repeat until`) can be implemented in a similar way.

Figure 4.5 demonstrates the use of XAAF data type in iterations such as `while loops`. Applying Rule 1 it is checked if `cond` is not `false`. Similar to `if` statements, the expression value is also computed using Shannon's Expansion Formula:

```
while (cond)
   lvalue=expr;
```

```
while (cond!=false)
 lvalue=cond*expr+!cond*lvalue;
```

Figure 4.5: WHILE loop

$$\texttt{lvalue} = \texttt{cond} * \texttt{expr} + !\texttt{cond} * \texttt{lvalue}$$

The modified `while loop` also covers the case for which `cond` is `true`. In this case `while loop` is the same as on the left side.

4.5 Scalability of symbolic simulation with XAAF

In Extended Affine Arithmetic there are two kinds of symbols:

- ε *symbols*. These symbols are introduced to model system uncertainties as contiguous ranges. They can be static or dynamic. The values of static symbols do not change over time. Dynamic symbols change their values and hence every simulation time step a new symbol is added. Dynamic uncertainties are usually used to model non-deterministic nature of noise for which probabilities are more realistic models than ranges. Probabilistic uncertainties are out of scope of this work.
- ω *symbols*. Comparison of uncertain values can include both comparison values $\{\texttt{false}, \texttt{true}\}$ within the same range. These symbols are used to cover both cases in one execution.

Space complexity. Let u be the maximum number of uncertainties added in the elaboration phase. The number of simulation time steps will be assigned with n. One more factor that determines the XAAF complexity is the number of mode symbols. They identify the possible set of discrete modes in which a system may operate. They are introduced by comparisons of uncertainties with thresholds in discrete parts of a system. The maximum number of comparisons is assumed to be k. Let c be the maximum number of operations performed on extended affine forms. The maximum memory and time complexity occurs in the case a new mode symbol is added each time step: The memory complexity is determined with:

$$c * (u_s + 2 + \underbrace{k * n}_{added\ \omega\ symbols}) \subseteq O(n)$$

where 2 is the memory occupied for the lower and upper bound of approximation error. Since all other parameters are constant the memory space grows linear with the number of simulation time steps n.

Run time complexity. The run time complexity grows in the worst case exponentially with n:

$$\begin{aligned} c * (u_s + 2) * k \sum_{i=1}^{n} 2^i = \\ = c * (u_s + 2) * k * 2 * (2^n - 1) \subseteq O(2^n) \end{aligned}$$

However, in reality the above computed worst case complexity is too pessimistic. Dynamics in system behavior introduce certain correlations of transitions between different discrete modes. The integrated LP solver, that computes the exact bounds of non-contiguous regions, identifies these modes. The correlations, that can reduce the number of mode symbols significantly, are taken into account. Hence, the number of mode symbols is usually lower than the number of simulated time steps n. This will be shown applying the methodology on the case studies described in Chapter 6.

4.6 Illustration examples

The first example shows the execution of a control flow statement with XAAF. The second example demonstrates the application of XAAF to a simple hybrid system such as a water-level control system.

4.6.1 Example 1 - Control flow example

Figure 4.6 shows a small control-flow with two *if* conditional statements. The right side of the figure shows the instrumentation of C program for the case that x is a range. The instrumentation is done according to the rules explained in Section 4.4. The execution of the control flow statement using XAAF library is explained in the following.

1. Step - Execution of the first `if` condition. The first condition compares a condition variable x with one. Comparing $x = 1 + \varepsilon_1 = [0, 2]$

```
double x=1;
// x is a single number

if (x>1)
  x=x-4;
else
 x=x+4;

x=x*2;

if (x>2)
  x=x+1;
```

```
XAAF x=1+AAInterval(-1., 1.);
// x is a range [0, 2]

if ((x>1)!=false)
  x=(x>1)*(x-4)+!(x>1)*(x+4);
else
 x=x+4;

x=x*2;

if ((x>2)!=false)
  x=(x>2)*(x+1)+!(x>2)*x;
```

Figure 4.6: Control flow example

with one, the condition $x > 1$ for one set of ε_1 values results in `true` and the other in `false`. Hence, the operator $>$ returns the XAAF, which covers both values $\{$`false`, `true`$\}$:

$$0.5 + \omega_1 0.5; \omega_1 \in \{-1, 1\}.$$

As explained in Section 4.3 the operator $>$ in this case additionally saves the ε_1 values for which the condition was `true`/`false`. They are saved in the form of constraints. Two constraints are saved. For the `true` condition value ($\omega_1 = 1$), the operator $>$ saves the constraint $1 + \varepsilon_1 > 1$. For the `false` condition value ($\omega_1 = -1$), the operator saves the constraint $1 + \varepsilon_1 \leq 1$. These constraints are used for improving the bounds of the new value of x, required for the next executions of relational operators. This will be shown in the following. The new value x results in:

$$\begin{aligned} x &= (0.5 + \omega_1 * 0.5) * (x - 4) + (0.5 - \omega_1 * 0.5) * (x + 4) \\ &= (1 + \varepsilon_1) - \omega_1 * 4; \omega_1 \in \{-1, 1\} \end{aligned} \tag{4.2}$$

where for $\omega_1 = -1$ its value is $5 + \varepsilon_1$ and $\omega_1 = 1$ the value of x is $-3 + \varepsilon_1$.

2. Step - Multiplication with a scalar coefficient. In the next instruction the new value of x is multiplied with a scalar coefficient 2. Applying Eq. 4.1 to Eq. 4.2 with $c = 2$ we get:

$$x = 2 * x = 2 * (1 + \varepsilon_1 - \omega_1 * 4) = 2 + 2\varepsilon_1 - \omega_1 * 8.$$

3. Step - Execution of the second `if` condition. In the next `if` statement, x is compared with 2. Now, since x contains ω_1 symbol, the operator $>$ finds the bounds of x computing the bounds of AAFs for each value of the previous condition. The AAF values are extracted by setting ω_1 to -1 and 1. In the following, x_1 will assign the AAF value for $\omega_1 = -1$ and x_2

the AAF value for $\omega_1 = 1$:

$$\begin{aligned} x_1 = 2 + 2\varepsilon_1 - \omega_1 * 8|_{\omega_1=-1} = 10 + 2\varepsilon_1 \\ x_2 = 2 + 2\varepsilon_1 - \omega_1 * 8|_{\omega_1=1} = -6 + 2\varepsilon_1 \end{aligned}$$

The lower and upper bounds of x_1 can be easily computed setting ε_1 to -1 and 1. However, this would lead to over approximation of the upper bound since for $\varepsilon_1 = 1$ does not satisfy the constraint $1+\varepsilon_1 \leq 1$; it does not belong to the set of ε_1 values for which the previous condition is `false`. To reduce over approximation and compute exact bounds, the bound computation is formulated as an LP (linear programming) problem and solved as such. The LP problem is defined as:

$$min(max)x_1 = min(max)(10 + 2\varepsilon_1)$$

subject to the previously saved constraint (`false` condition) and bounds of the variable ε_1:

$$1 + \varepsilon_1 \leq 1 \text{ and } -1 \leq \varepsilon_1 \leq 1.$$

LP routine returns the lower bound value 8 computed for $\varepsilon_1 = -1$. The upper bound is however computed for $\varepsilon_1 = 0$ and not $\varepsilon_1 = 1$. The computed value is 10. Comparing the computed bounds with 2 the condition for AAF x_1 is `true`. Note that for ε_1 values for which the previous condition was `false` $(1 + \varepsilon_1 \leq 1)$ the new condition is `true`.

The lower and upper bounds of x_2 are computed in a similar way. The difference is that the bounds are found subject to `true` condition $(1+\varepsilon_1 > 1)$:

$$min(max)x_2 = min(max)(-6 + 2\varepsilon_1)$$

The lower and upper bounds are found to be -6 and -4. Comparing these bounds with 2 the condition for x_2 is `false`. Here, it can also be noted that the new condition is `false` for ε_1 values for which the previous condition was `true` $(1 + \varepsilon_1 > 1)$.

The operator $>$ takes this fact into account and returns:

$$0.5 - \omega_1 * 0.5,$$

which is actually the negation of the previous condition. Now for $\omega_1 = -1$ the condition is `true` $(x_1 > 2)$ and for $\omega_1 = 1$ the condition is `false` $(x_2 < 2)$, as it should be.

The new value of x becomes:

$$\begin{aligned} x &= (0.5 - \omega_1 * 0.5) * (x + 1) + (0.5 + \omega_1 * 0.5) * x \\ &= 2.5 + 2\varepsilon_1 - \omega_1 * 8.5. \end{aligned}$$

Here, the operator $>$ did not update the set of constraints with new constraints since the bounds of each computed AAFs did not cross the value 2; the set of ε_1 values for each condition value stayed the same.

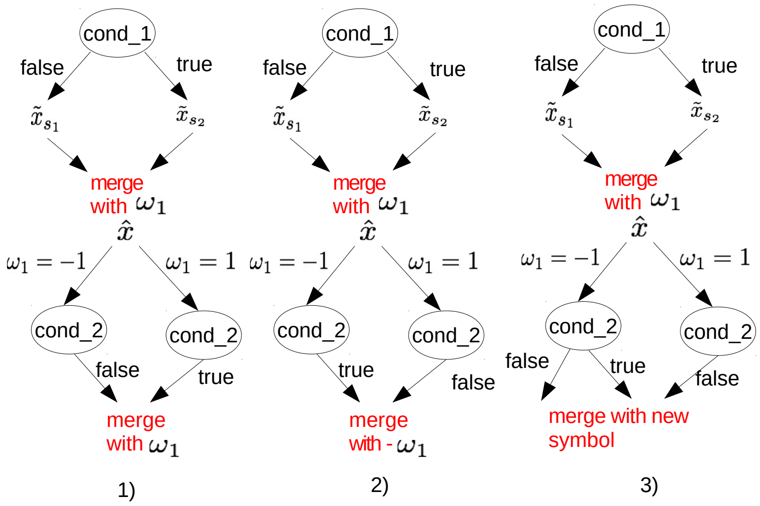


Figure 4.7: Illustration of merge cases

In general, considering the correlation between results of previous and current conditions, relational operations distinguish the following cases:

1. The current condition is `false`/`true` for the set of uncertain values for which the previous condition was `false`/`true`; the relational operators use the same mode symbol
2. The current condition is `true`/`false` for the set of uncertain values for which the previous condition was `false`/`true`; the relational operators use the same mode symbol with a negative sign (shown by example).
3. The current condition results in {`false`, `true`} for the set of uncertain values for which the previous condition was `false`/`true`; the relational operators adds a new mode symbol

Figure 4.7 illustrates all three cases.

4.6.2 Example 2 - Water level control system

As the second example a simple water level control system is chosen. It is shown in Figure 4.8. The state machine in the figure defines the principle of the control algorithm. For safety reasons, the water level W shall remain between the lower and upper limits ($ll \leq W \leq ul$). A (simple) control algorithm is as follows:

When the water level W exceeds ul, the monitor turns off the pump (U_1). The water in the tank drains at the rate of *outrate*. When the water level W is below ll, the monitor turns on pump U_1. The tank is filled at the rate of *inrate*. The continuous dynamic of the water tank can be described by the following linear differential equation:

$$\dot{W} = inrate - outrate. \tag{4.-2}$$

The functionality of the water level control system seems to be simple, but

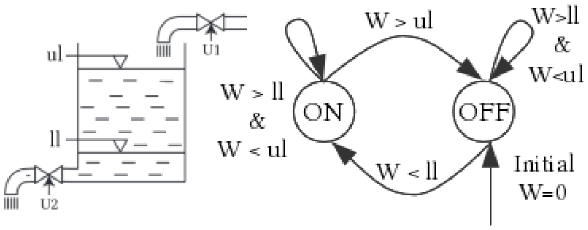


Figure 4.8: Functional model of a water level control system.

different kinds of uncertainties in the implementation require validation. First, the software is executed in a discrete-time way. Second, the accuracy of the sensors and rates may be uncertain. For a simple illustration let us assume $inrate = 1 \pm 10\%$, $outrate = 2 \pm 10\%$, and thresholds $ll = 5, ul = 10$.

For modeling and simulation SystemC AMS is used. A block diagram of a system modeled in SystemC AMS is shown in Figure 4.9. It is composed of the following parts:

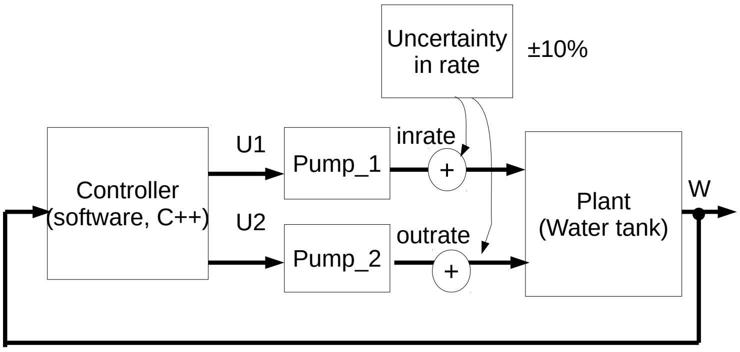
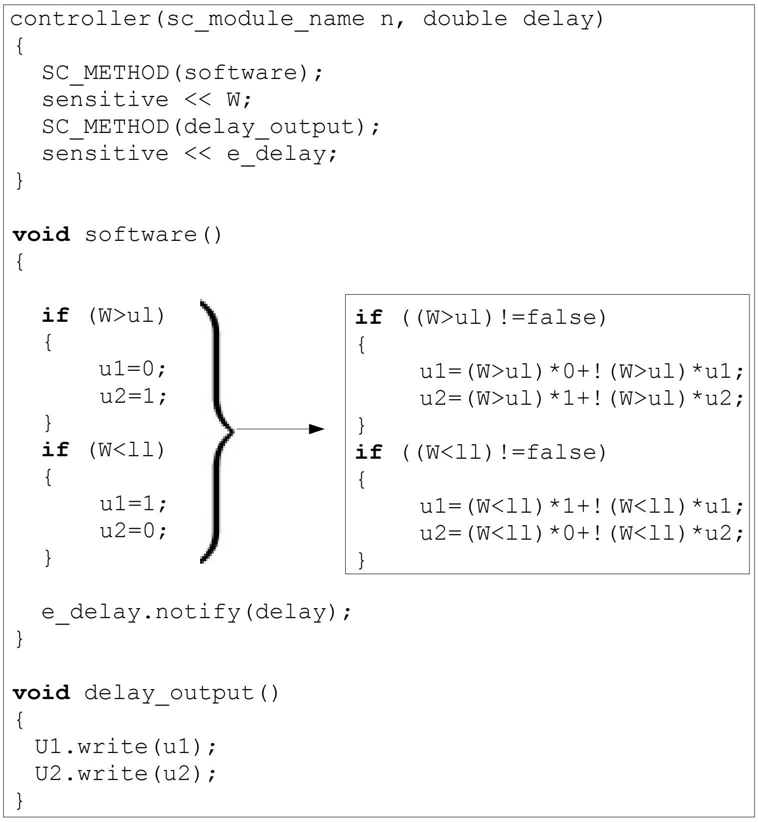


Figure 4.9: Block diagram of a water level control system.

- a controller which controls the water level in the tank
- two pumps U_1 and U_2 which are turned on/off by the controller
- a plant model which represents the water tank
- an uncertainty which models inaccuracy in *inrate* and *outrate*

The controller controls the water level W in the tank comparing its value with the thresholds, ll and ul. A controller is implemented as a simple SystemC module with thread function called *software*.

The thread function implements the controlling process. The time for the control function to turn on/off the pumps is notified by the delay event e_delay. The source code of the *software* function is given by Figure 4.10. For symbolic simulation, XAAF Abstract Data Type (ADT) is used as data type for $W, inrate, outrate$ and intermediate results. The computed water level W is not a single value but a set of values. Therefore, the conditions in the `if` branches can result in `true`, `false` but as way as $\{$`false`, `true`$\}$. Thus, the `if` statements are modified applying Rule 1 and 2. For `false` condition, the control signals $U1$ and $U2$ keep their values. For the `true` condition value, they change their values to $U1 = 0/U2 = 1$ in the first `if` statement and $U1 = 1/U2 = 0$ in the second `if` statement. Applying Rule 2, U_1 and U_2 are modified to cover the case for which conditions may result in $\{$`false`, `true`$\}$.



```
controller(sc_module_name n, double delay)
{
  SC_METHOD(software);
  sensitive << W;
  SC_METHOD(delay_output);
  sensitive << e_delay;
}

void software()
{

  if (W>ul)
  {
      u1=0;
      u2=1;
  }
  if (W<ll)
  {
      u1=1;
      u2=0;
  }

  e_delay.notify(delay);
}

void delay_output()
{
 U1.write(u1);
 U2.write(u2);
}
```

```
if ((W>ul)!=false)
{
    u1=(W>ul)*0+!(W>ul)*u1;
    u2=(W>ul)*1+!(W>ul)*u2;
}
if ((W<ll)!=false)
{
    u1=(W<ll)*1+!(W<ll)*u1;
    u2=(W<ll)*0+!(W<ll)*u2;
}
```

Figure 4.10: Source code of the control function

$$U_1 = (W > ul) * 0 + !(W > ul) * U1$$

$$U2 = (W > ul) * 1 + !(W > ul) * U2$$

The same holds for the second `if` statement with the difference that for *true* conditions U_1 is 1 and U_2 is 0.

$$U_1 = (W < ll) * 1 + !(W < ll) * U1$$

$$U2 = (W < ll) * 0 + !(W < ll) * U2.$$

The pump models are implemented as SystemC modules with *processing* function sensitive to the control signals U_1/U_2 . The source codes of pump models are given in Figure 4.11. For symbolic simulation, the codes are modified by applying Rules 1 and 2.

```
pump_u1(sc_module_name n, double rate)
{
  SC_METHOD(processing);
  sensitive << U1;
}

void processing()
{
  if (U1==1)
    inrate=rate;
  else
    inrate=0;
}
```

```
if (U1!=0)
 inrate=U1*rate+!U1*0;
else
 inrate=0;
```

```
pump_u2(sc_module_name n, double rate)
{
  SC_METHOD(processing);
  sensitive << U2;
}

void processing()
{
  if (U2==1)
    outrate=rate;
  else
    outrate=0;
}
```

```
if (U2!=0)
 outrate=U2*rate+!U2*0;
else
 outrate=0;
```

Figure 4.11: Source codes of the pumps

The plant in the model presents the tank whose behavior can be described with Eq. 4.6.2. Here, we implement it using SystemC AMS TDF (Timed Data Flow) model of computation. The *processing()* function in TDF computes the water level in discrete time steps.

$$W(n * T_s) = W((n-1) * T_s) + (inrate - outrate) * T_s$$

```
// defines old and new values of water level
XAAF old_W, new_W;

void processing()
{
  new_W=old_W+(inrate-outrate)*get_timestep().to_seconds();
  old_W=new_W;
  W.write(new_W); // W gets the current value
}
```

Figure 4.12: Source code of the tank

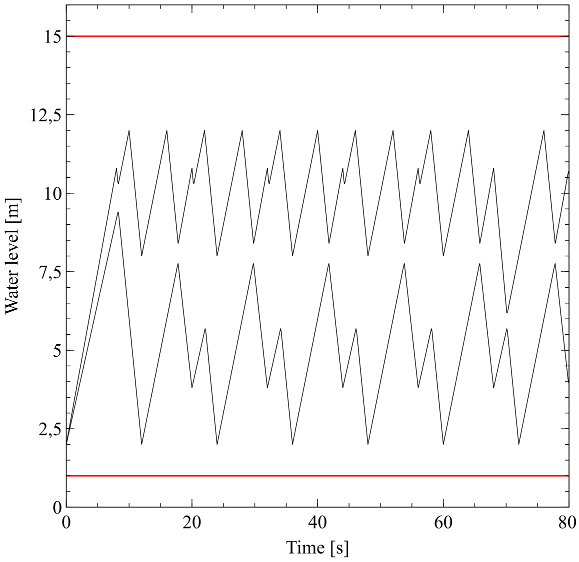


Figure 4.13: Possible water levels in the tank.

where T_s is a sampling period. The source code is given in Figure 4.12.

The symbol *old_W* is assigned to the water level at the previous time step, $(n-1)*T_s$, and *new_W*, the new value at current time $n*T_s$. For example, the following uncertainties are assumed:

1. Physical quantities, e.g., rates *inrate* and *outrate*, are measured/manufactured inaccurately due to an arbitrary reason.
2. The control function needs time on a processor e.g., T =2 ms for situations like delay in scheduling of processes.

The maximum inaccuracy of rates is assumed to be 10%. Using XAAF library this is modeled as:

$$inrate = inrate * (1 + AAInterval(-0.1, 0.1))$$
$$outrate = outrate * (1 + AAInterval(-0.1, 0.1))$$

which is equivalent to $inrate * (1 + \varepsilon_1 * 0.1)$ and $outrate * (1 + \varepsilon_2 * 0.1)$, respectively.

Figure 4.13 plots min/max values for each simulated time step in the worst case. The number of discrete time steps for water level computations is 800, each of 0.1 s. The number of mode symbols required to cover all possible transitions between "ON" and "OFF" states was 10. The total run time for symbolic simulation was 1.14 s.

Chapter 5

Extended Affine Arithmetic Assertions (XAA + A)

5.1 Description

Beside parameter uncertainties, affine forms can also be used to model allowed/forbidden areas of system properties. For this purpose XAA+A assertions are implemented. In XAA+A assertions the affine forms are combined with the set of temporal and frequency operators. This way system properties are fully described in the time, but also in the frequency domain. The syntax of XAA+A assertions is composed of three sets of operators:

1. *Analog operators.* This set contains arithmetic and comparison operators to compute and evaluate affine analog signals
2. *Boolean operators.* These operators perform standard Boolean operations and are used to combine more assertions into one.
3. *Temporal operators.* These operators are used to evaluate properties during simulation.

Table 5.1 lists available operators used to express properties in the form of XAA assertions. The language of XAA+A is defined through the following definitions. In these definitions AS assigns affine signals whose values are $XAAF$ data type and AA assigns the set of affine forms given by Eq. 3.1.

Definition 5.1. *The set of atomic propositions for describing properties in the time domain TBF (Time Boolean Function) is the set satisfying:*

$$c \in AS \wedge d \in AA \Rightarrow c \bullet d \in TBF$$
$$(\phi \in AS, time \in AA, \vartheta \in AA) \Rightarrow IN\left\{[time]\right\}(\phi, \vartheta) \in TBF$$
$$\beta, \gamma \in TBF \Rightarrow \beta \oslash \gamma \in TBF \wedge (!\beta) \in TBF$$

The second definition defines the smallest set of atomic propositions used

Table 5.1: XAA+A operators

XAA+A operators	symbols
arithmetic operators	$\ominus \in \{+, -, *, /\}$
relational operators	$\bullet \in \{<, >, \leq, \geq, ==, \,!{=}\}$
logic operators	$\oslash \in \{\&\&, \|, \Rightarrow, !\}$
affine analog operator	IN
affine analog freq. operators	min, max FIN, GFIN
temporal operators	$\square \in \{G, F\}$
affine analog time	$time = t_0 + \varepsilon\Delta t; t_0, \Delta t \in \mathbb{R}$
clock time intervals	$[t_1, t_2]$

to describe properties in the frequency domain, FBF (Frequency Boolean Function).

Definition 5.2. *The set of frequency formulas FBF is the set satisfying:*

$$\begin{gathered}
\alpha \in AS \Rightarrow FFT(\alpha) \in FF \\
\alpha \in AS \Rightarrow phase(FFT(\alpha)) \in FF \\
\beta \in FF, \gamma \in AA \Rightarrow FIN(f_1, f_2, \beta, \gamma) \in FBF \wedge GFIN(f_1, f_2, \beta, \gamma) \in FBF \\
\beta \in FF, \gamma \in AA \Rightarrow IN(min\,\{max\}\,\{[f_1, f_2]\}\,(\beta, \gamma)) \in FBF \\
\beta \in FF \Rightarrow, \gamma \in AA \Rightarrow min\,\{max\}\,\{[f_1, f_2]\}\,(\beta) \bullet \gamma \in FBF \\
\tau, \upsilon \in FBF \Rightarrow \tau \oslash \upsilon \in FBF \wedge !(\tau) \in FBF
\end{gathered}$$

where $f_1 \in \mathbb{R}$ is a lower bound and $f_2 \in \mathbb{R}$ is an upper bound of the frequency range within which a property in the frequency domain is verified. A value of f_1 is always less than a value of f_2, $f_1 < f_2$.

Definition 5.3. *The set of XAA+A is the set satisfying:*

$$\begin{gathered}
TBF \cup FBF \subset XAA + A \\
\alpha, \beta \in XAA + A \Rightarrow (\alpha \oslash \beta \in XAA + A) \wedge !(\alpha) \in XAA + A \\
\alpha \in XAA + A \Rightarrow \square(\alpha) \in XAA + A \wedge \square[t_1, t_2](\alpha)
\end{gathered}$$

The brief meaning of each of the XAA+A operators is given in the following.

Arithmetic operators stand for standard mathematical operations, and no further explanation is needed.

Relational operators. These operators are used to compare signal values with specified thresholds d. These operators can compare analog, and digital signals with d. If analog signals are ranges, the meaning of relational operators is as follows:

- $c < d$ - is satisfied if the upper bound of the range c is lower than the threshold d
- $c > d$ - is satisfied if the lower bound of the range c is higher than the threshold d
- $c \leq d$ - is satisfied if the upper bound of the range c is lower than or equal to the threshold d
- $c \geq d$ - is satisfied if the lower bound of the range c is higher than or equal to the threshold d

Logic operators stand for logic and $\wedge$, or $\vee$ and implies $\Rightarrow$, respectively. The unary operator ! stands for negation. The purpose of these operators is to combine multiple assertions into one.

Affine analog frequency operators. The set of affine analog frequency operators contains the following operators:

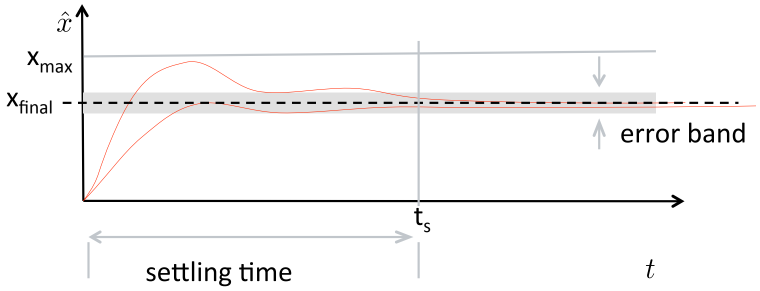
$$\begin{aligned} min\,[f_1 \;\; f_2]\,(arg) \\ max\,[f_1 \;\; f_2]\,(arg) \\ FIN(f_1, f_2, arg, spec) \\ GFIN(f_1, f_2, arg, spec) \end{aligned}$$

where the input parameter arg is $FFT\,\langle N\rangle\,(\alpha)$ or $phase(FFT\,\langle N\rangle\,(\alpha))$. $FFT\,\langle N\rangle\,(\alpha)$ computes Fast Fourier Transform (FFT) of the affine signal α. N is the number of points for which FFT is computed. If arg is $FFT(\alpha)$, the operators $min(max)$ find minimum (maximum) amplitude over the frequency interval $[f_1, f_2]$. If arg is $phase(FFT(\alpha))$ the operators find minimum (maximum) phase over the interval $[f_1, f_2]$. The specified frequencies f_1 and f_2 are real numbers where $f_1 < f_2$. If they are omitted, default values for f_1 and f_2 are $0\ Hz$ and half of the signal sampling frequency f_s, respectively.

As the first parameter the operators FIN and $GFIN$ accept either $FFT\,\langle N\rangle\,(s)$ or $phase(FFT\,\langle N\rangle\,(s))$. The FIN operator checks if for a signal s there is a frequency f, $(f_1 \leq \exists f \leq f_2)$ at which the signal amplitude/phase lies in the specification area specified by *spec*. If this frequency is found, the result of FIN operator is `true`, otherwise `false`. The $GFIN$ operator checks if for all frequencies f, $f_1 \leq \forall f \leq f_2$, of a signal s, the amplitude/phase lies in *spec*. If at least for one frequency a signal amplitude/phase is out of the specification area, the operator $GFIN$ returns `false`, otherwise `true`.

Affine analog operator: $IN\,\{[time]\}\,(arg, spec)$. The *time* parameter is the optional parameter. In the case it is specified, IN accepts only the affine signal as the argument arg and it is evaluated within the time interval *time*.

If the *time* parameter is not specified, the IN operator additionally accepts *min* and *max* frequency operators as the first argument. The operator IN returns **true** Boolean value at the time point, at which the first argument



$$overshoot = \frac{x_{max} - x_{final}}{x_{final}} [\%]$$

Figure 5.1: Step response of a control system

lies in the specification area *spec*.

Temporal operators: G and F. These operators are used to verify system properties during simulation. The operator G and F assign that a property must hold always or eventually during simulation, respectively. If the clock time interval is specified, the property is checked every t_1 clock cycle in the time interval $[t_1, t_2]$.

5.2 Specification of properties with XAA+As

The following part of the chapter lists some typical system properties and shows how they can be specified using XAA+As.

1. Step response This property is crucial in the verification of system stability and gives the answer to the question: "Is a system able to produce a bounded output for a bounded input?" Step response is defined as a system response to an ideal step applied to its input. The parameters of the step response are usually overshoot, settling time, and error band around the final value. A step response with its typical parameters is shown in Figure 5.1. *Overshoot* occurs when a system output exceeds the final value. It is usually specified in percentages and is calculated as:

$$overshoot = \frac{x_{max} - x_{final}}{x_{final}}$$

where x_{max} and x_{final} are shown in Figure 5.1. *Settling time* is defined as the maximum time necessary for the output signal to settle within the error band. This time is calculated from the time at which an ideal input step is applied. Using the operators of XAA+A language the parameters of the step response can be checked with the following assertion:

$$G(\hat{x}(t) \leq x_{max}) \ \ \&\&$$
$$F(IN[\tfrac{t_s}{2} + \varepsilon_1 \tfrac{t_s}{2}](\hat{x}(t), x_{final} + \varepsilon_2 tol) \Rightarrow G(IN(\hat{x}(t), x_{final} + \varepsilon_2 tol))))$$

where:

- $\hat{x}(t)$ is an affine signal shown in Figure 5.1.
- x_{max} is the maximum $\hat{x}$ value allowed by the overshoot.
- $G(\hat{x}(t) \leq x_{max})$ checks if the value of $\hat{x}(t)$ is always below the maximum value.
- $\frac{t_s}{2} + \varepsilon_1 \frac{t_s}{2}$ specifies the time interval $0 \leq t \leq t_s$ within which a system output should enter the error band around the final state value.
- $x_{final} + \varepsilon_2 tol$ is the error band which defines the allowed ringing of the signal $\hat{x}$ around the final value x_{final}.
- IN operator checks if the affine signal $\hat{x}(t)$ within the time interval $t \in [0, t_s] = \frac{t_s}{2} + \varepsilon_1 \frac{t_s}{2}$ settles to a value lying in the interval $x_{final} + \varepsilon_2 tol$.
- the implication operator $\Rightarrow$ assigns that as once the signal $\hat{x}(t)$ enters the error band it should stay their till the end of simulation (assigned by G operator).
- F operator checks if the property in the bracket will hold eventually during simulation.
- && is `and` operator used to combine overshoot and settling time properties into one property. It is satisfied only if both properties are *true*.

2. Operational range. This property defines the output range that does not cause a circuit (e.g., amplifier, integrator, etc.) to saturate. To describe this property with XAA+A, the allowed operational range is specified as an affine term:

$$spec(swing) = SW + \varepsilon\beta \qquad \varepsilon \in [-1, 1]$$

where SW represents the center of the range and β the maximum absolute distance from the center value. A circuit will not saturate if an output voltage stays within this area. XAA assertion, which describes this property, is as follows:

$$G(IN(vol\_out, spec(swing))).$$

The temporal operator G checks if the property $(IN(vol\_out, spec(swing)))$ holds always during simulation run.

Beside time behavior, XAA+A assertions are also able to describe system properties in the frequency domain. Typical examples for these kinds of properties are:

1. Stability in a control theory expressed in the form of phase and amplitude margin, stability margin, or sensitivity function
2. Filter specifications such as allowed ripple in the pass/stop band

Allowed ripples in pass/stop band. Without loss of generality, a low pass filter is used. The allowed ripple in the pass band defines the maximum ringing of the filter around the DC gain $H(\omega = 0)$. The allowed ripple in the

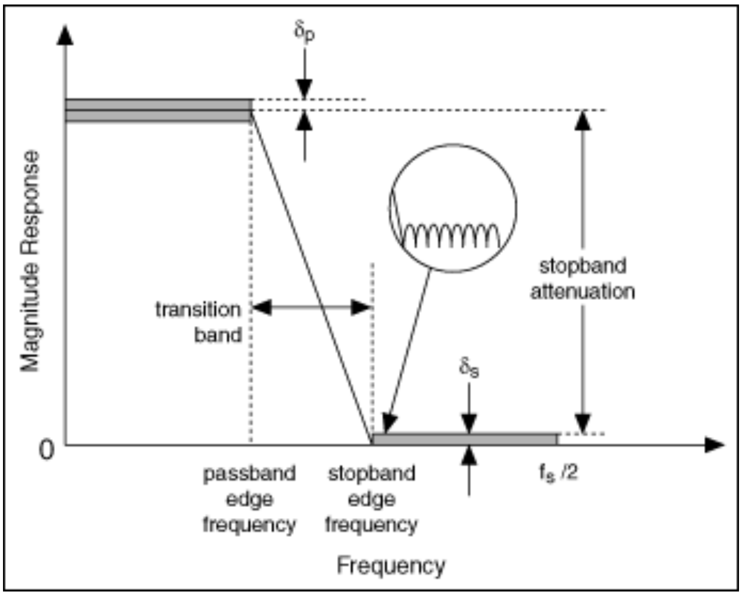


Figure 5.2: Magnitude response of an analog low pass filter

stop band specifies the minimum attenuation of the filter above the stopband edge frequency. These ripples are modeled using Affine Arithmetic as:

$$H(f) \in \begin{cases} H(0) + \varepsilon_1 \delta_p & 0 \leq f \leq f_p \\ \frac{\delta_s}{2} + \varepsilon_2 \frac{\delta_s}{2} & f \geq f_n \end{cases}$$

where δ_p and δ_n are the maximum allowed variation from H(0) and the minimum attenuation, respectively. The frequencies f_p and f_s assign passband and stopband edge frequencies, respectively. These requirements are shown in Figure 5.2. They can be checked with the following XAA+As:

$$\begin{aligned} & GFIN(0, f_p, FFT\langle N\rangle(h(t)), H(0) + \varepsilon_1 \delta_p) \\ \&\& \; & GFIN(f_n, FFT\langle N\rangle(h(t)), \frac{\delta_s}{2} + \varepsilon_2 \frac{\delta_s}{2}). \end{aligned}$$

The filter transfer function $H(f)$ is computed using FFT operator, which computes Fast Fourier Transform of the filter impulse response $h(t)$. The operator GFIN checks if the amplitude values of $H(f)$ meet desired requirements in the pass/stop band. The operator returns true value if the amplitudes of $H(f)$ for $\forall f \in [0, f_p]$ or $\forall f \geq f_n$ lie in the allowed specification areas. Since for the stop band the upper bound of the frequency is not specified, GFIN assumes the default value equal to half of the signal sampling frequency $\frac{f_s}{2}$.

5.3 Illustration example

The use of assertions is demonstrated on a closed loop control system with a PID (Proportional-Integral-Derivative) controller. Its block diagram is shown by Figure 5.3. The process P in a system is described with the following function:

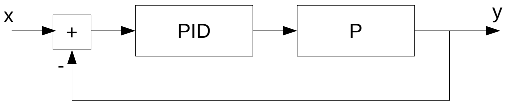


Figure 5.3: Block diagram of a system with PID controller

$$P(s) = \frac{1}{s^2 + as + 1}$$

where parameter a is 0.6. It is further assumed that the parameter a due to some uncertainties (here component tolerances) deviates from its nominal value. The exact value of a is not known but it lies in the interval $[0.4, 0.8]$. Using Affine Arithmetic the uncertainty of a can be modeled as $a = 0.6 + \varepsilon 0.2$.

The considered PID controller includes the noise filter for the derivative term and its transfer function is shown below:

$$C(s) = K_p(1 + \frac{1}{T_i s} + \frac{T_d s}{\frac{T_d}{20} s + 20}),$$

where the proportional gain K_p is 1.8, the integral time $T_i = 0.38s$ and the derivative time $T_d = 0.095s$. The ratio between the integral and derivative times is 4 ($T_i = 4 * T_d$). In [103] it is shown that this ratio is appropriate for many industrial processes.

A proper behavior of the control system must satisfy the certain number of requirements. The most important one is the stability of the closed loop. The stability criteria can be expressed in a vast number of terms: gain and phase margin, sensitivity function, stability margin, etc. For a simple illustration the stability margin is chosen.

This parameter is defined as the shortest distance between the Nyquist curve of the loop transfer function and the critical point -1. This distance is actually the inverse of the maximum value of the sensitivity function. The loop transfer function is determined with $L(s) = C(s)P(s)$, where $C(s)$ and $P(s)$ are the controller transfer function and the process transfer function, respectively. Mathematically, the stability margin can be expressed as:

$$\begin{aligned} M_s &= \inf_{\omega} |-1 - L(j\omega)| = \inf_{\omega} |1 + L(j\omega)| \\ &= \left[\sup_{\omega} \left|\frac{1}{1 + L(j\omega)}\right|\right]^{-1} = \left[\sup_{\omega} |S(j\omega)|\right]^{-1} \end{aligned}$$

where $S(j\omega) = \frac{1}{1+L(j\omega)}$ is the sensitivity function. In particular, the sensitivity function represents the disturbances amplification at the system output by the closed loop system. Recommended values for the stability margin M_s lie in the range of $[0.5, 0.75]$ [103].

The range of M_s specification can be represented as affine form:

$$spec(M_s) = M_s^{'} + \varepsilon\delta \qquad \varepsilon \in [-1, 1]$$

where $M_s^{'}$ represents the center value of the specified range and δ the maximum absolute distance from the center value. The range $[0.5, 0.75]$ of M_s is modeled with an affine term:

$$spec(M_s) = 0.625 + \varepsilon 0.125 \;\; \varepsilon \in [-1, 1]$$

Taking the uncertainty of the parameter a into account, the stability margin M_s can be rewritten as:

$$\begin{aligned} M_s &= \inf_{\omega}(|1 + C(j\omega)P(j\omega)|) \\ &= \inf_{\omega}\left(\left|1 + C(j\omega)\frac{P_{nom}(j\omega)}{1 + \varepsilon P_{dev}(j\omega)P_{nom}(j\omega)}\right|\right) \end{aligned}$$

As described in Section 3.5.1 $P_{nom}(s)$ is the process transfer function with the nominal value of a:

$$P_{nom}(s) = \frac{1}{s^2 + a_{nom}s + 1}$$

and P_{dev} is the process deviation due to deviation of the parameter a:

$$P_{dev}(s) = a_{dev}s.$$

A control system including uncertainties meets the stability margin specification if its stability margin M_s lies in the range $[0.5, 0.75]$; $(1+C(j\omega)P(j\omega))$ is computed as the transfer function of the system given in Figure 5.4. The transfer function is found as FFT of the the system response to Dirac-Impulse. For this purpose the following XAA+A operators are used:

$$min(FFT\left<N\right>(h(t)))$$

where $h(t)$ is the impulse response of the system shown in Figure 5.4 and $FFT\left<N\right>$ computes the FFT at N points. The operator min finds the frequency component with the minimum amplitude value.

Comparing M_s with the specified bounds $[0.5, 0.75]$ is done using the operator IN. Thus, XAA+A assertion describing the stability criteria is as follows:

$$IN(min(FFT\left<N\right>(h(t))), 0.625 + \varepsilon 0.125).$$

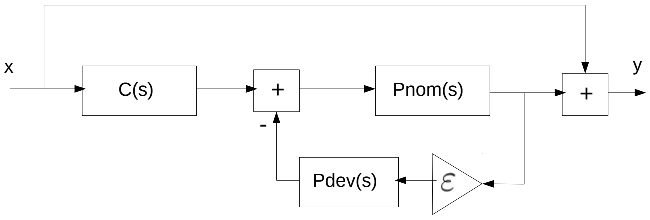


Figure 5.4: Calculation of stability margin M_s

The system was simulated in SystemC AMS. The assertion passed and the stability margin of the system from Figure 5.4 is shown in Figure 5.5. The FFT of the system response was calculated at $N = 2048$ points. The total run time of system simulation without XAA+A assertion took 0.036 s. Adding the assertion into simulation process the total time increased with a small overhead of 0.004 s. The new time was then 0.04 s. The application

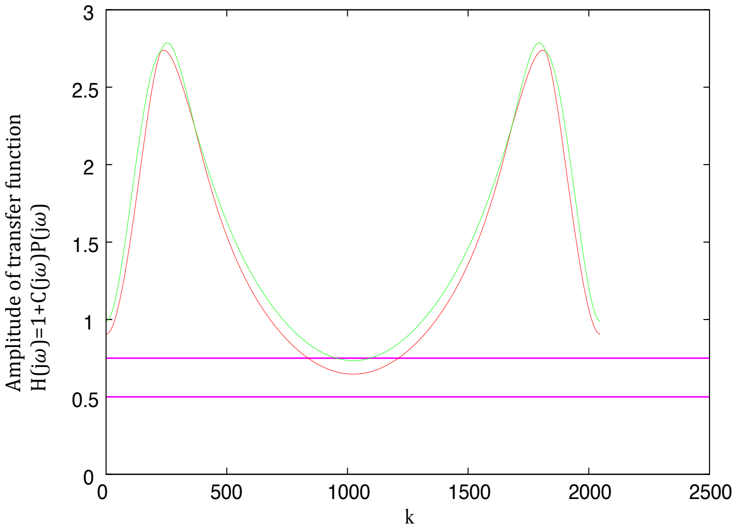


Figure 5.5: Possible value of M_s within the specification range $[0.5, 0.75]$

of this approach on complex case studies is shown in Chapter 6.

5.4 Implementation

The assertion library is developed as a separate library in SystemC AMS. All assertion operators are realized as separate modules. The modules for evaluation of digital signals use SystemC Modules of Computation (MOCs).

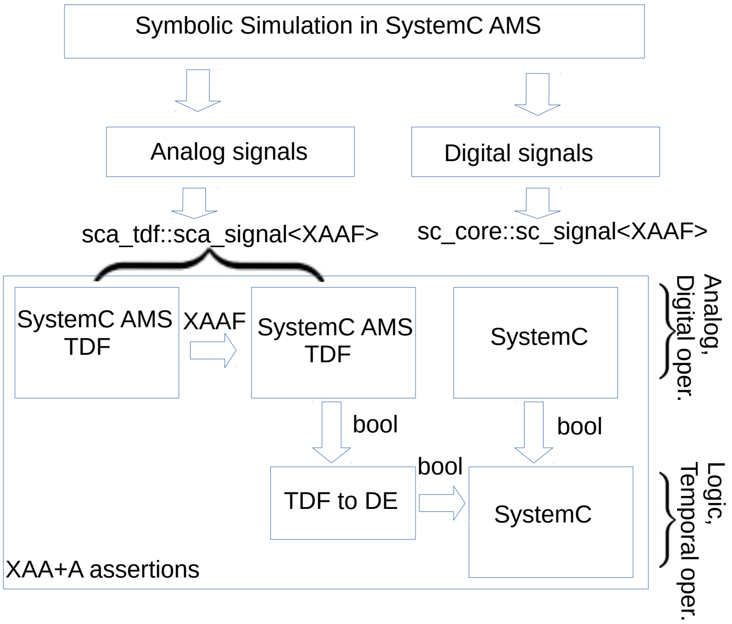


Figure 5.6: The implementation structure of XAA+A

Evaluation of analog affine signals is done via SystemC AMS TDF (Timed Data Flow) MOC. In the following the terms analog and digital operators will be used to assign the operators used for evaluation of analog and digital signals, respectively.

Figure 5.6 shows the implementation structure of XAA+A assertions. As described in Section 5.1 the relational operators can evaluate analog and digital signals. If analog signals are compared with specifications, the operators are implemented via TDF MOCs. For evaluation of digital signals SystemC MOC is used. To construct an XAA+A describing a full system property, analog and digital operators are usually connected with one another. The connection between operators is done through the existing signal interfaces in SystemC AMS:

- Analog operators are connected via $sca\_tdf :: sca\_signal < T >$.
- Digital operators are connected via $sc\_core :: sc\_signal < T >$.
- The connection between analog and digital operators is done via logic operators also implemented using SystemC MOC. Analog operators are connected to the logic operators via TDF to DE (Discrete Event) interfaces $sca\_tdf :: sc\_signal < T >$.

T stands for a data type of signal quantities. All signal quantities in a system (either analog or digital) are the same data type $XAAF$. The return values of assertion operators can be both types: $XAAF$ or Boolean `bool`. The `true`

Boolean value is for a property described with a corresponding operator that is satisfied at evaluation time. Similarly, `false` value indicates that the property is violated. The assertion operators that return `XAAF` data type are:

- arithmetic operators $\{+, -, *, /\}$
- min frequency operator
- max frequency operator
- FFT frequency operator
- $phase(FFT)$ frequency operator

The Boolean value `bool` is returned by:

- relational operators $\{<, \leq, >, \geq\}$
- analog IN operator
- analog frequency operator FIN
- analog frequency operator $GFIN$
- logic operators
- temporal operators

The evaluation of affine analog signals is done at each simulation time step defined by sampling frequency of the overall system. For this purpose, TDF *processing* function is used. The digital signals are evaluated at discrete steps defined by changes of signal values. For this purpose, a process method SC_METHOD of SystemC MOC is used.

Chapter 6

Evaluation

The applicability and efficiency of XAAF approach is demonstrated on two widely used Mixed-Signal circuits: a 3rd Order Δ-Σ Modulator and a Phase-locked loop circuit. Simulation is performed on a system level. For this purpose SystemC AMS simulation environment is used. The models are simulated over a range of operating conditions. The system responses including all possible trajectories over this range are obtained using one simulation run. The obtained simulation results are compared with multi-run methods: Monte-Carlo simulation and Design of Experiments (DoE). It is shown that the upper and lower bounds of trajectories computed by symbolic simulation efficiently enclose behaviors computed by numeric simulations.

6.1 3rd order Delta-Sigma Modulator

Delta-Sigma (Δ-Σ) modulators are common parts of today's analog-to-digital converters (ADCs). The aspects that bring these ADCs a higher efficiency over conventional ones are the use of oversampling, noise shaping and digital filter decimation. Conventional ADC as flash converters uses sampling frequencies close to twice the sampling frequency. Sampling in the time domain corresponds to shifting the input signal spectrum to higher frequencies in the frequency domain (modulation with carriers at frequencies $2f_s$, $3f_s$, $4f_s$, etc.). A sampling frequency close to the Nyquist rate requires a filter with tight constraints to isolate the elementary input signal spectrum from modulated bands. In traditional ADCs this filter is realized on their analog side and hence it is more complex to design and implement. However, Δ-Σ ADCs use a sampling rate much higher than the Nyquist one. Therefore, the requirements for antialiasing filter are much milder.

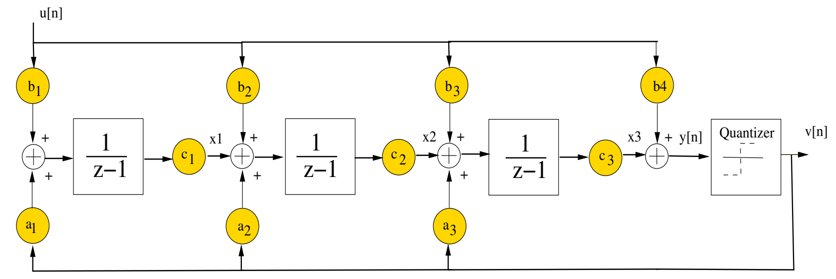


Figure 6.1: Block diagram of a 3rd Order Delta-Sigma Modulator [61]

6.1.1 Modulator description

Δ-Σ Modulator as a part of Δ-Σ ADC is designed such that it acts as a low pass filter for analog input signals and a high pass filter for the quantization noise. The quantization noise is added by 1 bit quantizer of Δ-Σ modulator which converts analog signal to the digital one (the bit stream). The signal from the modulator output is sent to the digital decimation filter, which down samples the signal reducing its sampling rate. These filters are due to their digital nature simpler to design than analog ones. Reducing the sampling rate, they average the signal brought to their input removing all high frequency components (quantization noise, aliases) and passing only the useful signal band.

Figure 6.1 shows the block diagram of the 3rd order Δ-Σ modulator inspired from [54]. The modulator property of crucial importance is the stability. Stability requires that the output stays bounded if the input is bounded. The increase of output values can lead to two unwanted modulator behaviors: *integrator saturation* and *quantizer overload.*

Rounding the input value to the certain quantization level, the quantizer adds the certain portion of quantization error. The maximum value of the quantization error depends on the number of quantization bits used to represent the output value. If the error between quantizer input and output value exceeds the maximum value, the quantizer is overloaded. It is of a great importance to provide a formal proof that over the certain range of input and initial conditions these effects never occur. Using the proposed method, the modulator stability is verified over several sets of input and initial values. The worst-case behavior is obtained in one symbolic simulation run.

6.1.2 Results of Symbolic Simulation

The modulator parameters used in simulation are taken from the work [54]: $b_1 = 0.0444$, $b_2 = 0.2881$, $b_3 = 0.7997$ $a_1 = -0.0444$, $a_2 = -0.2881$,

$a_3 = -0.7997$ $b_4 = 1.0$, $c_1 = c_2 = c_3 = 1.0$ The quantizer used in this modulator uses one bit to represent the output value. The quantization levels have values -1 and 1. Symbolic simulation is used to compute the worst-case modulator behavior over the considered ranges of initial and input conditions. For the considered modulator two undesired effects are checked: integrator saturation and a quantizer overload. For this modulator, saturation limits are assumed to be -2 and 2. Thus, the integrator outputs must stay within the range $[-2, 2]$ to avoid saturation.

On the other side, a quantizer is overloaded if a difference between the quantizer input and output exceeds the maximum quantization error. For the considered quantizer, the maximum value of the quantization error is 1. Hence, to avoid overload, the quantizer input should stay within the bounds $[-2, 2]$. In this work the modulator behavior is simulated over three sets of input and initial conditions.

The considered one-bit quantizer compares its input value $y[n]$ with zero. For the positive input values, the quantizer output is 1, while for negative ones it is -1. Figure 6.2 shows the quantizer implementation in SystemC AMS.

Since input and initial values are modeled as ranges, the quantizer input will also be a range. Hence, one set may lie above and the other below zero. Hence, the `if-else` statement is modified according to the rules presented in Section 4.4. The first branch computes both `if` and `else` branches. The modulator BIBO (Bounded Input Bounded Output) stability was checked over the sets of initial and input values. Computation times of the proposed symbolic approach for three sets of input and initial conditions are given in Table 6.1. All computations are performed on a 2.6 GHz machine.

```
SCA_TDF_MODULE(quantizer)
{
  sca_tdf::sca_in<XAAF> y;
  sca_tdf::sca_out<XAAF> v;

  void processing()
  {
    if ((y≥0)!=false)
      v.write((y≥0)*1+!(y≥0)*(-1));
    else
      v.write(-1);
  }

  quantizer(sc_module_name nm){} // constructor
};
```

Figure 6.2: Implementation of an one-bit quantizer in SystemC AMS

Table 6.1: Computation time of symbolic approach

Initial conditions	Number of simulation time steps N	Computation time
$x_1(0) \in [0.012, 0.013]$ $x_2(0) \in [0.01, 0.02]$ $x_3(0) \in [0.8, 0.82]$, u=0.54	$N = 100$	$2.2s$
$x_1(0) \in [0, 0.01]$ $x_2(0) \in [-0.01, 0]$ $x_3(0) \in [0.8, 0.82]$, u=[-0.6, 0.6]	$N = 40$	$84.4s$
$x_1(0) \in [-0.1, 0.1]$ $x_2(0) \in [-0.1, 0.1]$ $x_3(0) \in [-0.1, 0.1]$, u=[-0.5, 0.5]	$N = 30$	$804s$

6.1.3 Comparison with Monte-Carlo simulation

The results obtained by symbolic simulation are compared with numeric simulations based on the Monte-Carlo method and DoE. Using the Monte-Carlo approach multiple test cases are run for randomly chosen values of initial and input conditions. Each test case simulates one particular choice of condition values. For comparison, 10 random simulation runs are repeated.

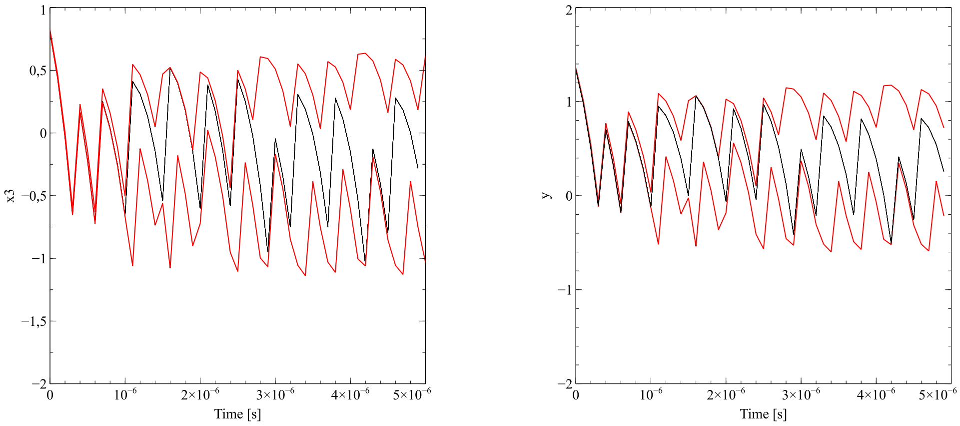
The first set of conditions is inspired from [104] that computes the worst-case behavior for $N = 38$ time steps. [104] performs reachability based on Interval Arithmetic (IA). However, the dependency problem of IA can lead fast to divergence of response bounds. This limits the number of time steps to be simulated.

Using the proposed methodology, the modulator behavior was symbolically simulated for $N = 100$ time steps. The results show 16 times higher computation speed over [104], even for higher number of simulation time steps. Ten ω symbols were required to cover all possible switches of the 1-bit quantizer between two output values. Note that the number of ω symbols was much lower than the number of simulation time steps. This is due to correlation of transitions which were identified by implemented relational operators. Here the $\geq$ operator was used for the implementation of the 1bit quantizer (see Figure 6.2).

Figure 6.3 shows the result of symbolic simulation compared with 10 random simulations. Left and right side of the figure show the output of the third integrator and the input of 1-bit quantizer, respectively. The red line assigns the total bounds of modulator quantities computed by symbolic approach. The black lines show the result of 10 numeric simulations.

A Monte-Carlo run took only 2 ms. However, the proposed methodology is still competitive considering the fact that this time is multiplied with thousands of runs to obtain a sufficient coverage. Even then, there is no guarantee that the worst-case behavior is found. The bounds computed by symbolic approach encloses the traces of pure numeric random simulations.

Since the bounds of the corresponding signals stay within the range $[-2, 2]$, no integrator saturation and quantizer overload occurr. This is au-



(a) The output of the third integrator **(b)** The quantizer input

Figure 6.3: Comparison of symbolic simulation with random simulation for the 1st set of initial conditions

tomatically checked using the assertion language described in Chapter 5. The assertions, which check if the bounds of corresponding signals lie in the allowed range, are as follows:

$$G(IN(x_3, 2 * \varepsilon_1)) \tag{6.1}$$

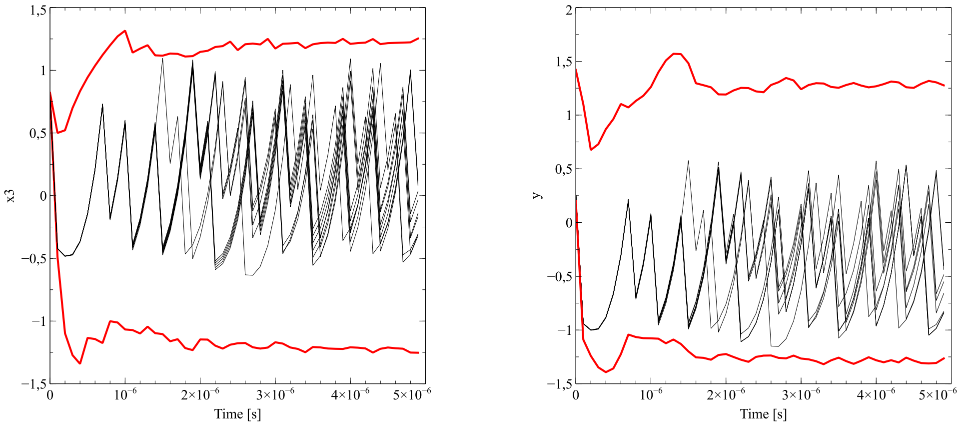
$$G(IN(y, 2 * \varepsilon_1)) \tag{6.2}$$

where $\varepsilon_1 \in [-1, 1]$. Both assertions were satisfied and the simulation passed. Table 6.1 reports the computation times not including assertions in the simulation process. Adding Assertions 6.1 and 6.2 in the simulation run the computation time increased to 2.87 s. Therefore, the assertions added a small overhead of 0.67 s.

Due to the nonlinear behavior of the quantizer, worst-case parameter values can lie anywhere inside the initial ranges and can be different at each time step. These values are usually not trivial to be found by random simulations. However, the LP solver integrated in the symbolic computations provides these values, as shown below in Table 6.2.

The simulation of the second set shows the benefits of this approach over the methodology presented in [54]. In contrast to [54], the modulator is simulated for the set of input values and not only one constant value. After N=20 computation time steps the modulator behavior is quite stable resulting in slight changes of the worst-case behavior.

Figure 6.4 shows the modulator behavior for additional 20 time steps. The left side of the figure shows the third integrator output while the right side the quantizer input. As in the previous case both assertions 6.1 and 6.2 passed. The computation took 84.4 s using 15 ω symbols to cover all



(a) The output of the third integrator

(b) The quantizer input

Figure 6.4: Comparison of symbolic simulation with random simulation for the 2nd set of initial conditions

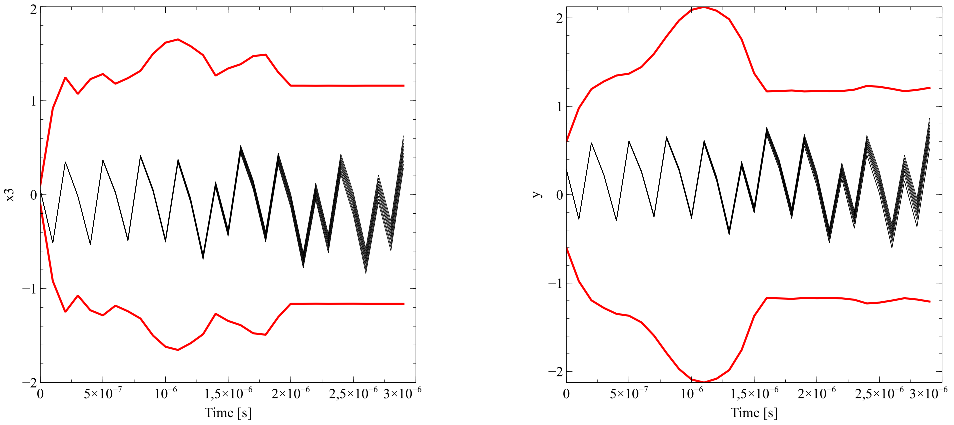
possible quantizer transitions. Including assertions in the simulation process the simulation time increased only in 0.6 s; the new computation time was 85 s.

Figure 6.5 shows the modulator behavior for the third set of conditions. The modulator shows an interesting property that the absolute values of worst-case bounds are the same over the symmetric ranges of input and initial conditions. The worst-case modulator response shows stable behavior after $N = 20$ time steps. The simulation was performed for additional 10 time steps. The computation time for the time horizon of $N = 30$ time steps took 804 s (around 14 min) on a 2.6 GHz machine. The number of used ω symbols to cover all possible transitions was 18. For the same set of conditions and the same time horizon, [53] reported more than 2 hours computation time on a similar machine.

As in the previous cases the first assertion was satisfied. However, the quantizer input exceeded 2 and the overload occurred. Hence, the second assertion failed and the simulation run was stopped at 1 µs.

The system output, computed by the symbolic simulation run, encloses traces obtained from numeric simulations. Symbolic simulation computes the total lower and upper bounds of system response whose values could not have been found easily by simulation with the bounds of initial and input ranges. This is due to the nonlinear discrete behavior of the 1-bit quantizer whose output value does not stay constant, but switches between two values during simulation.

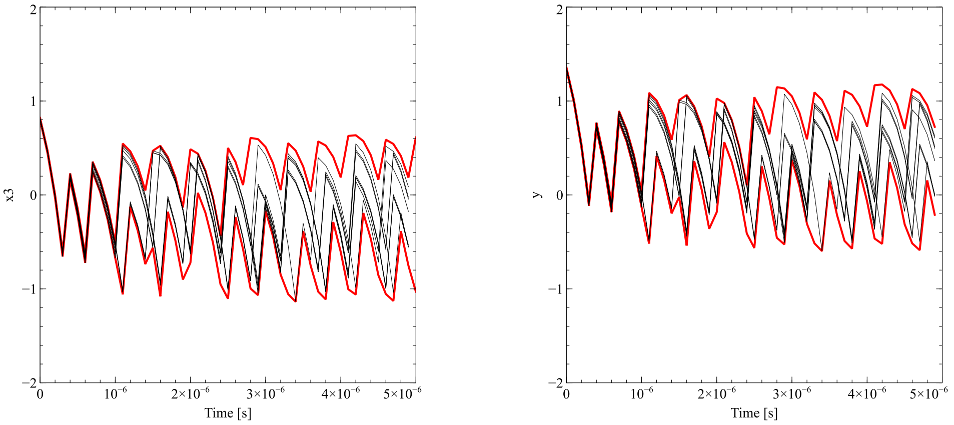
Due to the nonlinear behavior of the quantizer, the worst-case parameter values at each time step may be different and can lie anywhere inside the



(a) The output of the third integrator **(b)** The quantizer input

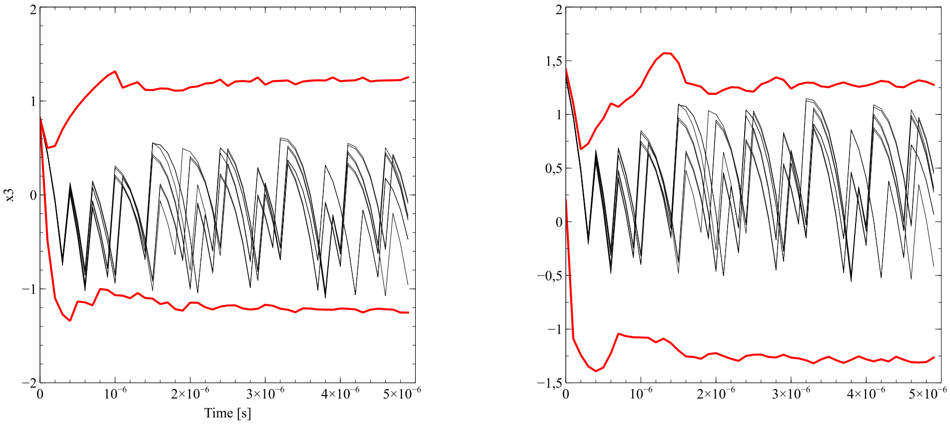
Figure 6.5: Comparison of symbolic simulation with random simulation for the 3rd set of initial conditions

range. Table 6.2 shows the input sequences for computation of the worst-case bounds of the third integrator $x_3[n]$. The value n in the table assigns the nth simulation time step. In contract to the previous two cases, Figure 6.5 on the right side shows the quantizer overload. For the certain input sequence the quantizer input y exceeds the value 2. The positive quantizer



(a) The output of the third integrator **(b)** The quantizer input

Figure 6.6: Comparison of symbolic simulation with DoE for the 1st set of initial conditions



(a) The output of the third integrator

(b) The quantizer input

Figure 6.7: Comparison of symbolic simulation with DoE for the 2nd set of initial conditions

overload occurs at $n = 10$ and $n = 11$ for the following input sequence:

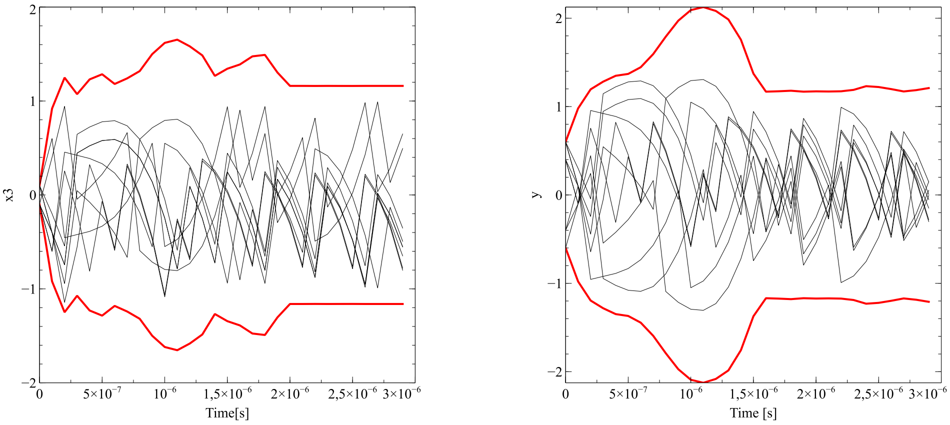
$$\begin{array}{rcl} u & = & 0.472608 \\ x_1[0] & = & 0.1 \\ x_2[0] & = & -0.1 \\ x_3[0] & = & 0.0491474 \end{array}$$

and $n = 12$ for:

$$\begin{array}{rcl} u & = & 0.5 \\ x_1[0] & = & 0.1 \\ x_2[0] & = & -0.09808 \\ x_3[0] & = & -0.00207 \end{array}$$

Similar holds for the lower bound (negative quantizer overload). The absolute values of sequences are the same, only the signs are changed. For $n = 10$ and $n = 11$ the sequence is:

$$\begin{array}{rcl} u & = & -0.472608 \\ x_1[0] & = & -0.1 \\ x_2[0] & = & 0.1 \\ x_3[0] & = & -0.0491474 \end{array}$$



(a) The output of the third integrator **(b)** The quantizer input

Figure 6.8: Comparison of symbolic simulation with DoE for the 3rd set of initial conditions

and $n = 12$ for:

$$\begin{aligned} u &= -0.5 \\ x_1[0] &= -0.1 \\ x_2[0] &= 0.09808 \\ x_3[0] &= 0.00207 \end{aligned}$$

These values are not trivial to be detected by random simulations. Most likely they would not be detected even with the higher number of simulation runs. Therefore, random simulation even with a high number of simulation runs could not guarantee that all critical cases are covered.

6.1.4 Comparison with Design of Experiments

The results of symbolic simulation were also compared with DoE[15]. Here, DoE used a simple metamodel which looks only at the corners of initial and input ranges. Figures 6.6-6.8 show the comparison results with symbolic simulation. One DoE run took around 2 ms. For simulation of all bounds, DoE multiplies this time with required number of runs that increases exponentially with the number of considered uncertainties. However, simulation of all corners does not guarantee that the worst-case behavior is detected. The worst-case parameter values do not necessary lie on the corners, as shown in Table 6.2.

Table 6.2: Input and initial values in the worst-case at nth simulation time step

set	$x_3[n=4]$		$x_3[n=10]$		$x_3[n=20]$	
	min	max	min	max	min	max
1st	u=0.54 x1[0]=0.012 x2[0]=0.01 x3[0]=0.8	u=0.54 x1[0]=0.013 x2[0]=0.02 x3[0]=0.82	u=0.54 x1[0]=0.012 x2[0]=0.01 x3[0]=0.8	u=0.54 x1[0]=0.013 x2[0]=0.02 x3[0]=0.82	u=0.54 x1[0]=0.012226 x2[0]=0.02 x3[0]=0.82	u=0.54 x1[0]=0.013 x2[0]=0.0128517 x3[0]=0.8
2nd	u=-0.6 x1[0]=0 x2[0]=-0.01 x3[0]=0.8	u=0.58215 x1[0]=0.01 x2[0]=-0.01 x3[0]=0.8	u=-0.52107 x1[0]=0 x2[0]=0 x3[0]=0.82	u=0.59655 x1[0]=0 x2[0]=-0.01 x3[0]=0.8087	u=-0.553976 x1[0]=0.00246 x3[0]=-0.01 x4[0]=0.8	u=0.565434 x1[0]=0.0160411 x2[0]=0.01 x3[0]=0.82
3rd	u=-0.286 x1[0]=-0.1 x2[0]=0.1 x3[0]=0.1	u=0.286 x1[0]=0.1 x2[0]=-0.1 x3[0]=-0.1	u=-0.1 x1[0]=-0.03034 x2[0]=0.0077047 x3[0]=-0.1	u=0.1 x1[0]=0.03034 x2[0]=-0.0077047 x3[0]=0.1	u=-0.09573 x1[0]=0.1 x2[0]=0.0128844 x3[0]=-0.1	u=0.09573 x1[0]=-0.1 x2[0]=-0.0128844 x3[0]=0.1

6.2 Charge-pump Phased-locked loop Circuit

A dual-path charge-pump Phased-locked Loop circuit (PLL) is the second case study on which the proposed methodology is applied. This circuit plays a role of the frequency synthesizer that generates the high frequency signals for the local oscillator of one IEEE 802.15.4 RF transceiver. The output frequency is N multiple of the reference frequency brought to the input of the circuit. The key property of a PLL circuit is its ability to lock to desired frequency in less than maximum specified settling time. This property is known as a PLL locking property. In this work a symbolic simulation is used to verify the locking property over a set of initial and parameter values.

6.2.1 PLL description

The structure of the circuit is shown in Figure 6.9. The SystemC virtual prototype of the circuit is provided by RWTH Aachen University (Institute for Integrated Analog Circuits) as the main case study in ongoing ANCONA (Analog-Coverage in der Nanoelektronik) project. The project is partly supported by the Federal Ministry of Education and Research (BMBF) and partly by the industry. It counts six University partners including my Institute for Design of Cyber Physical Systems, TU Kaiserslautern.

The SystemC model is generated automatically from the cadence design schematic and made available for the project partners. The basic parts of the circuit shown in Figure 6.9 are as follows:

- a voltage-controlled oscillator (VCO; top right) - generates a signal whose frequency should be N times the reference frequency
- a pure digital frequency divider (bottom) - divides the VCO frequency by N
- a phase-frequency detector (PFD; top left) - detects phase/frequency difference between a reference signal and a VCO signal (divided by N)
- a charge-pump (CP; connected to PFD) - charges/discharges capacitors in the low pass filter changing the filter output voltage. This voltage brought to one of the inputs of VCO, changes the output frequency bringing it eventually to the desired level.

Voltage Controlled Oscillator-VCO. The considered oscillator generates the output frequency according to the following frequency-voltage relation:

$$f_{vco} = f_0 - Kvco * v_1 - Kmod * v_2 - Ktune * v_3 \qquad (6.\text{-}13)$$

Its integration gives the phase of VCO output signal:

$$\Phi_v = 2\pi \int_0^t f_{vco} dt.$$

The symbols $Kvco$ and $Kmod$, $Ktune$ and f_0 stand for the oscillator coefficients and the oscillator start frequency, respectively. The voltages v_1, v_2

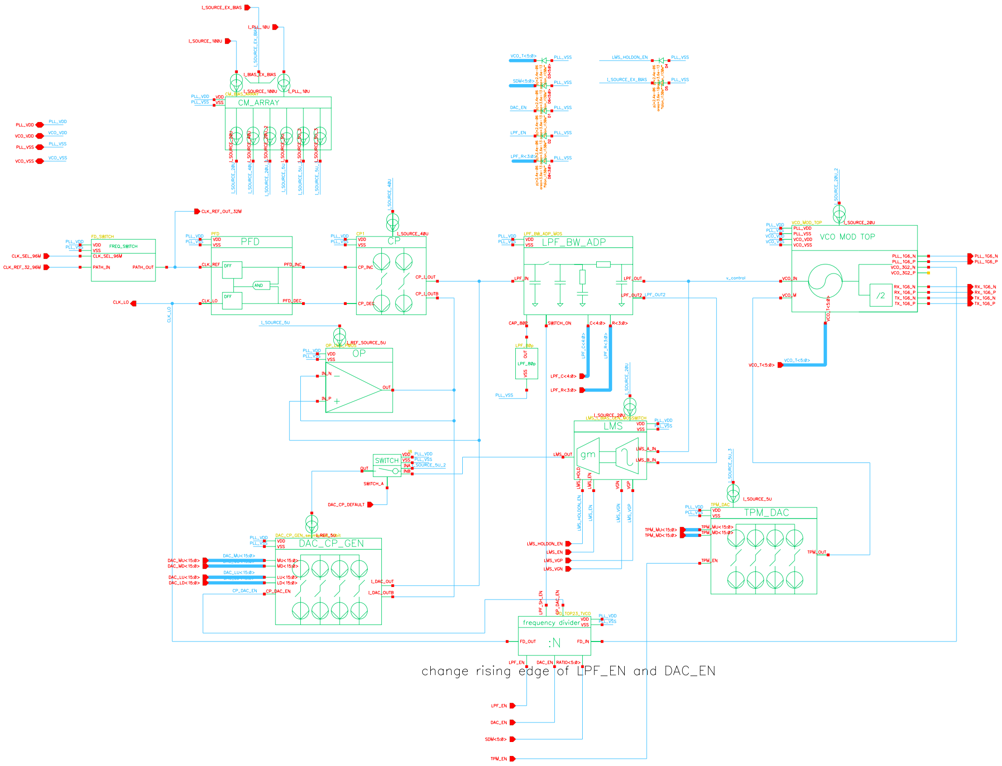


Figure 6.9: A dual-path charge-pump PLL circuit

and v_3 are the oscillator input voltages. The oscillator generates a rectangle signal with the frequency equal to f_{vco}. Since the input voltages of VCO change with time, the value of the output frequency will also be dependent on time. Hence, the period of the quadratic signal cannot be predicted in advance. This issue can be solved looking at the phase of the VCO signal. The positive edge of the quadratic signal can be detected comparing the phase Φ_v with 2π. If the phase is higher or equal to 2π the quadratic signal is 1, otherwise its value is 0.

In the considered frequency synthesizer, the quadratic signal is firstly inverted before it is brought to the input of the frequency divider (signal VCO_3G2_N in Figure 6.9). The positive edge of the inverted signal happens when the signal phase crosses the value π. If the phase is higher than π, the signal value is 1, otherwise 0.

Figure 6.10 shows the implementation of the PLL VCO in SystemC for a pure numeric simulation. Crossing the value π can be detected with the condition $\Phi_v - ths \geq 0$ where $ths = \pi$. Here it is important to mention that every time the phase value crosses π the value 2π must be added to the threshold voltage. This is done to assign the next clock cycle; resp. for each next cycle the phase value is compared with 3π, 5π, etc. Since the frequency divider is triggered only on positive edges of the VCO signal (changes from '0' to '1'), it is enough to write '1' to the output only at the moment when the phase value crosses the threshold.

However, the VCO code needs slight modifications to handle the range of phase Φ_v values. In this case, the comparison with the certain threshold can also result in both $\{\texttt{false}, \texttt{true}\}$. Thus, applying Rule 1 from Section 4.4 it is checked if the condition $(\Phi_v - ths) \geq 0$ is not equal to `false`. Figure 6.11 shows the instrumentation of the VCO code applying the rules given in Section 4.4. The values of `VCO_3G2_N` should be '1' only for `true` condition value, otherwise '0'. Also, the threshold `ths` for the next cycle should be updated only the condition results in `true`. This can be captured using Shannon Expansion as defined by Rule 2 in Section 4.4. This expression in general covers both cases; when the condition is $\{\texttt{false}, \texttt{true}\}$ but also `true`. For the `true` condition values, the conditional statement is the same as one for pure numeric simulation (given by Figure 6.10).

```
SC_MODULE(VCO)
{
   .......

   double ths=M_PI;

   void processing()
   {

     double fvco=f0-Kvco*v_1-Kmod*v_2-Ktune*v_3;

     double curr_time=sc_timestamp().to_seconds();

     double Phi_v=Phi_v+2*M_PI*fvco*(curr_time-prev_time);

     if ((Phi_v-ths)>=0)
     {
       VCO_3G2_N=1;
       ths=ths+2*M_PI;
     }
     else
       VCO_3G2_N=0;

     prev_time=curr_time;

   }
};
```

Figure 6.10: VCO implementation with double signal data type

Applying Rule 2, Shannon Expansion for computation of VCO_3G2_N

```
SC_MODULE(VCO)
{
   XAAF ths=M_PI;

   void processing()
   {
     XAAF fvco=f_0-Kvco*v_1-Kmod*v_2-Ktune*v_3;

     double curr_time=sc_timestamp().to_seconds();

     XAAF Phi_v=Phi_v+2*M_PI*fvco*(curr_time-prev_time);

     if (((Phi_v-ths)>=0)!=false)
     {
       VCO_3G2_N=!(Phi_v-ths)>=0)*0+(Phi_v-ths)>=0)*1;
       ths=ths+cond*2*M_PI;
     }
     else
       VCO_3G2_N=0;

     prev_time=curr_time;
   }
};
```

Figure 6.11: VCO implementation with XAAF signal data type

and *ths* is equal to:

$$\Phi_v = !cond * 0 + cond * 1 \tag{6.-12}$$

$$ths = !cond * ths + cond * (ths + 2\pi). \tag{6.-11}$$

where $cond = (\Phi_v - ths) \geq 0$. The operator $+$ with XAAF condition values is equivalent to the disjunction operator $\vee$ in Boolean algebra. Hence, it holds that $!cond + cond = 1$ and Eq. 6.-11 can be re-written as:

$$ths = ths + cond * 2 * \pi.$$

Frequency divider. In [56, 86, 105] a frequency divider is modeled as a simple division by a constant ratio N. This work uses a realistic model for a divider based on a counter. The counter is triggered on the positive edge of the oscillator signal that represents its clock signal clk. When the positive edge occurs, the counter value is increased by one. When it reaches the value of N (here $N = 110$), its value is reset to zero and the value of output signal is changed from 0 to 1. Figures 6.12-6.13 show the implementation of the frequency divider as a counter. Figure 6.12 shows the counter implementation for numeric simulation; `bool` type of clk signal. Figure 6.13 shows small modifications of the counter implementation for symbolic code execution.

As described in Section 4.1 and given by Table 4.1 the equality operator $==$ returns two possible values: `true` or `false`. However, due to uncertain active time of the oscillator signal the counter value is also uncertain. To evaluate the condition $(count - M6)\%32 == 0$ for an uncertain counter value, it is checked if $(count - M6)\%32$ is ≤ 0 and ≥ 0:

$$(count - M6)\%32 \leq 0 * (count - M6)\%32 \geq 0.$$

```
SC_MODULE(FD)
{
   sc_in<bool> clk;
   sc_out<bool> FD_out;
   ...
   bool out=0;

   if (clk==1)
   {
     if (count>=110)
     {
       count=0;
       out=!out;
     }
     else
     {
       if ((count-64)%32==0 && (count-64)>=32)
         out=!out;
     }
     FD_out.write(out);
     count=count+1;
   }
};
```

Figure 6.12: Implementation of a frequency divider for numeric simulation

The conjunction operator for `false` and `true` condition values corresponds to multiplication operation of two XAAFs whose possible values are 0, 1 or both.

Note that the output signal value is written to the output only if the clock signal *clk* is 1. To avoid writing to the output when clock may also be 0 (in the case clock takes both values clk={0, 1}), the output value is multiplied with *clk*. In this way the bottom D flip-flop of PFD is sensitive only on the changes of the counter output, triggered on the positive edge of the oscillator signal.

Phase-frequency detector-PFD. The basic structure of the PFD circuit is shown in Figure 6.14. It is composed of two D flip-flops and one logic AND circuit. The role of PFD is to generate the control signals for the charge pumps. The top D flip-flop is activated on the positive edge of the reference signal Ref and controls the top charge-pump current source. The oscillator signal whose frequency is divide by N VCO_div activates the bottom D flip-flop. The bottom flip-flop generates the control signal for the bottom charge-pump current source. When the outputs of both flip-flops are active, the AND circuit after some delay activates the flip-flop reset signals switching off both charge pumps. The delay of AND gate is $0.5ns$.

The implementations of the detector flip-flops and logic gate are shown in Figures 6.15-6.16. The left side of Figure 6.15 shows the D flip-flop implementations for pure numeric simulations (when the oscillator phase Φ_v

```
SC_MODULE(FD)
{
   sc_in<XAAF> clk;
   sc_out<XAAF> FD_out;
   ...
   XAAF out=0;

   if (clk!=0)
   {
     XAAF cond=(count>=110);

     if (cond==true)
     {
       count=0;
       out=!out;
     }
     else if (cond!=false)
       count=!cond*count+cond*0;

     XAAF cond1=((count-64)%32>=0*(count-64)%32<=0);
     XAAF cond2=(count-64)>=32);

     if ((cond1*cond2)!=false)
       out=!(cond1*cond2)*out+(cond1*cond2)*!out;

     FD_out.write(out*clk);
     count=!clk*count+clk*(count+1);
   }
};
```

Figure 6.13: Implementation of a frequency divider for symbolic simulation

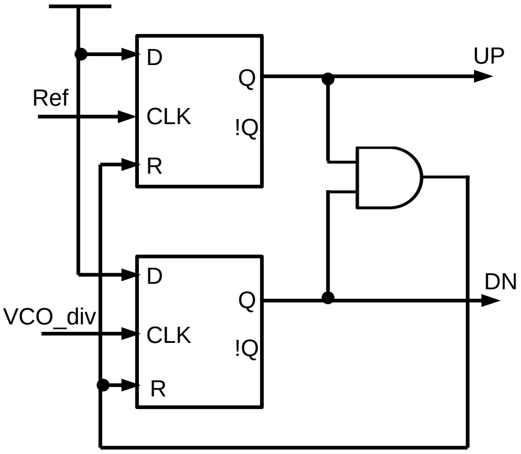


Figure 6.14: The structure of PFD circuit

value is a single value of type `double`). To handle the range of oscillator phase Φ_v values, the implementation is modified applying the same rules as for the oscillator circuit. The code instrumentation is shown on the right side of Figure 6.15. Possible values of output signals of D flip-flops on the right

```
SC_MODULE(DFF)
{
   sc_in<bool> CLK; // clock signal
   sc_in<bool> R; // reset signal
   sc_out<bool> Q; // output
   ...
   void processing()
   {
     if (R==1)
       Q=0;
     else if (CLK==1)
       Q=1;
   }
};
```

```
SC_MODULE(DFF)
{
   sc_in<XAAF> CLK;
   sc_in<XAAF> R;
   sc_out<XAAF> Q;
   ...
   void processing()
   {
     if (R!=0)
     {
       if (CLK!=0)
         Q=!CLK*Q+CLK*1;
       Q=!R*Q+R*0;
     }
     else if (CLK!=0)
       Q=!CLK*Q+CLK*1;
   }
};
```

Figure 6.15: Implementation of a D flip-flop for symbolic simulation

side of Figure 6.15 are 0, 1 but also $\{0,1\}$. For these values the multiplication operation of two XAAFs corresponds to conjunction operation $\wedge$ of two Boolean values. Hence, the AND operation of the logic gate is implemented as a simple multiplication $*$ of PFD output signals (UP and DN).

Charge Pump-CP. The charge pump is composed of two current sources that are controlled by the outputs of PFD UP and DN. If the control signals are one, the current sources are on and the charge is pumped into or out of the capacitors. If the control signals are zero, the current values are zero.

Hence, the behavior of the current sources can be described with the following equations:

$$I_{up} = UP * I_{inc}$$
$$I_{dn} = DN * I_{dec}$$

where I_{inc} and I_{dec} are generated by top and bottom current sources, respectively. For the considered circuit the current values are the same intensity but different sign. Hence, it holds: $I_{inc} = -I_{dec}$. The output current of the charge pump is computed as the sum of the currents obtained from both sources:

$$CP\_I\_OUT = UP * I_{inc} + DN * I_{dec}.$$

The considered PLL circuit has two charge pumps. The controlled signals of the second charge pump are inverted. Hence its output current is:

$$CP\_I\_OUTB = UP * I_{dec} + DN * I_{inc}.$$

Simulating the PLL circuit over a range of operating conditions, the switching times of current sources may be uncertain. Then, the controlled signals UP and DN can have the values: 0, 1, but also UNKNOWN $X = \{0,1\}$.

```
SC_MODULE(AND)
{
   sc_in<XAAF> UP;
   sc_in<XAAF> DN;
   sc_out<XAAF> OUT;
   ...
   AND(sc_module_name n, double delay_)
   {
     SC_METHOD(processing);
     sensitive << UP << DN;

     SC_METHOD(delay_output);
     sensitive << event_delay;

     delay=delay_;

   }

   void processing()
   {
      output=UP*DN;
      event_delay.notify(delay, SC_SEC);
   }

   void delay_output()
   {
      OUT=output;
   }
};
```

Figure 6.16: Implementation of an AND circuit for symbolic simulation

For example, for $UP = 1$ and $DN = \{0, 1\}$ the currents CP_I_OUT and CP_I_OUTB will have two possible values $\{I_{inc}, 0\}$ and $\{I_{dec}, 0\}$, respectively.

6.2.2 Results of symbolic simulation

The considered PLL circuit should generate the output frequency close to 3.52 GHz; the input reference frequency is 32 MHz and the division ratio is 110. The PLL parameters are shown in Table 6.3. The PLL locking is verified taking the following variations into account:

- variations of initial voltage values due to uncertainties in initial conditions
- 10% tolerances of current values given by the specification
- 10% of tolerances of passive components in TPM filter due to process variations

The PLL circuit is simulated over the ranges of considered variations. Since exact values of these variations are not known and not of interest, they will be referred to as uncertainties. Affine representation of each considered

Table 6.3: PLL parameters

PLL block	Parameters
Charge Pump	$I_{inc} = 40\,\mu\text{A}$ $I_{dec} = 40\,\mu\text{A}$
Low pass filter	$R_{on} = 1\,\Omega$ $R_{off} = 100\,\text{M}\Omega$ $C_p = 300\,\text{fF}$ $R_1 = 120\,\text{k}\Omega$ $C_1 = 80\,\text{pF}$ $v_{lp0} = v_{lp1} = v_{lp2} = 0.6\,\text{V}$
Voltage Controlled Oscillator	$K_{vco} = 100\,\text{MHz/V}$ $K_{mod} = 6\,\text{MHz/V}$ $K_{tune} = 12\,\text{MHz/V}$ $f_0 = 4\,\text{GHz}$
TPM current source	$I_{TPM1} = 40\,\mu\text{A}$ $I_{TPM2} = 10\,\mu\text{A}$
TPM filter	$R_{p1} = 10\,\text{k}\Omega$ $R_{p2} = 10\,\text{k}\Omega$ $C_p = 0.5\,\text{nF}$
DAC current source	$I_{DAC} = 5\,\mu\text{A}$
Phase Frequency Detector	$delay = 0.5\,\text{ns}$
Frequency divider	$N = 110$

uncertainty is given in Table 6.4. The proposed methodology computes the worst-case PLL behavior over the considered ranges using one symbolic simulation run. The PLL behavior is computed for 40000 time steps each equal to $Ts = \frac{1}{f_s}$; f_s is the sampling frequency equal to $20GHz$.

Table 6.4: Parameter uncertainties

Parameter	Uncertainty range	Affine Representation
v_{lp1}	$[0.59, 0.61][V]$	$0.6 + \varepsilon_1 * 0.01$
I_{inc}	$40 + [-0.1 * 40, 0.1 * 40][\mu A]$	$40 + \varepsilon_2 * 0.1 * 40$
I_{dec}	$40 + [-0.1 * 40, 0.1 * 40][\mu A]$	$40 + \varepsilon_3 * 0.1 * 40$
I_{TPM1}	$40 + [-0.1 * 40, 0.1 * 40][\mu A]$	$40 + \varepsilon_4 * 0.1 * 40$
I_{TPM2}	$10 + [-0.1 * 10, 0.1 * 10][\mu A]$	$10 + \varepsilon_5 * 0.1 * 10$
I_{DAC}	$5 + [-0.1 * 5, 0.1 * 5][\mu A]$	$5 + \varepsilon_6 * 0.1 * 5$
R_{p1}	$10000 + [-0.1 * 10000, 0.1 * 10000][k\Omega]$	$10000 + \varepsilon_7 * 0.1 * 10000$
R_{p2}	$10000 + [-0.1 * 10000, 0.1 * 10000][k\Omega]$	$10000 + \varepsilon_8 * 0.1 * 10000$
C_p	$0.5 + [-0.1 * 0.5, 0.1 * 0.5][nF]$	$0.5 + \varepsilon_9 * 0.1 * 0.5$

The total simulation time taking 9 uncertainties into account took 1759 s (around 29 min). The number of non-contiguous regions was 864. Thus, the number of ω symbols required to cover all possible transitions of VCO output signal from positive to negative edge was 10. The PLL circuit locked after 0.9 µs. This is automatically checked using the assertion approach explained in Chapter 5. The assertion describing the PLL locking property is given by the following:

$$F(IN(f_{out}, 3.52 + \varepsilon_1 0.001) \Rightarrow G(IN(f_{out}, 3.52 + \varepsilon_1 0.001)))$$

where:

- F is the temporal operator which assigns that the property in the bracket should hold eventually during simulation
- f_{out} is the output PLL frequency
- $3.52 + \varepsilon_1 0.001$ is the tolerance range around the desired output frequency 3.52 GHz.
- IN is the assertion operator which checks if its first argument f_{out} lies within the specified range $3.52 + \varepsilon_1 0.001$
- G is the temporal operator which assigns that the property in the bracket must hold always during simulation
- $\Rightarrow$ is the implication operator which assigns that when the first condition is satisfied the second condition should always (assigned by G operator) be fulfilled.

The above assertion was checked during simulation reporting the locking time around 0.9 µs. Including the assertion into simulation, the simulation run time increased to 32 min. Thus, the simulation overhead was only 3 min.

As explained in Section 3.4, for the analog part, the space complexity of Affine Arithmetic stays constant during simulation. The run time complexity changes linear with the number of simulation time steps n. Discontinuities introduced by discrete parts may lead to an exponential number of discrete states with n. In reality, this is too pessimistic, since systems usually converge to the stable states where the number of discrete states stay constant or change slightly. This is shown by the simulated PLL circuit; the number of discrete transitions was much lower than 2^n.

6.2.3 Scalability of symbolic simulation

Figure 6.17 shows the dependency of (one symbolic simulation) run time on the number of considered uncertainties. As expected, the run time was the lowest with no uncertainties added to the design. It took only 3 s.

Each uncertainty added to the model may introduce more than one discrete state per time step that must be handled by a symbolic simulation run. The oscillator input v_1 and charge pump currents have the highest impact on the VCO output frequency. Hence, their uncertainties have the highest impact on the uncertainty of PLL output frequency. The uncertainty of PLL frequency introduces uncertainties in the charge pump switching increasing the number of discrete states to be considered within one execution.

The slope of the curve was the highest for those two uncertainties. As the number of uncertainties increased, the ratio between run times was less than or equal to 0.5. As an example, increasing the uncertainty number k from, 5 to 8, the simulation overhead was $\frac{1}{5}$ of the run time for $k = 5$.

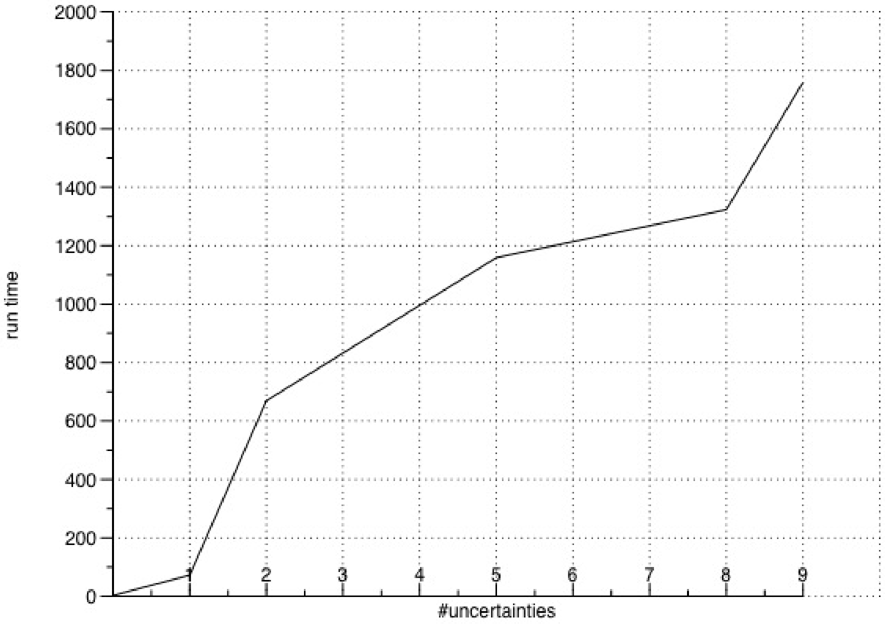


Figure 6.17: Run time versus number of uncertainties

6.2.4 Comparison with numeric simulation

The symbolic results were compared with the random simulation and the simulation based on DoE. Ten random tests are run over randomly chosen parameter values within the ranges given in Table 6.4. The comparison with the symbolic simulation is given in Figure 6.18. The figure clearly shows that the worst-case behavior computed by one symbolic simulation run includes the set of trajectories obtained from numeric simulations. Since the parameter values chosen by random simulation runs were close to each other, the difference between trajectories is hardly visible.

The results found by symbolic simulation are also compared with DoE. Here DoE uses a simple metamodel which estimates the worst-case behavior looking at the corners of ranges given in Table 6.4. Figure 6.19 shows the comparison of 10 DoE runs with the proposed symbolic simulation. The

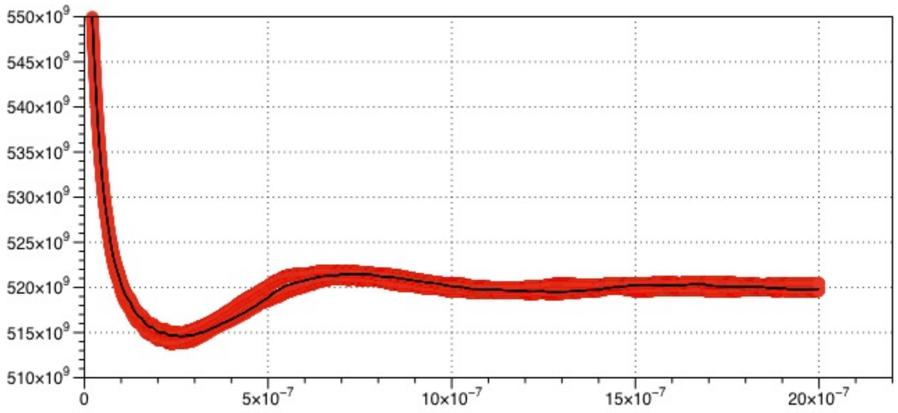


Figure 6.18: PLL output frequency: (red) symbolic simulation, (black) random simulation

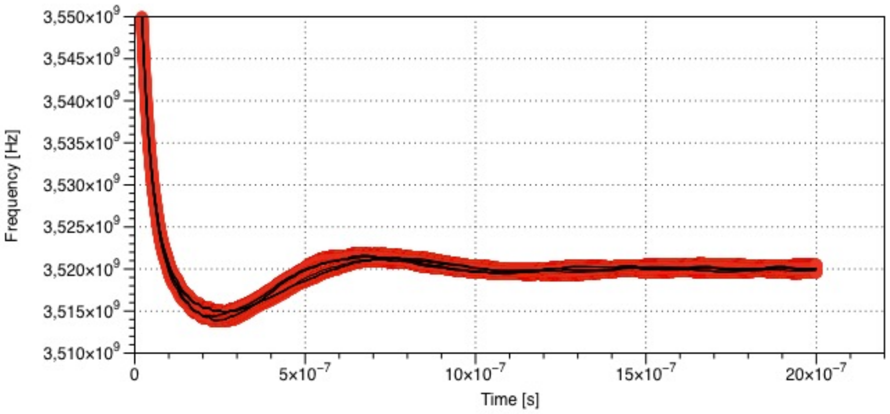


Figure 6.19: PLL output frequency: (red) symbolic simulation, (black) DoE-based simulation

maximum number of DoE runs changes exponentially with the number of uncertainties k. To simulate the PLL behavior over all possible combinations of corners, 2^k runs are required. For the considered circuit this number is equal to $2^9 = 512$. Even then, there is no guarantee that the worst-case behavior is reached. Due to complex dynamics of PLL circuit, the worst-case parameters usually do not lie on the corners and may change over the simulated time horizon. Including LP solver in the XAAF implementation, this information is easily extracted; this is partly the result returned by the solver. Table 6.5 shows the worst-case parameter values returned by the solver for 5 time steps.

Table 6.5: Worst-case parameter values

Time step	minimum frequency [Hz]	maximum frequency [Hz]
0.5 µs	3.5185e9 for v_{lp1}=0.59 V I_{cp} =36.3942 µA lower bounds of other uncertainties	3.52005e9 for v_{lp1} =0.596 554 V I_{cp} =44 µA upper bounds of other uncertainties
0.6 µs	3.52015e9 for v_{lp1} =0.527 568 V I_{cp} =36 µA lower bounds of other uncertainties	3.52149e9 for v_{lp1} =0.596 554 V I_{cp} =44 µA upper bounds of other uncertainties
0.8 µs	3.52075e9 for v_{lp1} =0.61 V I_{cp} =40.8063 µA lower bounds of other uncertainties	3.52148e9 for v_{lp1} =0.59 V I_{cp} =38.6246 µA upper bounds of other uncertainties
1.1 µs	3.51947e9 for v_{lp1} =0.59 V I_{cp} =40.5053 µA I_{TPM1} =44 µA I_{TPM2} =9 µA I_{DAC} =4.5 µA R_1 =11 kΩ R_2 =11 kΩ C_p =0.55 nF	3.52002e9 for v_{lp1} =0.59 V I_{cp} =36.4342 µA I_{TPM1} =36 µA I_{TPM2} =11 µA I_{DAC} =5.5 µA R_1 =9 kΩ R_2 =9 kΩ C_p =0.45 nF
1.3 µs	3.51936e9 for v_{lp1} =0.605 011 V I_{cp} =39.8913 µA I_{TPM1} =44 µA I_{TPM2} =9 µA I_{DAC} =4.5 µA R_1 =11 kΩ R_2 =11 kΩ C_p =0.55 nF	3.52036e9 for v_{lp1} =0.600 36 V I_{cp} =43.9569 µA I_{TPM1} =36 µA I_{TPM2} =11 µA I_{DAC} =5.5 µA R_1 =9 kΩ R_2 =9 kΩ C_p =0.45 nF
1.5 µs	3.51965e9 for v_{lp1} =0.595 493 V I_{cp} =40.4575 µA I_{TPM1} =44 µA I_{TPM2} =9 µA I_{DAC} =4.5 µA R_1 =11 kΩ R_2 =11 kΩ C_p =0.55 nF	3.52045e9 for v_{lp1} =0.600 548 V I_{cp} =44 µA I_{TPM1} =36 µA I_{TPM2} =11 µA I_{DAC} =5.5 µA R_1 =9 kΩ R_2 =9 kΩ C_p =0.45 nF

6.3 Discussion

A literature survey provides a rich set of techniques used to verify the key properties of the same case studies. Simulation-based techniques based on a finite number of simulation runs can analyze the system behavior over a limited set of operating conditions. Thus, the accuracy of verification results is still dependent on the number of simulation runs.

To increase confidence level of verification results, researchers apply the formal methods. The *Checkmate* tool proposed in [54] is one of the first attempts towards formal verification of modulator stability. However, [54] verified this property for only one value of input signal.

In [104] the authors propose combination of IA (Interval Arithmetic) with numeric simulation towards more formal verification of Mixed-Signal circuits. The case study is inspired from [54]. However, the modulator is also verified considering only one value of the input stimuli. Also, the dependency problem of IA restricts the number of time steps for which a system can be verified. This drawback of IA can lead to numerical instability causing the bounds to explode before the simulation ends. To overcome this problem, the authors in [85] proposes the computation of reachable sets combining Interval Arithmetic with Taylor approximations. These symbolic simplifications are expensive operations and increase the computation time significantly in contrast to [61].

The work [53] proposes a method that analyses the modulator behavior over the various set of initial conditions but also the set of input signal values. The modulator is modeled discrete-time hybrid automata. Checking stability property is specified as a MILP (Mixed Integer Linear Programming) problem and solved by the efficient solver MOSEK [106]. The solver finds the worst-case bounds of signal values for the input and initial set. However, this approach has several weaknesses. First, there is no automatic conversion of system dynamics to hybrid-automaton. Second, it is not clear how this approach can be applied in general. It is not easy to express complex system dynamics in the form of linear programming problems; for example the complex structure of the PLL design considered in this thesis.

PLL circuit is the second big case study explored by state of the art methods such as [56, 86, 105]. [56] successfully applies a zonotope-based reachability analysis on a 27 MHz dual-path charge-pump PLL circuit. Ranges of operating conditions are captured with the standard Interval Arithmetic [2]. The work proposes the continuization method to perform efficient reachability analysis for verifying PLL locking property. However, the method requires modeling PLL design with a hybrid-automata. Concretely, Phase-frequency detector is not implemented in a standard way (following it structure given in Fig. 6.14), but using a non-common hybrid automata. To reduce too high overapproximation resulted by Interval Arithmetic (IA), the methodology brings the uncertainty of charge pump switching to the state transition matrix. However, unnecessary overapproximation of switching times computed in a continuous model is still there, since IA can not identify dependency between correlated quantities. It is also not clear how the proposed methodology can be applied on the other PLL designs such as one verified in this thesis. The structure of the considered PLL design is not trivial to be implemented as a mixture of linear continuous dynamics and hybrid automata.

The PLL design from [56] was inspiration for authors in [105] which proposed the qualitative simulation for verification of PLL locking property. [105] considers also the jitter in the reference frequency. However, their tool also does not support a standard way of modeling Mixed-Signal Designs; systems must be modeled in the form of System of Recurrence Equations.

The other tool used for verification of PLL locking was proposes by [86] The tool is called NL-SMT solver and combines SAT solving techniques with Interval Arithmetic. The dependency problem of IA is solved through interval constraint propagation (ICP) [107]. However, the PLL circuit can pass through a large number of switching before it locks to desired frequency. This would require a high number of NL-SMT calls, that would slow down the verification process significantly. Also, simplifications done by ICP can increase the computation time. To guarantee the reasonable time of verification process, SAT solving required simulation assistance. Hence, the solver was invoked only for the regions not covered by simulation.

The XAA approach proposed in this work highlights two advantages of symbolic simulation:

- The method is applied directly on system dynamics; no translation to any kind of formal model such as SRE in [105], SMT constraints in [86] or hybrid automata in [53, 56]. Mixed-Signal designs are modeled following the circuit structure.
- Affine Arithmetic (AA) and its proposed extended form identify the correlation between system quantities. Simplifications used in IA are already the part of computation with (X)AA terms. This fills the gap between high accuracy and low computational cost.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The complexity and heterogeneity of today's mixed-signal designs makes verification a challenge. A particular challenge is the sensitivity of analog parts to even small variations in parameters, inputs, or initial conditions. These variations may degrade the system performance and must be included in the verification process. The proposed approach combines the benefits of formal and simulation methods. The symbolic representation of system variations brought the approach the following advantages:

1. Computation of the safe bounds of the worst-case system response over a set of uncertain values using one simulation run
2. Analysis the impact of uncertainties on the total system response; the extended form proposed in this thesis allows the propagation and computation with uncertainties through all system parts: analog, digital and software.

The proposed methodology is applied on two Mixed-Signal Designs: a third order Δ-Σ Modulator and a PLL circuit of one IEEE 802.15.4 RF transceiver. It verifies the key properties of both circuits: the modulator stability and PLL locking. The circuits are modeled and simulated in the SystemC AMS modeling and simulation environment. The continuous parts are modeled using the AMS extension of SystemC simulator, in particular Timed-Data Flow Model of computation (TDF MOC).

The digital parts are implemented through SystemC MOCs triggered on the certain events (positive edges of predefined clocks, changes in input signals). Interaction between analog and discrete parts is obtained through the existing interfaces that perform conversions from continuous to discrete-event behavior and via versa. The inputs and outputs of the system modules are specified by the input and output ports. Analog modules specify input and output ports via `sca_tdf::sca_in<T>, sca_tdf::sca_out<T>`. Digital parts use `sc_core::sc_in<T>, sc_core::sc_out<T>`. The modules interface with each other via TDF `sca_tdf::sca_signal<T>` and SystemC `sc_core::sc_signal<T>` signals. The Template parameter T specifies the value type of the signal.

XAAF is implemented as a standalone C++ library that provides the C++ class called XAAF. To turn the numerical simulation into a symbolic one, one must specify the XAAF for the value type of the signal T. XAAF allows capturing uncertain values as ranges and computing with them through the whole system including the software and digital parts. Propagation of uncertainties through control flow statements required slight modifications of software codes. The rules for this purpose and their application in the standard C/C++ statements are presented. Performing these rules, arithmetic and comparison operations on XAAFs, one simulation run computes the whole set of possible system outputs over the considered set of uncertain values. The simulation results show that the worst-case behavior is computed for sequences of uncertain values which are not trivial and probably would never be found by pure numeric simulation runs.

7.2 Evaluation of Hypothesis

The simulation and verification of the considered PLL circuit evaluates the first hypothesis introduced in Section 1.3. This hypothesis claims that the presented approach can also deal with complex systems. The considered PLL circuit beside basic parts (PFD, Charge Pump, VCO and frequency divider) counts additional 11 blocks; most of these blocks belong to the digital part of the circuit that controls the basic functionality.

Here, the value of VCO frequency is dependent on three voltage values: the first one is generated by the loop low pass filter, the second one by the digitally controlled current source connected to the first order RC circuit and the third input voltage is the tuned voltage.

The proposed methodology for both case studies computes the worst-case behavior over the considered ranges of input and initial operating conditions. In addition, LP solver allows to extract the sequence of input, initial and parameter values which lead to the worst-case behavior. Due to nonlinear behaviors of discrete parts these values do not often lie on the corners and their values may differ from time step to time step. This is shown in Table 6.2 and Table 6.5. Comparison with numeric simulation-based techniques based on Monte-Carlo and DoE shows the correctness of the hypothesis 2 that claims that the symbolic response encloses the set of numeric trajectories.

7.3 Future work

Future work addresses the following points:

- **Combination of BDDs with Affine Arithmetic**. Here, the idea and the structure of Binary Decision Diagrams can be applied. The set of Boolean values for the representation of terminal node values can be replaced with the set of Affine Forms that also cover the values 0 and 1. The arithmetic representation of condition values in control flow statements using ω symbols can be replaced with BDD structure: each BDD node would represent the corresponding condition and possible condition values $\{0, 1\}$ the terminal BDD nodes. The conditions saved in the intermediate nodes would be specified as constraints respect to which LP solver would find the exact bounds of affine ranges and hence the worst-case bounds of the total system response. Uncertain values $X =$ `{false, true}` of evaluated conditions would represent `false` and `true` child of the node, respectively. All existing algorithms applied on Reduced BDDs for the detection of redundant nodes and their elimination would also be here applied. Additional advantage of this structure is an easy implementation of logical operators.

- **Automatic Instrumentation of Control Flow Statements**. The rules for the instrumentation of control flow statements, presented in this thesis, can be applied in general. However, for huge software codes the manual modifications of existing implementations consumes a lot of time. Fortunately, the software parts that control analog functionality are usually not complex and can be handled in a reasonable time. However, automation of this step would certainly make this approach more attractive.
- **Investigation of the method applicability in other types of systems**. The advanced technology brings us to the modern world where computers are embedded in almost any physical system from cars, airplanes, robots, smart homes, to even social systems. Such systems are known as Cyber-Physical Systems [108]. With the increased applicability of these systems in everyday life, the functional safety is of crucial importance. Faults, unforeseen scenarios and parameter variations must be included in the verification process and seen as part of regular operation. Experiences show that the failing of these systems is in most cases caused by interactions of multiple variations of different kind. These interactions are often too complex to be covered by simply running multiple tests. There is a demand for exhaustive methods. One interesting alternative could be symbolic simulation presented in this thesis. Here, the plan is to investigate the applicability of the proposed extended symbolic approach for modeling such variations and handling their interaction.

Last but not least, the extension of Affine Arithmetic towards covering probabilistic uncertainties could be an interesting work for the future research.

Appendix A

The XAAF Library

The `XAAF` library is a standalone C++ library that provides the `XAAF` class. To use the library one needs to include the header file "xaaf.h" and link the source files to the compiled library libxaaf.a for Linux/Unix or libxaaf.lib for Windows. The instantiation of and computation with `XAAF` terms is possible through the following methods and functions.

A.1 Instantiation of XAAF terms

The `XAAF` can be instated using one of three available constructors:

- `XAAF(double);`

  This method is used to instance an `XAAF` term as a single value with no ω symbols. A set of double values represents a subset of `XAAF` set.
- `XAAF(const AAF &);`

  This method is used to instance an `XAAF` term as a single range with no ω symbols. The range is represented with `AAF` (affine arithmetic form). `AAF` is implemented as `AAF` C++ class which is integrated in `XAAF` class. A set of `AAF` terms represents a subset of `XAAF`.
- `XAAF(const XAAF &);` – Copy constructor

A.2 Overloaded C++ operators

Computation with `XAAFs` requires overloading of existing C++ operators. The following part lists all overloaded operators available in th `XAAF` class.

- Assignment operator =

  In the `XAAF` library the assignment operator is overloaded for `double`, `AAF` or `XAAF` as the right operand:

  `XAAF & operator = (const double);`

  This operator assigns a `double` value to an `XAAF` variable

`XAAF & operator = (const AAF &);`

This operator assigns a range represented with `AAF` to an `XAAF` variable.

`XAAF & operator = (const XAAF &);`

This operator assigns an `XAAF` value to an `XAAF` variable.

- Stream extraction operator <<

  This operator is used to print an `XAAF` to the standard output `std::cout`. Since the `XAAF` variable is on the right side of the operator, it is implemented as a friend function:

  `friend std::ostream & operator « (std::ostream &, const XAAF &);`

- Arithmetic operators

  The overloaded arithmetic operators allow performance of standard mathematical operations with `XAAF`. In the `XAAF` library the following binary operations are implemented:

  1. Addition
  2. Subtraction
  3. Multiplication
  4. Division

  Addition operation is allowed through the following methods:

  `XAAF operator + (const XAAF &) const;`

  This operator performs addition of two `XAAF` variables. It is implemented as a class method since both operands are `XAAF`.

  `XAAF operator + (double) const;` – addition with scalar value of `double` type. It is implemented as a class method since the left operand is `XAAF`.

  `XAAF operator + (const AAF &) const;` – addition with a range represented with `AAF`. It is implemented as a class method since the left operand is `XAAF`.

  All above binary operators have also the corresponding compound assignment and hence they are also overloaded. For addition it is the operator ±. These operators are always implemented as methods since the left operand is always a variable of `XAAF` data type.

  To allow computation with XAAFs where the left operands are not XAAF data type, the following non-member functions are also the part of XAAF library.

  `XAAF operator + (double, const XAAF &);` – addition with a scalar value of `double` type where the scalar value is the left operand. Since the left operand is not of type `XAAF`, it is implemented as non-member function.

`XAAF operator + (const AAF &, const XAAF &);` – addition with `AAF` as the left operand. Since the left operand is not of type `XAAF`, it is implemented as the non-member function.

The same holds for other binary operators.

In addition, `XAAF` class provides two unary operators:

`XAAF operator - () const;` and

`XAAF operator ! () const;`

The return value of operator - () is equal to the multiplication of `XAAF` variable with -1. The unary operator ! () is negation of XAAF value on which the operator is applied. Thus, this operator is applicable only on XAAF whose value is 0, 1 or both $\{0, 1\}$.

- Relational operators $\{<, \leq, \geq, ==, !=\}$

  In order to allow comparison of two `XAAF` terms, the standard relational operators are overloaded. All operators except == and != are overloaded to return `XAAF` that can take three possible values: `false`, `true` or both `{false, true}`. The equality operator == returns `true` only if both operands are equal, otherwise `false`. The equality operator != is the negation of ==.

A.3 Retrieving information about XAAF terms

This section gives a list of methods which allow a user to retrieve basic information about `XAAF` terms.

- Method **getlength** : `unsigned getlength(void) const;`

  This method has no input parameters and returns the number of ω symbols contained in the `XAAF` for which the method is called.

- Method **getmean**: `AAF getmean(void) const;`

  This method returns a center value of the `XAAF` for which the method is called.

- Method **getIndexes**: `unsigned * getIndexes(void) const;`

  This method returns an array of indexes of all deviation symbols in the `XAAF` for which the method is called.

- Method **getMax**: `double getMax(void) const;`

  This method returns a total maximum of the `XAAF` for which the method is called.

- Method **getMin**: `double getMin(void) const;`

  This method returns a total minimum of the `XAAF` for which the method is called.

- Method **getFirstIndex**: `unsigned getFirstIndex(void) const;`

  This method returns an index of the first deviation symbol in the `XAAF` for which the method is called.

- Method **getLastIndex**: `unsigned getLastIndex(void) const;`
  This method returns an index of the last deviation symbol in the `XAAF` for which the method is called.
- Method **xaafprint**: `void xaafprint(void) const;`
  This method prints the `XAAF`, for which the method is called, to the standard output `std::cout`.

List of Figures

List of Tables

Abbreviations

AA	Affine Arithmetic
AAF	Affine Arithmetic Form
A/D	Analog to Digital
ADT	Abstract Data Type
ADC	Analog to Digital Converter
AMS	Analog Mixed-Signal System
ACTL	Analog Computation Tree Logic
AQTS	Approximated Quotient Transition System
BMC	Bounded Model Checking
BDD	Binary Decision Diagram
CTL	Computation Tree Logic
DAE	Differential Algebraic Equation
DoE	Design of Experiments
DOE	Differential Ordinary Equation
DSP	Digital Signal Processing
DE	Differential Equation
D/A	Digital to Analog
DFF	D Flip-Flop
ELN	Electrical Linear Network
FFT	Fast Fourier Transform
GSTE	Generalized Symbolic Trajectory Evaluation
GLPK	GNU Linear Programming Kit
HA	Hybrid Automata
HDL	Hardware Description Language
IIR	Infinite Impulse Response
IA	Interval Arithmetic
ICP	Interval Constraint Propagation
LHA	Linear Hybrid Automata
LHPN	Labeled Hybrid Petri Net
LLVM	Low Level Virtual Machine
LTL	Linear Temporal Logic
LSF	Linear Signal Flow
LP	Linear Programming

MOC	Model of Computation
MTA	Monitoring Timed Automata
MSA	Mixed Signal Assertions
MNA	Modified Node Analysis
$OBDD$	Ordinary Binary Decision Diagram
ODE	Ordinary Differential Equation
PSL	Property Specification Language
PLL	Phase-locked Loop
RF	Radio Frequency
SAT	Satisfiability Theory
SMT	Satisfiability Modulo Theories
SVA	SystemVerilog Assertions
STL	Signal Temporal Logic
SRE	System of Recurrence Equations
$SISSI$	SystemC Intermediate Verification Language Symbolic Simulator
TDF	Timed Data Flow
$THPN$	Timed Hybrid Petri Net
VCO	Voltage-Controlled Oscillator
XAA	Extended Affine Arithmetic
$XAAF$	Extended Affine Arithmetic Form
$XAA + A$	Extended Affine Arithmetic Assertions

Literature

[1] M. Andrade, J. Comba, and J. Stolfi, “Affine Arithmetic (Extended Abstract),” *Interval ’94, St.Petersburg, Russia*, 1994.

[2] R. E. Moore, *Interval Analysis.* Eaglewood Cliffs, NJ: Prentice-Hall, 1966.

[3] J. L. Comba and J. Stolfi, “Affine arithmetic and its Applications to Computer Graphics,” in *SIBGRAPI’93*, 1993, pp. 9–18.

[4] C. F. Fang, R. A. Rutenbar, M. Püschel, and T. Chen, “Toward Efficient Static Analysis of Finite-Preicision Effects in DSP Applications via Affine Arithmetic Modeling,” in *Proceedings of the 40th annual Design Automation Conference (DAC)*, 2003, pp. 496–501.

[5] C. F. Fang, T. Chen, and R. A. Rutenbar, “Floating-Point Error Analysis Based On Affine Arithmetic,” in *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 2, 2003, pp. 561–564.

[6] F. Fang, T. Chen, and R. A. Rutenbar, “Lightweight Floating-Point Arithmetic: Case Study of Inverse Discrete Cosine Transform,” *EURASIP Journal Sig. Proc.: Special Issue on Applied Implementation of DSP and Communication Systems*, vol. 9, pp. 879–892, 2002.

[7] A. Lemke, L. Hedrich, and E. Barke, “Analog Circuit Sizing Based on Formal Methods Using Affine Arithmetic,” in *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2002, pp. 486–489.

[8] C. Grimm, W. Heupke, and K. Waldschmidt, “Refinement of Mixed-Signals Systems with Affine Arithmetic,” in *Design, Automation and Test in Europe.* IEEE Comput. Soc, 2004, pp. 372–377. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1268875

[9] D. Grabowski, C. Grimm, and E. Barke, “Semi-Symbolic Modeling and Simulation of Circuits and Systems,” in *IEEE International*

Symposium on Circuits and Systems (ISCAS). IEEE, 2006, pp. 983–986. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1692752

[10] C. E. Clark, "Importance sampling in Monte Carlo analysis," *Operations Research*, vol. 9, no. 5, pp. 603–620, 1961.

[11] D. E. Hocevar, M. R. Lightner, and T. N. Trick, "A Study of Variance Reduction Techniques for Estimating Circuit Yields," *IEEE Transactions on Computer Aided-Design*, vol. CAD-2, no. 3, pp. 180–192, 1983.

[12] K. Antreich, H. Gräb, and C. Wieser, "Practical Methods for Worst-Case and Yield Analysis of Analog Integrated Circuits," *International Journal of High Speed Electronics and Systems*, vol. 4(3), pp. 261–282, 1993.

[13] G. E. Müller-L., "Limit parameters: the general solution of the worst-case problem for the linearized case," in *IEEE International Symposium on Circuits and Systems*, vol. 3, 1990, pp. 2256–2259.

[14] S. R. Nassif, A. J. Strojwas, and S. W. Director, "A Methodology for Worst-Case Analysis of Integrated Circuits," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 5, pp. 104–113, Nov. 2006.

[15] M. Rafaila, C. Decker, C. Grimm, and G. Pelz, "Simulation-Based Sensitivity and Worst-Case Analyses of Automotive Electronics," in *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2010, pp. 309–312.

[16] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[17] A. Balivada, Y. Hoskote, and J. A. Abraham, "Verification of transient response of linear analog systems," in *Proceedings of 13th IEEE VLSI Test Symposium (VTS'95)*, 1995, pp. 42–47.

[18] L. Hedrich and E. Barke, "A Formal Approach to Verification of Linear Analog Circuits with Parameter Tolerances," in *Design, Automation and Test in Europe 1998 (DATE '98)*, 1998, pp. 649–654.

[19] L. Hedrich and E. Barke, "A Formal Approach to Nonlinear Analog Circuit Verification," in *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*, 1995, pp. 123–127.

[20] S. Steinhorst and L. Hedrich, *Formal Methods in System Design*, 2010, vol. 36, no. 2, ch. Advanced Methods for equivalence checking of analog circuits with strong nonlinearities, pp. 131–147.

[21] A. Salem, "Semi-Formal Verification of VHDL-AMS Descriptions," in *IEEE International Symposium on Circuits and Systems*, 2002, pp. 333–336.

[22] M. H. Zaki, S. Tahar, and G. Bois, "Formal Verification of Analog and Mixed Signal Designs: Survey and Comparision," in *IEEE Northeast Workshop on Circuits and Systems*, 2006, pp. 281–284.

[23] R. Alur, C. Coucoubetis, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theoretical Computer Science*, vol. 138, pp. 3–34, 1995.

[24] R. Alur, C. Courcoubetis, T. Henzinger, and P.-H. Ho., "Hybrid Automata: An algorithmic approach to the specification and verification of hybrid systems," *Hybrid Systems*, vol. LNCS 736, pp. 209–229, 1993.

[25] R. Alur and D. L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[26] O. Stursberg, S. Kowalewski, and S. Engell, "On the generation of timed discrete approximations for continuous systems," *Mathematical and Computer Models of Dynamical Systems*, vol. 6, pp. 51–70, 2000.

[27] O. Maler and G. Batt, *Formal Methods in Systems Biology.* Springer, 2008, vol. LNCS 5054, ch. Approximating Continuous Systems by Timed Automata, pp. 77–89.

[28] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, *Tools and Algorithms for the Construction and Analysis of Systems.* Springer-Verlag, 1995, vol. LNCS 1019, ch. A User Guide to HYTECH, pp. 41–71.

[29] T. A. Henzinger, "The Theory of Hybrid Automata," in *Proceedings of 11th Annual IEEE Symposium on Logic in Computer Science*, 1996, pp. 278–292.

[30] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "Algorithmic Analysis of Nonlinear Hybrid Systems," *IEEE Transactions on Automatic Control*, vol. 43, pp. 540–554, 1998.

[31] W. Hartong, L. Hedrich, and E. Barke, "On discrete modelling and model checking of nonlinear analog systems," in *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV '02)*, 2002, pp. 401–413.

[32] W. Hartong, R. Klausen, and L. Hedrich, *Advanced Formal Verification.* Kluwer Academic Publishers, 2004, ch. Formal Verification for Nonlinear Analog Systems: Approaches to Model and Equivalence Checking, pp. 205–245.

[33] T. R. Dastidar and P. P. Chakrabarti, "A verification system for transient response of analog circuits using model checking," in *18th International Conference on VLSI Design*, 2005, pp. 195–200.

[34] D. Grabowski, D. Platte, L. Hedrich, and E. Barke, "Time Constrained Verification of Analog Circuits using Model-Checking Algorithms," in *Proceedings of the First Workshop on Formal Verification of Analog Circuits (FAC 2005)*, 2005.

[35] K. Chatterjee, P. Dasgupta, and P. Chakrabarti, "A Branching Time Temporal Framework for Quantitative Reasoning," *Journal of Automated Reasoning*, vol. 30, pp. 205–232, 2003.

[36] R. Alur, C. Courcoubetis, and D. Dill, "Model-Checking for Real-Time systems," in *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, 1990, pp. 414–425.

[37] S. Little, D. Walter, N. Seegmiller, C. Myers, and T. Yoneda, "Verification of Analog and Mixed-Signal Circuits Using Timed Hybrid Petri Nets," in *Proceedings of 2nd Automated Technology for Verification and Analysis (ATVA 2004)*, vol. LNCS 3299, 2004, pp. 426–440.

[38] M. Freibothe, J. Schönherr, and B. Straube, "Formal Verification of the Quasi-Static Behavior of Mixed-Signal Circuits by Property Checking," in *Proceedings of the First Workshop on Formal Verification of Analog Circuits (FAC 2005)*, 2005.

[39] S. Little, N. Seegmiller, D. Walter, C. Myers, and T. Yoneda, "Verification of Analog/Mixed-Signal Circuits Using Labeled Hybrid Petri Nets," in *International Conference on Computer-Aided Design (ICCAD 2006)*, 2006, pp. 275–282.

[40] S. Little, D. Walter, K. Jones, and C. Myers, "Analog/Mixed-Signal Circuit Verification Using Models Generated From Simulation Traces," in *Automated Technology for Verification and Analysis (ATVA 2007)*, 2007, pp. 114–128.

[41] S. Little, D. Walter, K. Jones, C. Myers, and A. Sen, "Analog/Mixed-Signal Circuit Verification Using Models Generated From Simulation Traces," *International Journal of Foundations of Computer Science*, vol. 21, no. 2, pp. 191–210, 2010.

[42] D. Walter, S. Little, N. Seegmiller, C. J. Myers, and T. Yoneda, "Symbolic Model Checking of Analog/Mixed-Signal Circuits," in *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2007, pp. 316–323.

[43] 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, December 2008.

[44] S. Little, D. Walter, C. Myers, R. Thacker, S. Batchu, and T. Yoneda, "Verification of Analog/Mixed-Signal Circuits Using Labeled Hybrid Petri Nets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 617–630, 2011.

[45] R. Kurshan and K. L. McMillan, "Analysis of Digital Circuits Through Symbolic Reduction," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 11, pp. 1356–1371, 1991.

[46] M. R. Greenstreet and I. Mitchell, *Hybrid Systems: Computation and Control (HSCC).* Springer, 1999, vol. 1569, ch. Reachability Analysis Using Polygonal Projections, pp. 103–116.

[47] E. Asarin, O. Bournez, T. Dang, and O. Maler, "Approximate Reachability Analysis of Piecewise-Linear Dynamical Systems," in *Third International Workshop on Hybrid Systems: Computation and Control (HSCC'00)*, vol. LNCS 1790, 2000, pp. 20–31.

[48] E. Asarin, T. Dang, and O. Maler, "The d/dt Tool for Verification of Hybrid Systems," *Computer-Aided Verification*, vol. LNCS 2404, pp. 365–370, 2002.

[49] A. Chutinan and B. H. Krogh, "Approximating quotient transition systems for hybrid systems," in *Proceedings of the American Control Conference*, 2000, pp. 1689–1693.

[50] A. Chutinan and B. H. Krogh, "Verification of infinite-state dynamic systems using approximate quotient transition systems," in *IEEE Transactions on Automatic Control*, vol. 46, no. 9, 2001, pp. 1401–1410.

[51] A. Chutinan and B. H. Krogh, "Computational Techniques for Hybrid System Verification," *IEEE Transactions on Automatic Control*, vol. 48, no. 1, pp. 64–75, 2003.

[52] B. I. Silva and B. H. Krogh, "Formal Verification of Hybrid Systems Using CheckMate: A Case Study," in *Proceedings of the 2000 American Control Conference*, vol. 3, 2000, pp. 1679–1683.

[53] T. Dang, A. Donzè, and O. Maler, *Formal Methods in Computer Aided Design.* Springer Berlin Heidelberg, 2004, vol. LNCS 3312, ch. Verification of Analog and Mixed-Signal Circuits Using Hybrid System Techniques, pp. 21–36.

[54] S. Gupta, B. H. Krogh, and R. A. Rutenbar, "Towards Formal Verification of Analog Designs," in *IEEE/ACM International Conference on Computer Aided Design*, 2004, pp. 210–217.

[55] O. Grunberg and D. E. Long, "Model Checking and Modular Verification," in *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, 1994, pp. 843–871.

[56] 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," in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, November 2011, pp. 659–666.

[57] H. D. Foster, A. C. Krolnik, and D. J. Lacey, *Assertion-Based Design 2nd Edition.* Kluwer Academic Publishers, 2004.

[58] D. A. S. Committee, "IEEE Standard for Property Specification Language (PSL)," Version 1.1 Standard IEEE 1850, Tech. Rep., 2005.

[59] "IEEE Standard for System Verilog Unified Hardware Design, Specification and Verification Language Standard IEEE 1800," Design Automation Standards Committee, Tech. Rep., Nov. 2005.

[60] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *FORMATS/FTRTFT*, vol. LNCS 3253, 2004, pp. 152–166.

[61] G. A. Sammane, M. H. Zaki, Z. J. Dong, and S. Tahar, "Towards Assertion Based Verification of Analog and Mixed Signal Designs Using PSL," in *Forum on Specification and Design Languages 2007 (FDL'07)*, 2007, pp. 293–298.

[62] D. Nickovic and O. Maler, "AMT: A Property-Based Monitoring Tool for Analog Systems," in *Proceedings of 5th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2007)*, vol. LNCS 4763, 2007, pp. 304–319.

[63] R. Mukhopadhyay, S. K. Panda, P. Dasgupta, and J. Gough, "Instrumenting AMS assertion verification on commercial platforms," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, no. 2, p. Article No. 21, March 2009.

[64] A. Jesser, L. Hedrich, S. Lämmermann, R. Weiss, J. Ruf, T. Kropf, W. Rosenstiel, A. Pacholik, and W. Fengler, "Analog Simulation Meets Digital Verification- A Formal Assertion Approach for Mixed-Signal Verification," in *Proceedings of the 14th Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI'07)*, Oktober 2007, pp. 507–514.

[65] Z. Dong, M. H. Zaki, G. A. Sammane, S. Tahar, and G. Bois, "Run-time verification using the VHDL-AMS Simulation Environment," in *Proceedings of IEEE Northeast Workshop on Circuits and Systems*, 2007, pp. 1513–1516.

[66] L. Tan, J. Kim, and I. Lee, "Testing and Monitoring Model-based Generated Program," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, pp. 128–148, 2003.

[67] Z. Dong, M. H. Zaki, G. A. Sammane, S. Tahar, and G. Bois, "Checking Properties of PLL Designs using Run-time Verification," in *Proceedings of Internatonal Conference on Microelectronics (ICM'07)*, Dec. 2007, pp. 125–128.

[68] S. Lämmermann, A. Jesser, M. Rathgeber, J. Ruf, L. Hedrich, T. Kropf, and W. Rosenstiel, "Checking Heterogeneous Signal Characteristics Applying Assertion-Based Verification," in *Frontiers in Analog Circuit Verification-FAC*, 2009.

[69] S. Lämmermann, J. Ruf, T. Kropf, W. Rosenstiel, A. Viehl, A. Jesser, and L. Hedrich, "Towards assertion-based verification of heterogeneuous system designs," in *Proccedings of Design, Automation and Test in Europe 2010 (DATE '10)*, 2010, pp. 1171–1176.

[70] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[71] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Ramakrishnan and J. Rehof, Eds. Springer Berlin Heidelberg, 2008, vol. LNCS 4963, pp. 367–381.

[72] C. Wang, S. Chaudhuri, A. Gupta, and Y. Yang, "Symbolic pruning of concurrent program executions," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 23–32. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595702

[73] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Proceedings of the 18th International Static Analysis Symposium (SAS'11)*, 2011, pp. 95–111.

[74] Y. P. Khoo, B.-Y. E. Chang, and J. S. Foster, "Mixing type checking and symbolic execution," *ACM SIGPLAN Notices - PLDI*

'10, vol. 45, no. 6, pp. 436–447, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1809028.1806645

[75] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[76] J. Jeon, K. K. Micinski, and J. S. Foster, "SymDroid: Symbolic Execution for Dalvik Bytecode," Department of Computer Science, University of Maryland, College Park, Tech. Rep. CS-TR-5022, Jul 2012.

[77] R. E. Bryant, "Can a Simulator Verify a Circuit?" in *Formal Aspects of VLSI Design*. Elsevier, 1986, pp. 125–126.

[78] R. E. Bryant and C.-J. H. Seger, "Formal Verification of Digital Circuits Using Symbolic Ternary System Models," in *DIMAC Workshop on Computer-Aided Verification*, 1990.

[79] C.-J. H. Seger and R. E. Bryant, *Formal Methods in System Design*, 1995, vol. 6, no. 2, ch. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories, pp. 147–190.

[80] A. Jain, "Formal hardware verification by symbolic trajectory evaluation," Ph.D. dissertation, Carnegie-Mellon University, July 1997.

[81] C.-T. Chou, *Computer-Aided Verification*, 1999, vol. LNCS 1633, ch. The Mathematical Foundation of Symbolis Trajectory Evaluation, pp. 196–207.

[82] J. Yang and C.-J. H. Seger, "Introduction to generalized symbolic trajectory evaluation," in *Proceedings of 2001 International Conference on Computer Design (ICCD 2001)*, 2001, pp. 360–365.

[83] C. Wilson, D. L. Dill, and R. E. Bryant, "Symbolic Simulation with Approximate Values," in *Proceedings of 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)*, vol. LNCS 1954, 2000, pp. 470–485.

[84] H. Le, D. Grosse, V. Herdt, and R. Drechsler, "Verifying SystemC using an Intermediate Verification Language and Symbolic Simulation," in *The 50th Annual Design Automation Conference 2013 (DAC '13)*, May 2013, pp. 1–6.

[85] M. H. Zaki, G. A. Sammane, S. Tahar, and G. Bois, "Combining Symbolic Simulation and Interval Arithmetic for the Verification of

AMS Designs," in *7th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2007, pp. 207–215.

[86] L. Yin, Y. Deng, and P. Li, "Verifying Dynamic Properties of Nonlinear Mixed-Signal Circuits via Efficient SMT-Based Techniques," in *International Conference on Computer-Aided Design (ICCAD)*, 2012, pp. 436–442.

[87] W. Heupke, C. Grimm, and K. Waldschmidt, "Semi-Symbolic Simulation of Nonlinear Systems," in *Forum on Specification and Design Languages (FDL)*. ECSI, 2005. [Online]. Available: http://i-tecs.fr/ecsi/libraryV1/uploads/1-AMS17_paper.pdf

[88] J. Stolfi and L. de Figueiredo, "An Introduction to Affine Arithmetic," *TEMA Trend. Mat. Apl. Comput.*, vol. 4, pp. 297–312, 2003.

[89] D. Grabowski and C. Grimm, "Ein Verfahren zur effizienten Analyse von Schaltungen mit Parametervarianzen Symbolische Modellierung von Unsicherheiten," in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, 2006.

[90] L. H. D. Figueiredo and J. Stolfi, "Affine Arithmetic: Concepts and Applications," *Numerical Algorithms*, vol. 37, no. 1-4, pp. 147–158, 2004.

[91] M. Barnasconi and C. Grimm, Eds., *SystemC AMS extensions User's Guide*. OSCI, 2010. [Online]. Available: www.systemc.org

[92] *SystemC 2.0.1 Language Reference Manual*, OSI, 2003.

[93] M. Barnasconi, K. Einwich, C. Grimm, and A. Vachoux, Eds., *Standard SystemC AMS Language Reference Manual*. OSCI, 2010. [Online]. Available: www.systemc.org

[94] D. Grabowski, M. Olbrich, C. Grimm, and E. Barke, "Range Arithmetics to Speed up Reachability Analysis of Analog Systems," in *Forum on Specification, Verification and Design Languages 2007*, 2007.

[95] D. Grabowski, M. Olbrich, and E. Barke, "Analog Circuit Simulation Using Range Arithmetics," in *Proceedings of the 2008 Asia and South Pacific Design Automation Conference (ASP-DAC '08)*. Seoul, Korea: IEEE Computer Society Press, 2008, pp. 762–767.

[96] M. S. Rump and M. Kashiwagi, "Implementation and improvements of affine arithmetic," *Nonlinear Theory and Its Applications, IEICE*, vol. 6, no. 3, pp. 341–359, 2015.

[97] M. Rathmair, F. Schupfer, C. Radojicic, and C. Grimm, "Extended Framework for System Simulation with Affine Arithmetic," in *Forum on Specification and Design Languages.* IEEE, 2012, pp. 168–175. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6337004&contentType=Conference+Publications&refinements=4294557734&queryText=FDL+2012

[98] W. Heupke, C. Grimm, and K. Waldschmidt, "Modeling Uncertainty in Nonlinear Analog Systems with Affine Arithmetic," in *Applications of Specification and Design Languages for SoCs*, A. Vachoux, Ed. Springer Netherlands, 2006, pp. 155–169. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-4998-9_9

[99] W. Krämer, "Generalized Intervals and the Dependency Problem," Bergische Universität Wuppertal, Tech. Rep., 2006.

[100] W. Walker, P. Harremoës, J. Rotmans, J. van der Sluijs, M. van Asseit, P. Janssen, and M. Krayer von Krauss, "Defining Uncertainty: A Conceptual Basis for Uncertainty Management in Model-Based Decision Support," *Integrated Assessment*, vol. 4, no. 1, pp. 5–17, 2003.

[101] B. Widrow and I. Kollár, *Quantization Noise: Roundoff Error in Digital Computation, Signal Processing, Control and Communications.* Cambridge University Press, 2008.

[102] *GNU Linear Programming Kit*, December 2010.

[103] K. J. Åström and T. Hägglund, *PID Controllers: Theory, Design and Tuning (2nd Edition).* Instrument Society of America, 1995.

[104] G. A. Sammane, M. H. Zaki, S. Tahar, and G. Bois, "Constraint-Based Verification of Delta-Sigma Modulators using Interval Analysis," in *50th Midwest Symposium on Circuits and Systems*, 2007, pp. 726–729.

[105] I. Seghaier, H. Aridhi, M. H. Zaki, and S. Tahar, "A Qualitative Simulation Approach for Verifying PLL Locking Property," in *ACM Great Lakes Symposium on VLSI*, 2014, pp. 317–322.

[106] *The Mosek Optimization Toolbox for Matlab Version 3.0 User's guide and Reference Manual*, November 2003.

[107] Tight integration of satisfiability and constraint solving. [Online]. Available: http://isat.gforge.avacs.org/

[108] E. A. Lee, "Cyber physical systems: Design challenges," in *International Symposium on Object/Component/Service - Oriented Real-Time Distributed Computing (ISORC),*, May 2008.

Curriculum Vitae

Name: Čarna Radojičić

Nationality: Serbia

EMail: radojicic@cs.uni-kl.de

Education:

2000-2004: High PTT School, Belgrade

2004-2010: Master of Science, University of Novi Sad,
Faculty of Technical Sciences, Serbia

2012.-exp. 2016: PhD in Computer Science,
Design of Cyber-Physical Systems,
University of Kaiserslautern

Employment and studies abroad:

Sept.2009 - March 2010: Guest Student at the Institute of Computer
Technology, Vienna University of Technology

Okt. 2010 - Nov. 2012: Project assistant at the Institute of Computer
Technology, Vienna Univ. of Technology

2012: Research assistant at Design of Cyber-Physical Systems,
University of Kaiserslautern