March 8 2010

SystemC AMS extensions User's Guide

Abstract

This is the SystemC Analog Mixed Signal (AMS) extensions User's Guide.

Keywords

Open SystemC Initiative, SystemC, Analog Mixed Signal, Heterogeneous Modeling and Simulation.

This page is intentionally left blank.

Copyright Notice

Right to Copy Documentation

Destination Control Statement

Disclaimer

Trademarks

Bugs and Suggestions

Please report bugs and suggestions about this document to:

http://www.systemc.org/

This page is intentionally left blank.

About this document

This user's guide is an informative document explaining how to use the SystemC AMS extensions. This document is part of the release of the SystemC AMS extensions language standard.

Contributors

This document was created under the leadership of the following people:

Martin Barnasconi, NXP Semiconductors (AMS Working Group Chair)
Christoph Grimm, TU Vienna (AMS Working Group Vice-Chair)

The following people have also contributed to the creation of this user's guide:

Markus Damm, TU Vienna
Karsten Einwich, Fraunhofer IIS/EAS
Marie-Minerve Louërat, UPMC
Torsten Maehne, EPFL
François Pecheux, UPMC
Alain Vachoux, EPFL

AMS Working Group

At the time the AMS standard was created, the AMS working group had the following membership:

Bas Arts, NXP Semiconductors
John Aynsley, Doulos
Kenneth Bakalar, Mentor Graphics
Martin Barnasconi, NXP Semiconductors
David Black, XtremeEDA
Christof Bodner, Infineon
Paul Chun, Intel
Julien Denoulet, UPMC
Karsten Einwich, Fraunhofer IIS/EAS
Stefan Erb, Infineon
Alan Fitch, Doulos
Patrick Garda, UPMC
Thorsten Gerke, Synopsys
Mark Glasser, Mentor Graphics
Wolfgang Granig, Infineon
Christoph Grimm, TU Vienna
Eric Grimme, Intel
Philipp Hartmann, OFFIS
Walter Hartong, Cadence
Gino van Hauwermeiren, NXP Semiconductors
Thomas Herndl, Infineon
Martin Klein, NXP Semiconductors
François Lemery, ST Microelectronics
David Long, Doulos
Marie-Minerve Louërat, UPMC
Torsten Maehne, EPFL
Michael Meredith, Forte
Chunduri Mohan, Intel
Josef Münzer, CISC Semiconductor
Abhilash Nair, Texas Instruments
Gerhard Nössing, Infineon
Frank Oppenheimer, OFFIS
Oury Patrick, Cadence
François Pecheux, UPMC
Markus Pistauer, CISC Semiconductor
Rajendra Pratap, Cadence
Vincent Regnauld, NXP Semiconductors
Martin Schell, Infineon
Wolfgang Scherr, Infineon
Martin Schnieringer, VaST
Andreas Schuhai, CISC Semiconductor
Serge Scotti, ST Microelectronics
Pratul Singh, Cadence
David Smith, Synopsys
Aravinda Thimmapuram, NXP Semiconductors
Thomas Uhle, Fraunhofer IIS/EAS
Alain Vachoux, EPFL
Louie Valena, CoWare
Gaurav Verma, Mentor Graphics
Predrag Vukovic, NXP Semiconductors
Charles Wilson, XtremeEDA
Jagan Yeccaluri, Intel

This page is intentionally left blank.

Preface

This user's guide is meant as an introductory guide for electronic system-level engineers and architects who would like to use the SystemC AMS extensions for their system-level design and verification tasks. The main aim is to provide a self-learning guide on how to use the SystemC AMS extensions by explaining the modeling fundamentals and giving examples on how to start with AMS system-level design at higher levels of abstraction. It assumes that the user has some prior knowledge on SystemC modeling and simulation and C++ in general and is familiar with analog/mixed-signal design and modeling.

After going through this guide, the reader should be in a position to start using the SystemC AMS extensions, and should be able to:

- Get insight in the applicable use cases and requirements of the SystemC AMS extensions.
- Understand the introduced models of computation and associated execution semantics.
- Use the language constructs to create discrete-time and continuous-time models at different levels of abstraction.
- Combine SystemC and the AMS extensions to design a mixed-signal system.
- Perform time- and frequency-domain analysis and tracing of AMS signals.

The AMS design methodology, modeling style, and examples given in this user's guide are based on the Open SystemC Initiative AMS language standard. Any simulator implementation compatible with this standard can be used to build and execute these examples.

This document is an informative guide, intended to clarify the usage and *intended* behavior of the SystemC AMS extensions. The precise and complete definition of the SystemC AMS extensions is standardized in the AMS Language Reference Manual.

This page is intentionally left blank.

Contents

Copyright Notice iii

About this document v

Preface vii

1. Introduction 1

1.1. Motivation 1
1.2. SystemC AMS extensions 1
1.2.1. Use cases and requirements 2
1.2.2. Model abstractions 3
1.2.3. Modeling formalisms 3
1.2.4. Time-domain and frequency-domain analysis 4
1.2.5. Language architecture 4

2. Timed Data Flow modeling 7

2.1. Modeling fundamentals 7
2.1.1. TDF module and port attributes 7
2.1.2. TDF model topologies 8
2.1.3. Time step assignment and propagation 11
2.1.4. Multiple schedules or clusters 13
2.1.5. Signal processing behavior of TDF models 13
2.2. Language constructs 14
2.2.1. TDF modules 14
2.2.2. TDF ports 17
2.2.3. TDF signals 20
2.3. Modeling discrete-time and continuous-time behavior 20
2.3.1. Discrete-time modeling 21
2.3.2. Continuous-time modeling 21
2.3.3. Structural composition of TDF modules 26
2.3.4. Multirate behavior 28
2.3.5. Introducing delays 29
2.4. Interaction between TDF and discrete-event domain 30
2.4.1. Reading from the discrete-event domain 30
2.4.2. Writing to the discrete-event domain 31
2.4.3. Using discrete-event control signals 32
2.5. TDF execution semantics 32
2.6. Application examples 33
2.6.1. BASK modulator 33
2.6.2. BASK demodulator 35
2.6.3. TDF simulation of the BASK example 36
2.6.4. Interfacing the BASK example with SystemC 37

3. Linear Signal Flow modeling 41

3.1. Modeling fundamentals 41
3.1.1. Setup of the LSF equation system 41
3.1.2. Time step assignment and propagation 42
3.2. Language constructs 42
3.2.1. LSF modules 42
3.2.2. LSF ports 43
3.2.3. LSF signals 44
3.3. Modeling continuous-time behavior 44
3.3.1. Structural composition of LSF modules 44

3.3.2. Continuous-time modeling 46
3.4. Interaction between LSF and discrete-event or TDF models 47
3.4.1. Reading from and writing to discrete-event models 47
3.4.2. Reading from and writing to TDF models 47
3.4.3. Using discrete-event or TDF control signals 48
3.4.4. LSF model encapsulation 49
3.5. LSF execution semantics 50
3.6. Application examples 50
3.6.1. PID controller 50
3.6.2. Continuous-time sigma-delta modulator 52

4. Electrical Linear Networks modeling 55

4.1. Modeling fundamentals 55
4.1.1. Setup of the equation system 55
4.1.2. Time step assignment and propagation 56
4.2. Language constructs 56
4.2.1. ELN modules 56
4.2.2. ELN terminals 58
4.2.3. ELN nodes 58
4.3. Modeling continuous-time behavior 58
4.3.1. Structural composition of ELN modules 59
4.3.2. Continuous-time modeling 60
4.4. Interaction between ELN and discrete-event or TDF models 61
4.4.1. Reading from and writing to discrete-event models 61
4.4.2. Reading from and writing to TDF models 62
4.4.3. ELN model encapsulation 63
4.5. ELN execution semantics 64
4.6. Application examples 65
4.6.1. POTS front-end 65

5. Small-signal frequency-domain analyses 69

5.1. Modeling fundamentals 69
5.1.1. Setup of the equation system 69
5.1.2. Analysis methods 69
5.2. Language constructs 70
5.2.1. Small-signal frequency-domain description in TDF modules 70
5.2.2. Port access 70
5.3. Utility functions 71
5.3.1. Frequency-domain delay 71
5.3.2. Laplace transfer functions 71
5.3.3. S-domain definitions 72
5.3.4. Z-domain definitions 73
5.3.5. Detection of small-signal frequency-domain analyses 74
5.4. Small-signal frequency-domain analysis with combined TDF, LSF and ELN models 75

6. Simulation and tracing 77

6.1. Simulation control 77
6.1.1. Time-domain simulation 77
6.1.2. Small-signal frequency-domain simulation 78
6.2. Tracing 78
6.2.1. Trace files and formats 79
6.2.2. Tracing signals and comments 80
6.3. Testbenches 82

7. Modeling strategies 85

7.1. Behavioral modeling using the available models of computation 85
7.1.1. Macromodeling with Electrical Linear Networks 86
7.1.2. Behavioral modeling with Linear Signal Flow 87
7.1.3. Behavioral and baseband modeling with Timed Data Flow 89
7.2. Modeling embedded analog/mixed-signal systems 91
7.2.1. Partitioning behavior to different models of computation 91
7.2.2. Modeling of architecture-level properties 92
7.3. Design refinement and mixed-level modeling 93
7.3.1. Mixed-signal, mixed-level simulation 93
7.3.2. Design refinement and use cases 94
7.4. Modeling and coding style 96
7.4.1. Namespaces 96
7.4.2. Dynamic memory allocation 97
7.4.3. Module parameters 98
7.4.4. Separation of module definition and implementation 100
7.4.5. Class templates 101
7.4.6. Public and private class members 102

Appendix A. Language reference 105

A.1. TDF modules 105
A.2. TDF ports 105
A.3. TDF signals 106
A.4. Embedded Laplace transfer functions 106
A.4.1. sca_tdf::sca_ltf_nd 106
A.4.2. sca_tdf::sca_ltf_zp 107
A.4.3. sca_tdf::sca_ss 107
A.5. LSF primitive modules 108
A.5.1. sca_lsf::sca_add 108
A.5.2. sca_lsf::sca_sub 109
A.5.3. sca_lsf::sca_gain 110
A.5.4. sca_lsf::sca_dot 110
A.5.5. sca_lsf::sca_integ 111
A.5.6. sca_lsf::sca_delay 111
A.5.7. sca_lsf::sca_source 112
A.5.8. sca_lsf::sca_ltf_nd 113
A.5.9. sca_lsf::sca_ltf_zp 114
A.5.10. sca_lsf::sca_ss 115
A.5.11. sca_lsf::sca_tdf::sca_gain, sca_lsf::sca_tdf_gain 116
A.5.12. sca_lsf::sca_tdf::sca_source, sca_lsf::sca_tdf_source 116
A.5.13. sca_lsf::sca_tdf::sca_sink, sca_lsf::sca_tdf_sink 117
A.5.14. sca_lsf::sca_tdf::sca_mux, sca_lsf::sca_tdf_mux 117
A.5.15. sca_lsf::sca_tdf::sca_demux, sca_lsf::sca_tdf_demux 118
A.5.16. sca_lsf::sca_de::sca_gain, sca_lsf::sca_de_gain 119
A.5.17. sca_lsf::sca_de::sca_source, sca_lsf::sca_de_source 119
A.5.18. sca_lsf::sca_de::sca_sink, sca_lsf::sca_de_sink 120
A.5.19. sca_lsf::sca_de::sca_mux, sca_lsf::sca_de_mux 121
A.5.20. sca_lsf::sca_de::sca_demux, sca_lsf::sca_de_demux 121
A.6. ELN primitive modules 122
A.6.1. sca_eln::sca_r 122
A.6.2. sca_eln::sca_c 123
A.6.3. sca_eln::sca_l 123
A.6.4. sca_eln::sca_vcvs 124
A.6.5. sca_eln::sca_vccs 125
A.6.6. sca_eln::sca_ccvs 125
A.6.7. sca_eln::sca_cccs 126
A.6.8. sca_eln::sca_nullor 127
A.6.9. sca_eln::sca_gyrator 127

A.6.10. sca_eln::sca_ideal_transformer 128
A.6.11. sca_eln::sca_transmission_line 129
A.6.12. sca_eln::sca_vsource 130
A.6.13. sca_eln::sca_isource 131
A.6.14. sca_eln::sca_tdf::sca_r, sca_eln::sca_tdf_r 132
A.6.15. sca_eln::sca_tdf::sca_c, sca_eln::sca_tdf_c 132
A.6.16. sca_eln::sca_tdf::sca_l, sca_eln::sca_tdf_l 133
A.6.17. sca_eln::sca_tdf::sca_rswitch, sca_eln::sca_tdf_rswitch 134
A.6.18. sca_eln::sca_tdf::sca_vsource, sca_eln::sca_tdf_vsource 134
A.6.19. sca_eln::sca_tdf::sca_isource, sca_eln::sca_tdf_isource 135
A.6.20. sca_eln::sca_tdf::sca_vsink, sca_eln::sca_tdf_vsink 136
A.6.21. sca_eln::sca_tdf::sca_isink, sca_eln::sca_tdf_isink 136
A.6.22. sca_eln::sca_de::sca_r, sca_eln::sca_de_r 137
A.6.23. sca_eln::sca_de::sca_c, sca_eln::sca_de_c 138
A.6.24. sca_eln::sca_de::sca_l, sca_eln::sca_de_l 138
A.6.25. sca_eln::sca_de::sca_rswitch, sca_eln::sca_de_rswitch 139
A.6.26. sca_eln::sca_de::sca_vsource, sca_eln::sca_de_vsource 140
A.6.27. sca_eln::sca_de::sca_isource, sca_eln::sca_de_isource 140
A.6.28. sca_eln::sca_de::sca_vsink, sca_eln::sca_de_vsink 141
A.6.29. sca_eln::sca_de::sca_isink, sca_eln::sca_de_isink 142

Appendix B. Symbols and graphical representations 143

Appendix C. Glossary 145

Index 147

1. Introduction

1.1. Motivation

There is a growing trend for tighter interaction between embedded hardware/software (HW/SW) systems and their analog physical environment. This leads to systems, in which digital HW/SW is functionally interwoven with analog and mixed-signal blocks such as RF interfaces, power electronics, sensors, and actuators, as shown for example by the communication system in Figure 1.1. Such systems are called *Embedded Analog/Mixed-Signal* (E-AMS) *systems*. Examples of E-AMS systems are cognitive radios, sensor networks or systems for image sensing. A challenge for the development of E-AMS systems is to understand the interaction between HW/SW and the analog and mixed-signal subsystems at the architectural level. This requires new means to model and simulate the interacting analog/mixed-signal subsystems and HW/SW subsystems at functional and architectural level.

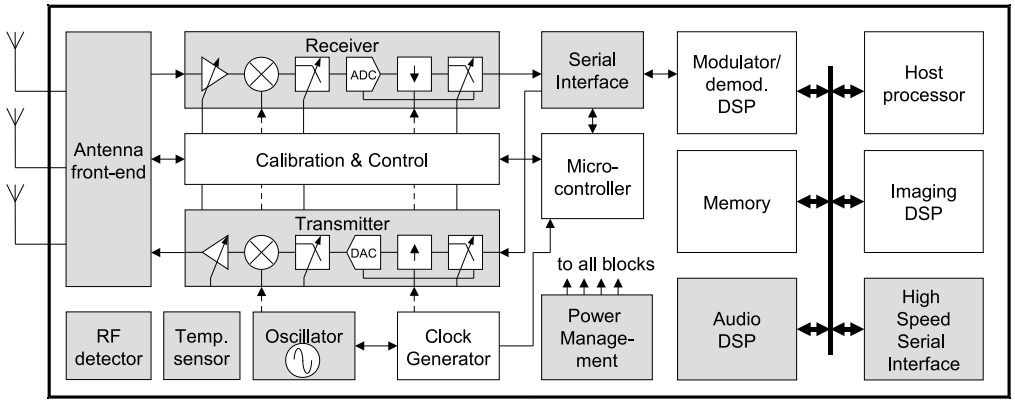


Figure 1.1. A Communication System, example of an embedded analog/mixed-signal architecture

SystemC supports the refinement of HW/SW systems down to cycle-accurate behavior by providing a discrete-event simulation framework. A methodology for generalized modeling of communication and synchronization built upon this framework is also available: Transaction Level Modeling (TLM). It allows designers to perform abstract modeling, simulation, and design of HW/SW system architectures. However, the SystemC simulation kernel has not been designed to handle the modeling and simulation of analog/ continuous-time systems and lacks the support of a refinement methodology to describe analog behavior from a functional level down to the implementation level.

In response to the needs from telecommunication, automotive, and semiconductor industries, AMS extensions are introduced based on SystemC, to provide a uniform and standardized methodology for modeling E-AMS systems.

1.2. SystemC AMS extensions

The SystemC AMS extensions are built on top of the SystemC language standard IEEE 1666-2005 and define additional language constructs, which introduce new execution semantics and system-level modeling methodologies to design and verify mixed-signal systems.

The class definitions provided by the AMS language standard form the foundation for the creation of a C++ class library implementation, which can be used in combination with an IEEE 1666-2005 compatible SystemC implementation. Such an implementation can be used to create AMS system-level models to build an executable specification, to validate and optimize the AMS system architecture, to explore various algorithms, and to provide the software development team with an operational virtual prototype of an entire AMS system, including also the analog functionality. To support these use cases, the SystemC AMS extensions define the necessary modeling formalisms to model AMS system-level behavior at different levels of abstraction.

1.2.1. Use cases and requirements

As depicted in Figure 1.2, the SystemC AMS extensions can be used for a wide variety of use cases such as:

- Executable specification;
- Virtual prototyping;
- Architecture exploration, and
- Integration validation.

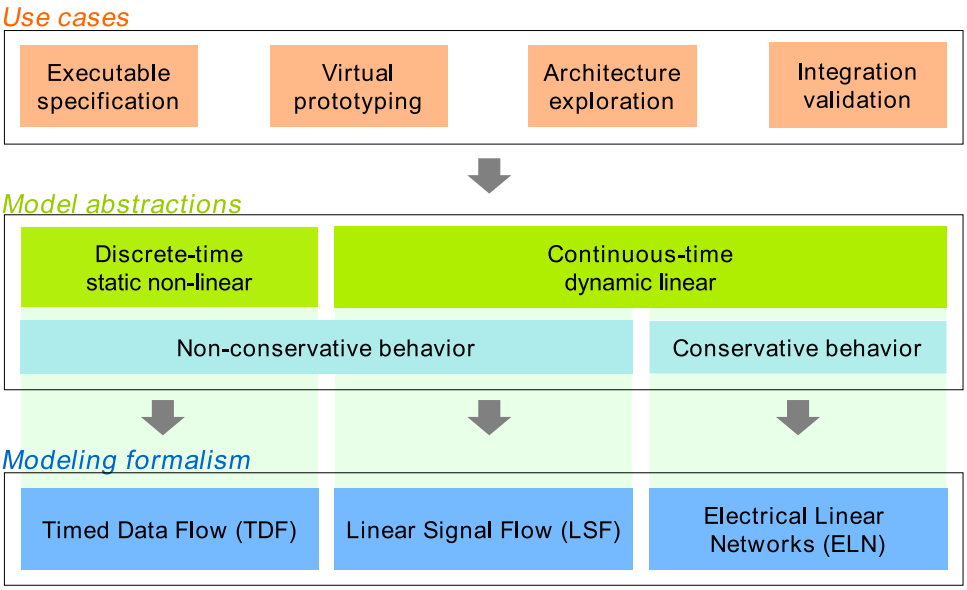


Figure 1.2. Use cases, model abstractions, and modeling formalisms

Executable specification

An executable specification is made to verify the correctness of the system requirement specification by creating an executable description of the system by using simulation. For this use case, models at a high level of abstraction are created, which do not necessarily need to relate to the physical architecture or implementation of the system. The models are, therefore, called functional or algorithmic models.

SystemC and the AMS extensions define both the system-level modeling language and their execution semantics for simulation purposes. They are entirely implemented in the form of C++ libraries, which are linked to the compiled AMS models to create an executable description of the system. This entirely C++-based modeling approach offers a unique flexibility as it allows, e.g., the easy integration of embedded software, 3rd party libraries, and legacy code into the system models.

Virtual prototyping

The virtual prototyping use case aims at providing software developers with a high-level untimed or timed model, that represents the hardware architecture, and provides high simulation speed. Especially for E-AMS systems, where software or firmware is interacting directly with AMS hardware, interoperability using SystemC Transaction-Level Modeling (TLM) extensions is important.

The usage of Timed Data Flow modeling for (over)sampled continuous-time and signal processing behavior provides high simulation speed with appropriate accuracy. In this way, the AMS subsystem can become part of the virtual prototype for further development of the HW/SW subsystem.

Architecture exploration

The architecture exploration use case will evaluate if and how the ideal functions and algorithms defined during the executable specification phase can be mapped onto the envisioned system architecture. The key properties of the system architecture are defined and should match with the actual functionality required.

Architecture exploration is structured in two phases: In the first phase, the executable specification is refined by adding the non-ideal properties of an implementation to get a better understanding of their impact on the overall system behavior. In the second phase, the architecture's structure and interfaces are refined to get a more accurate model by introducing architectural elements and communication between these elements.

Integration validation

After the architecture definition and design of the analog and digital HW/SW components, these components are integrated and their correctness is verified within the overall system. For the integration validation use case, the interfaces of all subsystems must be modeled accurately. The interfaces and data types used in the models should match the physical implementation. For analog circuits this relates to electrical nodes. For digital circuits, this relates to pin accurate buses. For HW/SW systems, TLM interfaces might be appropriate.

1.2.2. Model abstractions

The SystemC AMS extensions add new abstraction methods for system-level modeling and simulation of AMS systems to the existing SystemC framework. The model abstractions supported by the SystemC AMS extensions are based on well-known methods for abstracting analog and mixed-signal behavior. As shown in Figure 1.2, the abstraction levels distinguish discrete-time from continuous-time behavior and non-conservative from conservative descriptions. Chapter 7 will present the available abstraction methods in more detail.

Discrete-time vs. continuous-time descriptions

Discrete-time modeling abstracts signals (e.g., audio or video streams) or physical quantities (e.g., voltages, currents, and forces) as sequences of values only defined at discrete time points. Values may be either real values or discrete values (e.g., integer or logic values). Values between time points are formally not defined, although it is common to consider them as constant. Behaviors are then abstracted as procedural assignments involving sampled signals. The description of static (algebraic) non-linear behaviors (e.g., using polynomials) is supported. Discrete-time modeling is particularly suited for describing signal-processing-dominated behaviors, for which signals are naturally (over)sampled. It can be also used for describing continuous-time behaviors, provided that the discrete abstraction produces reasonable approximations.

Continuous-time modeling gets closer to the physical world, as signals and physical quantities are abstracted as real-valued functions of time. The time is now considered as a continuous value. Behaviors are then described using mathematical equations that can include time-domain derivatives of any order (so-called differential algebraic equations (DAEs) or ordinary differential equations (ODEs)). Equations must be solved by using a dedicated linear or non-linear solver, which usually requires complex numerical or symbolic algorithms. Continuous-time modeling is particularly suited for describing physical behaviors, as it can naturally account for dynamic effects.

Non-conservative vs. conservative descriptions

Continuous-time models can be divided into two classes: non-conservative and conservative models. Non-conservative models express behaviors as directed flows of continuous-time signals or quantities, on which processing functions such as filtering or integration are applied. Non-linear dynamic effects can be properly described, but mutual effects and interactions between AMS blocks, such as impedances or loads, are not naturally supported.

Conservative models are the most detailed continuous-time models at system level and circuit level, as energy conservation laws (Kirchhoff's laws) must be satisfied. As a result, the set of equations to be solved is larger and possibly more complex than the ones inferred by non-conservative models.

1.2.3. Modeling formalisms

The SystemC AMS extensions define the essential modeling formalisms required to support AMS behavioral modeling at different levels of abstraction. These modeling formalisms are implemented by using

different models of computation: Timed Data Flow (TDF), Linear Signal Flow (LSF), and Electrical Linear Networks (ELN).

Timed Data Flow (TDF)

The execution semantics based on TDF introduce discrete-time modeling and simulation without the overhead of the dynamic scheduling imposed by the discrete-event kernel of SystemC. Simulation is accelerated by defining a static schedule, which is computed before simulation starts, and which executes the processing functions of the scheduled TDF modules according to the stream direction of the dataflow. The sampled, discrete-time signals, which propagate through the TDF modules may represent any C++ type. If, e.g, a real-valued type such as *double* is used, the TDF signal can represent a voltage or current at a given point in time. Complex values can be used to represent an equivalent baseband signal. TDF modeling is presented in Chapter 2.

Linear Signal Flow (LSF)

The Linear Signal Flow formalism supports the modeling of continuous-time behavior by offering a consistent set of primitive modules such as addition, multiplication, integration, or delay. An LSF model is made up from a connection of such primitives through real-valued time-domain signals, representing any kind of continuous-time quantity. An LSF model defines a system of linear equations that is solved by a linear DAE solver. LSF modeling is presented in Chapter 3.

Electrical Linear Networks (ELN)

Modeling of electrical networks is supported by instantiating predefined linear network primitives such as resistors or capacitors, which are used as macro models for describing the continuous-time relations between voltages and currents. A restricted set of linear primitives and switches is available to model the electrical energy conserving behavior. ELN modeling is presented in Chapter 4.

1.2.4. Time-domain and frequency-domain analysis

The SystemC AMS extensions support both time-domain (transient) and frequency-domain analysis, by introducing new execution semantics and additional functions for simulation control.

Time-domain simulation can be applied to descriptions made using the TDF, LSF or ELN models of computation. The analysis computes the time-domain behavior of the overall system, possibly composed by different models of computation and could even include descriptions defined in the discrete-event domain. The execution semantics for time-domain simulation of TDF, LSF and ELN models are described in Chapter 2, 3, and 4, respectively.

Frequency-domain simulation can be applied to the same descriptions, combining different models of computation, where the analyses computes the small-signal frequency-domain behavior of the overall system. Besides small-signal frequency-domain analyses, also small-signal frequency-domain noise analysis is available. Chapter 5 will describe both analysis methods in more detail.

The simulation control and signal tracing techniques for time-domain and frequency-domain simulation are presented in Chapter 6. Also the creation and basic structure of testbenches is explained in this chapter.

1.2.5. Language architecture

The SystemC AMS extensions are fully compatible with the SystemC language standard as shown in Figure 1.3. The AMS language standard defines the execution semantics of the TDF, LSF, and ELN models of computation and gives an insight on the underlying enabling technology such as the linear solver, scheduler, and synchronization layer. Currently, the interfaces to and class definitions of this enabling technology is implementation-defined. The AMS designer (end-user) can take advantage of dedicated classes and interfaces to create TDF, LSF or ELN models, by using the predefined modules, ports, terminals, signals and nodes.

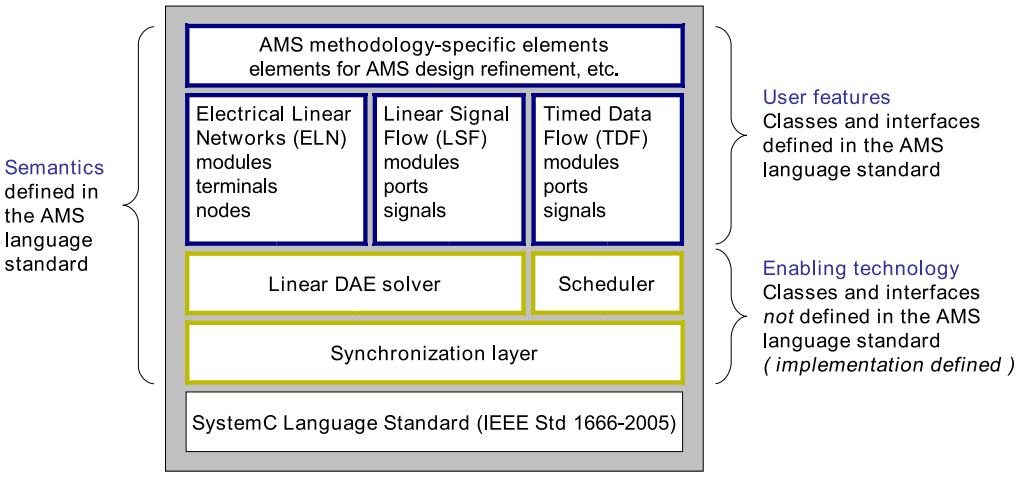


Figure 1.3. Architecture of the AMS language standard

This page is intentionally left blank.

2. Timed Data Flow modeling

2.1. Modeling fundamentals

The Timed Data Flow (TDF) model of computation is based on the well-known Synchronous Data Flow (SDF) modeling formalism. Unlike the untimed SDF model of computation, TDF is a discrete-time modeling style, which considers data as signals sampled in time. These signals are tagged at discrete points in time and carry discrete or continuous values like amplitudes.

Figure 2.1 shows the basic principle of the Timed Data Flow modeling. In this figure, there are three communicating *TDF modules* called A, B, and C. A *TDF model* is composed of a set of connected TDF modules, which form a directed graph called *TDF cluster*. TDF modules are the vertices of the graph, and *TDF signals* correspond to its edges. A TDF module may have several input and output *TDF ports*. A TDF module containing only output ports is also called a producer (source), while a TDF module with only input ports is a consumer (sink). TDF signals are used to connect ports of different modules together.

Each TDF module contains a C++ method that computes a mathematical function f (i.e., f_A, f_B, and f_C), which depends on its direct inputs and possible internal states. The overall behavior of the cluster is therefore defined as the mathematical composition of the functions of the involved TDF modules in the appropriate order, f_C (f_B (f_A (...))), indicated with {A→B→C} in Figure 2.1.

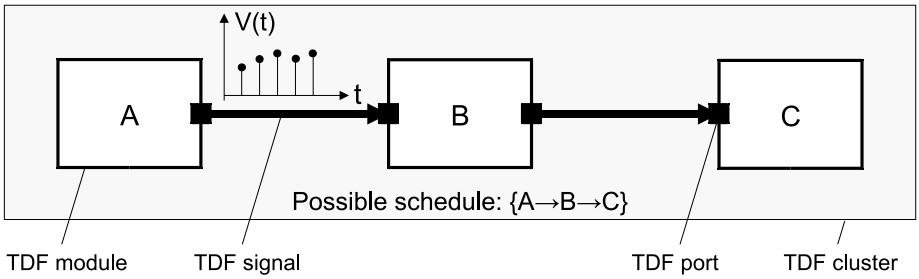


Figure 2.1. A basic TDF model with 3 TDF modules and 2 TDF signals

A given function is *processed* (or "fired" according to the SDF formalism) if and only if there are enough samples available at the input ports. In this case, the input samples are read by the TDF module, where the function uses these values to compute one or more resultants, which are written to the appropriate output ports. In TDF, the number of samples read from or written to the module ports is fixed during simulation, but the numbers of read and written samples by a TDF module are not necessarily equal. A time stamp is associated to each sample using the local TDF module time. The fixed interval between two samples is called *time step*.

2.1.1. TDF module and port attributes

The flexibility and expressiveness of TDF modeling comes from the ability to define the attributes of each TDF module and of each of its ports. In TDF, it is possible:

- To assign a particular time step to a TDF module (*module time step* assignment). Figure 2.2 a shows a TDF module A with a module time step (Tm) of 20 µs.
- To assign a particular time step to a given port of a module belonging to the cluster (*port time step* assignment). Figure 2.2b shows a TDF module B with a TDF input port time step (Tp) of 10 µs.
- To assign a particular rate to a given port of a module belonging to the cluster (*port rate* assignment). Figure 2.2b shows a TDF module B, where at each module activation 2 samples are read (input port rate R set to 2, indicated with R:2).
- To assign a particular delay to a given port of a module belonging to the cluster (*port delay* assignment). Figure 2.2c shows a TDF module C, where at each module activation, the sample corresponding to the previous time step is written (output port delay D set to 1 sample, indicated with D:1).

- To assign a particular time offset to a given port of a module belonging to the cluster (*port time offset* assignment). Figure 2.2d shows a TDF module D, with a module time offset (Tpf) of 1 µs. A time offset can only be assigned to specialized ports to connect to the discrete-event domain, so called TDF *converter ports*.

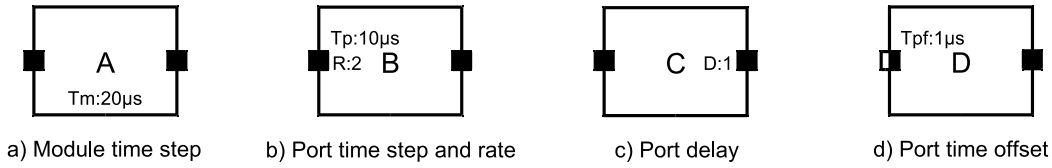


Figure 2.2. TDF module and port attributes

Provided that the attribute assignment on the ports and modules of a TDF model are compatible, the order of activation of the TDF modules in a cluster and the number of samples they read (consume) and write (produce) can be statically determined before simulation starts. Thus, and more formally, a TDF cluster can be defined as the set of connected TDF modules, which belong to the same static schedule. If the assignments are not compatible, the static schedule cannot be established and the TDF cluster is said to be not schedulable (see also Section 2.1.3). Therefore, after the required TDF cluster consistency check, the schedule defines a sequence, in which the algorithmic or procedural description of each TDF module is executed.

The main advantage of this approach is that the execution of TDF models does not rely on the evaluate/ update mechanism of SystemC's discrete-event kernel, and, therefore, can be simulated more efficiently. TDF models are processed independently, using a local time annotation mechanism. Interactions between TDF models and pure SystemC models are supported through specific converter ports, as discussed in Section 2.4.

2.1.2. TDF model topologies

Figure 2.3 shows an example of a TDF model with multirate characteristics. A port rate assignment with rate value 2 (R:2) has been performed on the output port of TDF module A. Ports with no rate attribute are considered to have a rate of 1 (not graphically represented). When module A is activated, 2 samples are written. Since both modules B and C read one sample at each activation, a possible schedule for this TDF cluster is {A→B→C→B→C}.

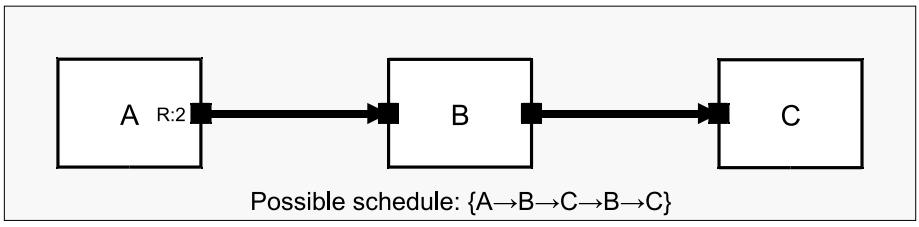


Figure 2.3. Multirate TDF model using port rate assignment

In order to handle TDF models containing loops, it is compulsory to introduce a delay on a module port belonging to one of the modules of the loop. This port delay has to be defined during elaboration of the simulation, to make the static scheduling feasible. A simple example is given in Figure 2.4, without loop, that shows a module A with a delay of one sample associated to the output port (D:1). A possible schedule is {A→B} but also {B→A}, since at module B first activation, the input port of module B will read the sample already available thanks to the assigned delay defined in the elaboration phase.

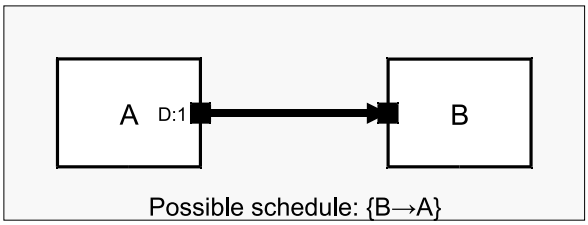


Figure 2.4. TDF model with port delay

The initial value of the sample of a port with a delay is determined by the constructor of the corresponding data types. For basic data types (*double*, *int*, etc.), the constructor does not necessarily assign an initial value, resulting in an undefined value. The user is advised to set the values of the initial samples in case port delays are used.

Figure 2.5 shows an example of a TDF model containing a loop, a quite common situation when dealing with signal processing with feedback. A mandatory port delay assignment with delay value 1 (D:1) has been performed on the output port of TDF module C. Assigning a delay to the output port of module C, allows module B to be "fired" when the first sample of module A becomes available on input in0 of module B. A possible schedule for this TDF model is {A→B→C}.

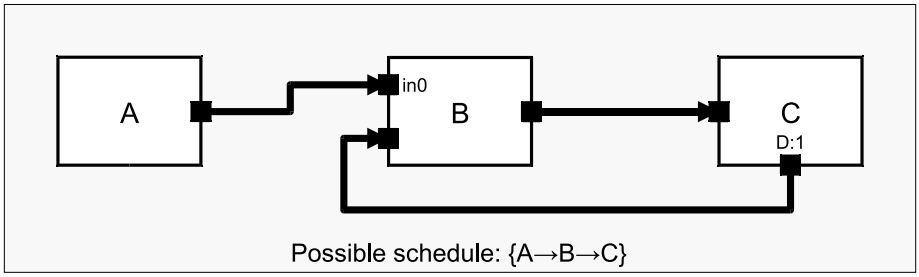


Figure 2.5. TDF model with loop, and port delay assignment

Figure 2.6 shows a more complex example mixing multirate and delay. A possible cluster schedule is {A→B→B→C→D}. Module B is executed twice because of the port rate (R:2) assignments performed on the two connected ports (output port of module A and input port of module C). The port delay assignment on the output port of module D (D:1) is required for the schedule to be computed properly.

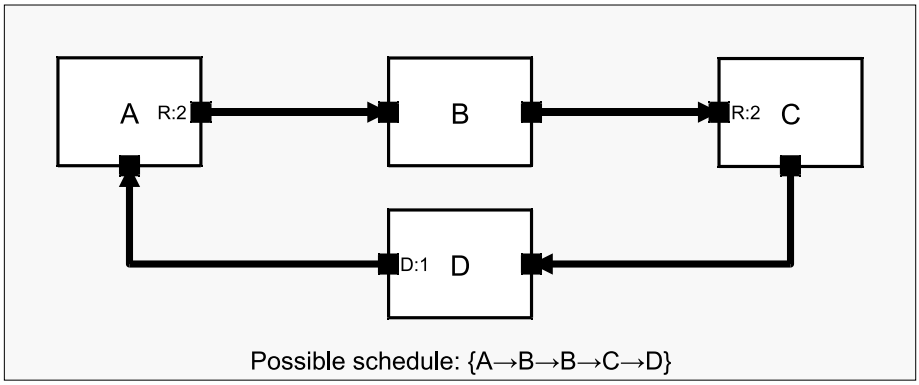


Figure 2.6. Multirate TDF model with loop

Another prerequisite for a proper schedule is that the sum of samples *produced* at the output ports within a loop must be equal to the sum of samples *consumed* by the input ports within the loop. Otherwise, any

finite schedule would accumulate surplus samples somewhere in the cluster when executing it repeatedly. For example, in the case the rate of the input port of module C in Figure 2.6 were changed from 2 to 1, the schedule {A→B→C→D→B→C→D} would result in one extra sample at the output of module D after executing the schedule once (see Figure 2.7)

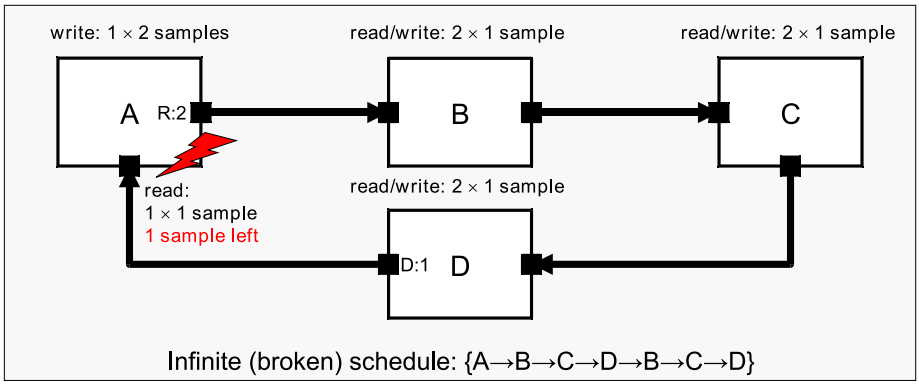


Figure 2.7. Multirate TDF model containing a loop with incompatible rates, resulting in accumulation of samples in the cluster yielding to an infinite (broken) schedule

Figure 2.8 shows how it is possible to connect a TDF model with the discrete-event domain, by means of TDF *converter ports* (indicated with ◧). For example, a discrete-event signal is available at the TDF converter port of TDF module A. Module D has a TDF converter input port, reading a discrete-event control signal. Special care should be taken with the interaction between the TDF and discrete-event domain. This is described in Section 2.4.

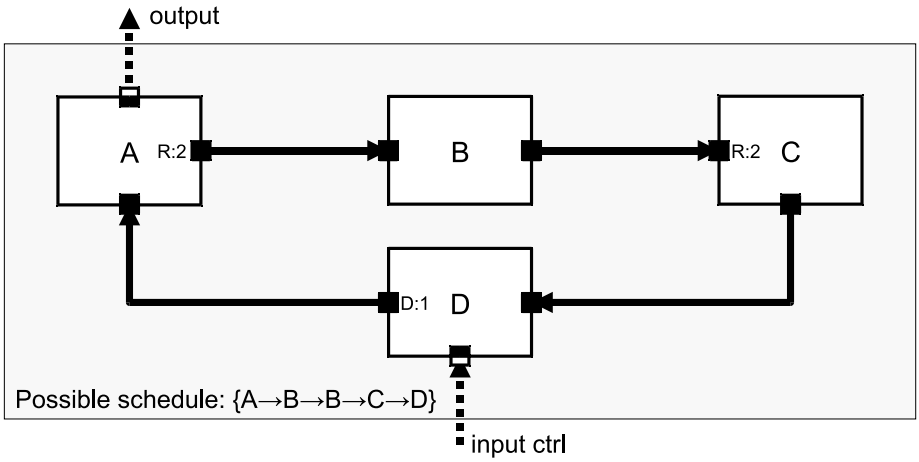


Figure 2.8. TDF model interfacing with discrete-event domain

Another special case is when a TDF model becomes part of a closed loop, which includes a path through the discrete-event domain, as shown in Figure 2.9. The TDF cluster itself contains no loop, so there is no port delay assignment necessary to calculate a valid schedule. Module A reads a sample from the discrete-event domain at the first delta cycle of the time point associated to the sample using a TDF converter input port. Module C writes a sample to the discrete-event domain in the same delta cycle, using a TDF converter output port. Note that TDF samples read from module C and passed through the discrete-event module D to the input of module A will be be delayed by one TDF time step due to the evaluate/update mechanism of the SystemC kernel.

More details on the interaction between the TDF and discrete-event domain is described in Section 2.1.4 and 2.4.

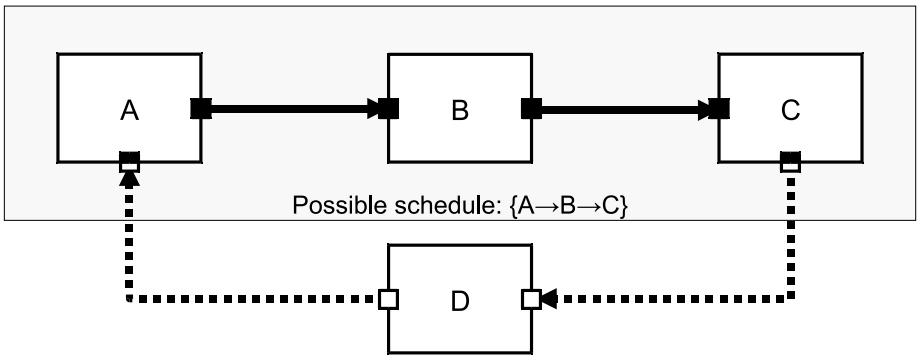


Figure 2.9. TDF model with loop via the discrete-event domain

2.1.3. Time step assignment and propagation

The definition of port rates and delays is very useful to handle different frequency domains within the same TDF model, as well as to create complex TDF module structures involving nested loops. The main point here is that the consistency of a cluster exclusively relies on the compatibility of port rate and delay values and is thus intrinsically independent of the chosen time step (sampling period) to run it. Once this consistency check has been validated for a particular cluster, it may operate at any frequency by means of a port time step assignment or a module time step assignment.

Figure 2.10 illustrates the simplest case, in which all rates are set to 1 (not graphically represented). Starting with a port time step of 10 µs assigned to the input port of module C (denoted as Tp:10µs), this figure shows how this time step value is used to transitively calculate the time steps of the other ports and modules (denoted as italic values *Tp* and *Tm*). When there is no specific rate (R) nor delay (D) assigned to a port, a rate of 1 and a delay of zero samples are assumed by default.

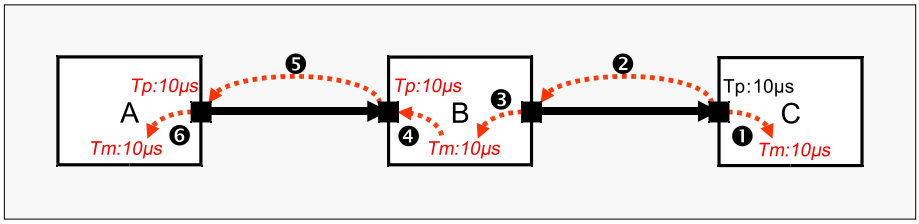


Figure 2.10. Propagation of the time step Tp:10µs set on the input port of module C

The time step propagation is performed upstream and downstream of the target element of the performed time step assignment (port or module) in the TDF model. This process is illustrated by dotted arrows in Figure 2.10. For instance, the port time step assignment on the input of module C propagates downstream by setting the module C time step to 10 µs (*Tm:10µs*, dotted arrow ❶). Similarly, the time step assigned on the input port of module C (Tp:10µs) is propagated upstream to the output port of module B (dotted arrow ❷). Then, the module B time step is assigned with the same time step (*Tm:10µs*, dotted arrow ❸), which is in turn forwarded to the input port of module B (*Tp:10µs*, dotted arrow ❹), to the output port of module A (*Tp:10µs*, dotted arrow ❺), and finally to the module A time step (*Tm:10µs*, dotted arrow ❻).

Consistency of time step assignment and propagation

The example of Figure 2.10 illustrates a propagation example with only one port time step assignment (input port of TDF module C). If the TDF model does not contain any loop, the presented propagation scheme always generates a valid time step assignment, whether the single time step has been assigned to a port or to a module. Once two or more port and/or module time steps have been assigned in a TDF cluster, a consistency check has to be made to ensure their compatibility with the propagated time steps, depending on the port rates.

Figure 2.11 below shows a module, where the input port time step is set to 10 µs (Tp:10µs) with a rate of 2 (R:2), and the module time step is set to 20 µs (Tm:20µs). As the output port rate is not set, it will use the default rate of 1, resulting in an output port time step of 20 µs.

Figure 2.11. Port time step, port rate, and module time step should be consistent

The module time step should be consistent with the rate and time step of any port within a module. The relation between these time steps and rates becomes:

module time step = input port time step · input port rate = output port time step · output port rate

In the example of Figure 2.11, the following relation is checked: 20 µs = 10 µs · 2 = 20 µs · 1.

In the example of Figure 2.12, multiple modules form a cluster, where two time steps are set by the user: the time step of module A is set to 20 µs (Tm:20µs ❶) and the input port time step of module C is set to 10 µs (Tp:10µs ❶). Furthermore, the user has set the rate of the output port of module A to 2 (R:2). Therefore module A is activated two times less frequently than modules B and C, as module A writes 2 samples per activation, see Figure 2.3.

The specified port time step at the input of module C (Tp:10µs ❶) propagates downstream to module C thus setting its time step to 10 µs (*Tm:10µs*, dotted arrow ❷). Similarly, the time step assigned to the input port of module C (Tp:10µs ❶) is propagated upstream to the output port of module B (dotted arrow ❸). Then, the module B time step is assigned with the same time step (*Tm:10µs*, dotted arrow ❹), which in turn is forwarded to input port of module B (*Tp:10µs*, dotted arrow ❺), and propagated upstream to the output port of module A (*Tp:10µs*, dotted arrow ❻). Since the output port rate of module A is 2, the propagated module time step should become 20 µs (Tm:20µs, dotted arrow ❼), which matches with the user specified time step of module A (Tm:20µs ❶).

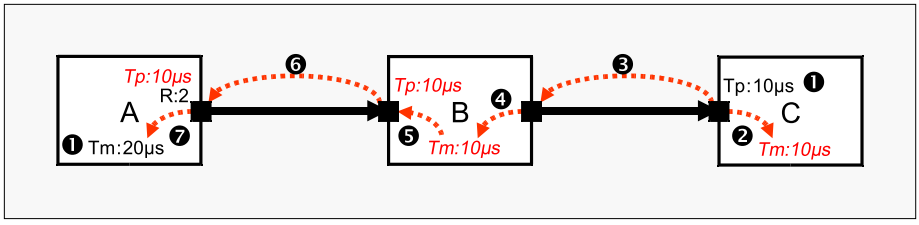


Figure 2.12. Time step propagation for a multirate TDF model with consistent time step assignments done by the user

Figure 2.13 shows the same TDF model with an incompatible time step propagation, which leads to a non schedulable cluster. The expected module A time step, resulting from propagation is 20 µs (*Tm:20µs*, dotted arrow ❽), which is different from the assigned module time step of module A (Tm:10µs ❶). Therefore, no consistent schedule can be derived.

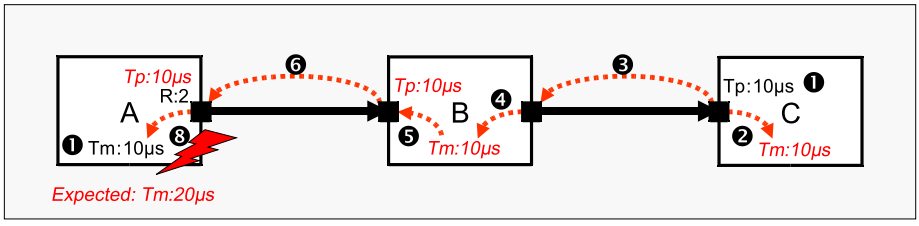


Figure 2.13. Time step propagation for a multirate TDF model
with inconsistent time step assignments done by the user

In the case the TDF model contains loops, the defined port rates, delays, and time steps must be consistent with the time steps propagated through the loop upstream and downstream, to make the TDF model schedulable.

2.1.4. Multiple schedules or clusters

It is possible to have more than one TDF cluster within the same application. In this case, each TDF cluster has its own data flow characteristics (sampling rate, sampling period, etc), scheduling and execution order.

The main element to indirectly change the cluster structure, is to use the TDF converter ports. As explained in Figure 2.8, these ports facilitate an interface to the discrete-event domain and thus define where a static schedule will start or stop. Figure 2.14 shows an example, in which TDF converter ports are used to deliberately split a cluster. Note that the dashed signal indicates the use of a discrete-event signal in between module B and module C.

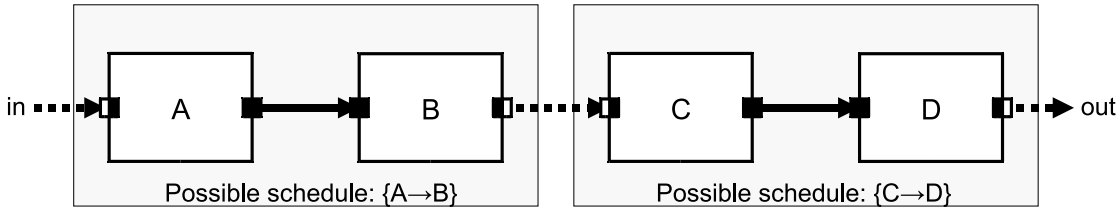


Figure 2.14. Use of TDF converter ports to deliberately split a cluster in two independent ones

Due to the introduction of a discrete-event signal in the chain of modules, the execution of the schedule for each cluster becomes independent. The converter port of module B will write its sample value in the evaluation phase of the SystemC kernel, at the first delta cycle of the associated time point of the sample. The converter port of module C will read a sample, for the corresponding time point, during the same evaluation phase in the same delta cycle. This implies that module C will read the *previous* value from module B, as the value written by module B will only be changed in the update phase of the SystemC kernel, which follows after the completion of the delta cycle's evaluation phase for a certain point in time. This results in an effective delay of one TDF time step for the samples read by module C.

More details on the interaction between the TDF and discrete-event domain is described in Section 2.4.

2.1.5. Signal processing behavior of TDF models

Figure 2.15 illustrates how a cluster of TDF modules processes signals by repetitively activating the processing functions of the contained modules in the order of the derived schedule. It generates samples for each module as a function of time. Because the rates are all set to 1, the processing is obvious: Module A writes a sample at time 0 µs, which is read by module B at time 0 µs, and module B writes a sample at time 0 µs, which is read by module C at time 0 µs. From the perspective of the generated samples, it is important to notice that it is the write operation of the sample produced by module A that actually enables module B to be fired. Respectively, the generation of a sample by module B triggers module C.

The output of module A produces a continuous-value signal (V_{in}), which values are only available at discrete time points. The time step between these samples is equidistant, and defined by the time step of the output port of module A (Tp:10µs). Signal V_{in} is fed into module B, in this example assumed to be a simple amplifier, with a constant gain. The samples of the amplified output signal (V_{out}) become available at the output of module B at the same time steps as module A.

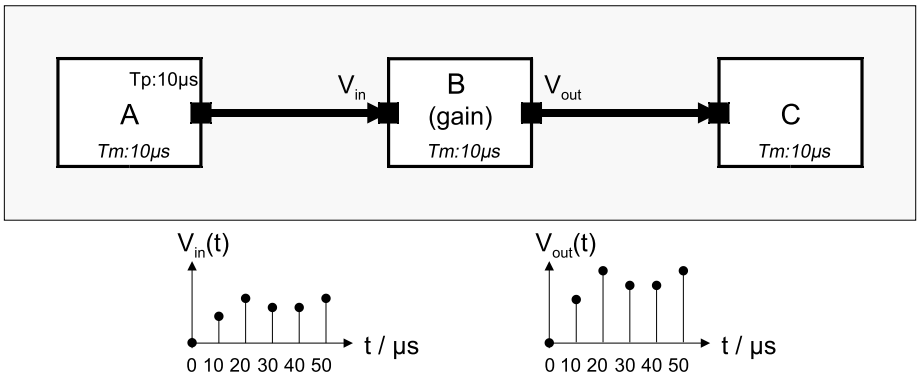


Figure 2.15. TDF module activation (processing) with read and written samples

Besides using TDF modules to describe discrete-time behavior, a TDF module can be used to encapsulate continuous-time behavior. Section 2.3 will explain the usage of TDF to model discrete-time and continuous-time behavior.

2.2. Language constructs

2.2.1. TDF modules

A TDF module is a user-defined primitive module to define discrete-time or to embed continuous-time behavior. The example below shows the typical structure of a TDF module.

```
SCA_TDF_MODULE(my_tdf_module) ❶
{
  // port declarations
  sca_tdf::sca_in<double>  in; ❷
  sca_tdf::sca_out<double> out;

  SCA_CTOR(my_tdf_module) {} ❸

  void set_attributes() ❹
  {
    // module and port attributes
  }

  void initialize() ❺
  {
    // initial values of ports with a delay
  }

  void processing() ❻
  {
    // time-domain signal processing behavior or algorithm
  }

  void ac_processing() ❼
  {
    // small-signal frequency-domain behavior
  }
};

class my_second_module : public sca_tdf::sca_module ❽
{
 public:
  // port declarations
  // ...

  my_second_module( sc_core::sc_module_name ) {} ❾

  // definition of the TDF member functions as done above
  // ...
```



```
};
```

❶ Primitive module declaration using the macro **SCA_TDF_MODULE** to define a new class derived from class **sca_tdf::sca_module**.
❷ A TDF module can have multiple input and output ports. Only TDF ports should be instantiated, see Section 2.2.2.
❸ Mandatory constructor using the predefined macro **SCA_CTOR**, which needs to have the module name as an argument.
❹ Optional member function **set_attributes**, in which TDF module and port attributes can be defined. The user is not allowed to call this member function directly. It is called by the simulation kernel during elaboration.
❺ Optional member function **initialize**, to initialize data members representing the module state and especially the initial samples of ports with assigned delays The user is not allowed to call this member function directly. It is called by the simulation kernel, at the end of elaboration, just before transient simulation starts.
❻ Mandatory member function **processing**, which encapsulates the actual signal processing function. The user is not allowed to call this member function directly. It is called by the simulation kernel as part of time-domain (transient) simulation, where each module activation advances the local module time by the assigned or derived module time step.
❼ Optional member function **ac_processing**, which encapsulates the small-signal frequency-domain (AC) and small-signal frequency-domain noise behavior. The user is not allowed to call this member function directly. It is called by the simulation kernel while executing small-signal frequency-domain analyses (see Chapter 5).
❽ TDF module declaration by creating a new class publicly derived from class **sca_tdf::sca_module**.
❾ Constructor, which always needs to have a parameter of class **sc_core::sc_module_name** to assign a name to the module.

A TDF module contains elements such as ports, signals, parameters, and member functions for time-domain (transient) and small-signal frequency-domain (AC) analyses. Together, these elements implement the behavior of the module.

Module attributes

Module and port attributes such as sampling rate, delay, and time step, can be defined in the member function **set_attributes**. The member function may use any legal C++ statement in addition to the definition of module or port attributes. This member function is called at elaboration time. The example below shows the assignment of a module time step of 10 ms and a delay of one TDF sample to the port *out*.

```
void set_attributes()
{
  set_timestep(10.0, sc_core::SC_MS); // module time step assignment of a of 10 ms

  out.set_delay(1); // set delay of port out to 2 samples
}
```

How to define port attributes inside this member function is explained in Section 2.2.2.

Module initialization

The member function **initialize** can be used to set local variables used as state variables, to read port or module attributes such as time steps or port rates, or to initialize ports with a delay. This member function is executed only once, just before the actual module activation starts (see next section). The example below shows the initialization of an internal state variable *s* and the use of the port member function **get_timestep** and **initialize**. The available port member functions are explained in Section 2.2.2.

```
void initialize()
{
  s = 4.56; ❶

  std::cout << out.name() << ": Time step = " << out.get_timestep() << std::endl; ❷

  out.initialize(1.23); ❸
```

```
}
```

❶ Set local state variable 's' (private data member of type double)
❷ Get time step of output port *out*.
❸ Initialize the first sample of output port *out* with value 1.23.

How to use port initialization inside this member function is explained in Section 2.2.2.

Module activation (processing)

The member function **processing** is the only mandatory function that needs to be overloaded in any TDF module, since it actually defines the discrete-time or continuous-time behavior of the TDF module. This member function is executed at each module activation (see Section 2.3). The example below shows a very simple case, in which the value of an internal data member *val* is written to an output port.

```
void processing()
{
  out.write(val); // writes value to output port out
}
```

Module local time

The member function **get_time** can be used within the **processing** function to obtain the actual, local module time. It returns the time of the first input sample of the current module activation, as a type of class **sca_core::sca_time**. At elaboration and initialization, the actual module time returned by this function is zero (**sc_core::SC_ZERO_TIME**), as the module has not been activated yet. The example below shows how the local module time can be obtained.

```
void processing()
{
  sca_core::sca_time local_time;
  local_time = get_time(); // get actual, local module time
}
```

For multirate TDF models, the local time of the individual TDF modules can differ. Furthermore, there may be time offsets between the local TDF module time and the SystemC kernel time. Therefore, the function **get_time** should be used inside a TDF module, as a replacement for **sc_core::sc_time_stamp**.

Module constructor

The macro **SCA_CTOR** helps to define the standard constructor of a module of class **sca_tdf::sca_module**. It has only one mandatory argument, which is the module name. In cases where parameters need to be passed via the constructor, the user may define a regular constructor with an arbitrary number of parameters.

Member data should be initialized in the initialization list of the constructor, so that all members are properly initialized before the constructor of **my_tdf_module** is called.

```
my_tdf_module( sc_core::sc_module_name nm, double param_ )
: param(param_) {}
```

Constraints on usage

A TDF module is a primitive of the TDF model of computation. Therefore it cannot instantiate submodules. The structural composition of TDF modules is possible by defining classes derived from the regular SystemC class **sc_core::sc_module**, or using the equivalent macro **SC_MODULE**. This is discussed in Section 2.3.3.

The member functions **set_attributes**, **initialize**, **processing**, and **ac_processing** should not be called directly by the user. These member functions are called as part of the execution semantics for time-domain simulation (Section 2.5) or small-signal frequency-domain analyses (Section 5.1.2).

SystemC functions to describe discrete-event behavior such as creating methods and threads, specifying sensitivity, waiting for events, and so on are not allowed to be called in a TDF module. Otherwise, the execution semantics for SystemC discrete-event processing could interfere with the execution of the

TDF modules. This means member functions and macros like **SC_HAS_PROCESS**, **SC_METHOD**, **SC_THREAD**, **wait**, **next_trigger**, **sensitive** should not be used in a TDF module.

As the local time of a TDF module is calculated independently from the time in the discrete-event domain (SystemC kernel time), the function **sc_core::sc_time_stamp** should not be used inside a TDF module. Instead, the member function **get_time** should be used.

In case SystemC signals are needed for processing in a TDF module, specialized converter ports have to be used, as described in the next section.

2.2.2. TDF ports

A TDF port is an object that provides a TDF module with a means to communicate with other connected modules. Due to the nature of the TDF modeling formalism, a TDF port can be either an input port or an output port, but not *inout* (which is available in SystemC). TDF ports can be declared for any data type defined by C++, SystemC, the SystemC AMS extensions, a third-party library, or the user.

There are currently four classes of TDF ports:

- TDF ports of class **sca_tdf::sca_in<T>** (input port) or **sca_tdf::sca_out<T>** (output port).
- TDF *converter ports* of class **sca_tdf::sca_de::sca_in<T>** (input converter port) or **sca_tdf::sca_de::sca_out<T>** (output converter port).

TDF ports are used to connect TDF modules using signals of class **sca_tdf::sca_signal<T>**. TDF converter ports allow TDF modules to interact with discrete-event signals of class **sc_core::sc_signal<T>** or **sc_core::sc_buffer**. This is explained in Section 2.4.

The port template classes allow the use of different data types, e.g., *double*, *int* or *bool*. The data type *double* is often used to represent the amplitude of a continuous-value signal. The example below shows the instantiation of the four available TDF port classes.

```
SCA_TDF_MODULE(my_tdf_module)
{
  sca_tdf::sca_in<double> in;   ❶
  sca_tdf::sca_out<double> out; ❷

  sca_tdf::sca_de::sca_in<bool> inp;  ❸
  sca_tdf::sca_de::sca_out< sc_dt::sc_logic > outp; ❹

  // rest of module not shown
};
```

❶ TDF input port that carries a continuous-value (real) signal.
❷ TDF output port that carries a continuous-value (real) signal.
❸ TDF input converter port from the discrete-event domain, using a boolean signal.
❹ TDF output converter port to the discrete-event domain, using a SystemC logic signal.

Port attributes

A number of attributes can be assigned to TDF ports. They are used to control the evaluation and execution of the TDF cluster, to which the TDF module belongs. TDF port attributes have to be set in the member function **set_attributes** of the TDF module, in which the port is declared (see 2.2.1). The following member functions are available for TDF ports to set or get the attributes:

- The member functions **set_timestep** and **get_timestep** will set and return, respectively, the time step (sampling period) between two consecutive samples.
- The member functions **set_rate** and **get_rate** will set and return, respectively, the number of samples that have to be read or written to the port per module execution. The default rate is 1 (single-rate port).
- The member functions **set_delay** and **get_delay** will set and return, respectively, the number of samples, which are inserted before reading or writing the first time to the port. The default value depends on the default constructor of the data type. In case of C++'s base type like *bool*, *int*, *long*, *float*, and *double*, the

initial value could be undefined. Therefore, it is recommended to initialize the port with an initial value, if a delay has been specified for a port (see the section called "Port initialization").

- Member function **set_timeoffset** and **get_timeoffset** will set or return the actual time of the first sample of the port. This function is only available for converter ports.

The example below shows the use of these member functions:

```
  void set_attributes()
  {
    out.set_timestep(0.01, sc_core::SC_US);   // set time step of port out
    out.set_rate(1);                          // set rate of port out to 1
    out.set_delay(2);                         // set delay of port out to 2 samples
    outp.set_timeoffset(0.2, sc_core::SC_US); // set absolute time of first sample of converter port
  }

  void initialize()
  {
    out.get_rate();        // return the rate of port out
    out.get_delay();       // return the delay of port out
    out.get_timestep();    // return actual timestep of port out
    outp.get_timestep();   // return actual timestep of converter port outp
    outp.get_timeoffset(); // return absolute time of first sample of converter port outp
  }
```

Port initialization

The initial values of TDF ports with a specified delay have to be specified in the member function **initialize** of the corresponding TDF module. The example below shows the initialization of port *out*, which delay has been set to 2 samples.

```
  void initialize() // use initialize method of TDM module to initialize ports
  {
    // initialize port out (which has a delay attribute of 2)
    out.initialize(1.23);    // initialize first  sample with value 1.23 or
    out.initialize(1.23,0);  // initialize first  sample with value 1.23
    out.initialize(4.56,1);  // initialize second sample with value 4.56
  }
```

Port read and write access

Samples can be read from a TDF input port by calling its member function **read** from within the member function **processing** of the corresponding TDF module. In case of a multirate port, the sample index can be passed as an argument to read.

In the case of a single rate TDF input port, reading from this port is done as follows:

```
SCA_TDF_MODULE(my_tdf_sink)
{
  sca_tdf::sca_in<double> in;

  SCA_CTOR(my_tdf_sink) : in("in") {}

  void processing()
  {
    // local variable
    double val; // variable to store value read from port in

    val = in.read(); // reading first sample from the input port
  }
};
```

Consecutive read accesses during the same module activation returns the same value, i.e., the input sample is not consumed by the read access.

In the case of a multirate TDF input port, reading from this port is done as follows:

```
SCA_TDF_MODULE(my_multi_rate_sink)
{
  sca_tdf::sca_in<double> in;

  SCA_CTOR(my_multi_rate_sink) : in("in") {}
```



```
  void set_attributes()
  {
    in.set_rate(2); // 2 samples read per module activation
  }

  void processing()
  {
    // local variable
    double val; // variable to store values read from port in

    val = in.read();  // read first sample
    val = in.read(0); // same method with index for first sample
    val = in.read(1); // same method with index for second sample
  }
};
```

The rate attribute of the input port defines the number of samples available per module activation. In the example above, the port rate of 2 gives access to 2 samples with respective index 0 and 1. As for single rate ports, consecutive read accesses during the same module activation return the same value.

Samples can be written to a TDF output port by passing the sample value as argument to the member function **write** from within the member function **processing** of the corresponding TDF module. In case of a multirate port, the sample index can be passed along with the sample value as an argument to write.

In the case of a single rate TDF output port, writing to this port is done as follows:

```
SCA_TDF_MODULE(my_const_source)
{
  sca_tdf::sca_out<double> out;

  my_const_source( sc_core::sc_module_name, double val_ = 1.0 )
  : out("out"), val( val_ ) {}

  void processing()
  {
    out.write( val ); // writes val as a new sample to the port out
  }

 private:
  double val; // value to be written to the port out
};
```

Consecutive write accesses during the same module evaluation overwrite the sample value, i.e., only the last written output sample is emitted.

In the case of a multirate TDF output port, writing to this port is done as follows:

```
SCA_TDF_MODULE(my_multi_rate_const_source)
{
  sca_tdf::sca_out<double> out;

  my_multi_rate_const_source(sc_core::sc_module_name, double val_ = 1.0 )
  : out("out"), val( val_ ) {}

  void set_attributes()
  {
    out.set_rate(2);  // 2 samples written per module activation
  }

  void processing()
  {
    out.write(val);   // writes val as the first sample to the port out
    out.write(val,0); // writes val as the first sample to the port out by specifying the index 0
    out.write(val,1); // writes val as the second sample to the port out by specifying the index 1
  }

 private:
  double val; // value to be written to the port out
};
```

The rate attribute of the output port defines the number of samples, which can be written to the port per module activation. In the example above, the port rate of 2 gives write access to 2 samples with respective index 0 and 1. As for single rate ports, consecutive write accesses during the same module activation overwrite the previous sample value.

Read and write access to SystemC discrete-event signals is done using so called converter ports of class **sca_tdf::sca_de::sca_in<T>** or **sca_tdf::sca_de::sca_out<T>**. The usage of these converter ports is discussed in Section 2.4.

Port and sample time

The member function **get_time** can only be used after elaboration is finished, i.e., in the TDF module's member functions **initialize** and **processing**, to obtain the actual time of the requested sample at an input or output port. In case no argument is used, it returns the time of the first sample, which has been read from or written to a port. An argument can be passed to this function to specify the sample index, where 0 indicates the first sample.

```
void processing()
{
  sca_core::sca_time t;

  t = out.get_time();  // return time of the first sample of port out
  t = out.get_time(0); // same method, the first sample has index 0

  t = in.get_time(1);  // return time of second sample of port in, with index 1
}
```

Constraints on usage

The TDF port member functions **set_timestep**, **set_delay**, **set_rate**, and **set_timeoffset** for TDF converter ports can only be called in the TDF module member function **set_attributes**, as this information is required for the elaboration phase.

The TDF port member functions **get_timestep**, **get_delay**, **get_rate**, **get_time** and **get_timeoffset** for TDF converter ports can only be called after elaboration is finished, i.e., in the TDF module member function **initialize** or **processing**.

2.2.3. TDF signals

TDF signals are used to connect TDF ports of different primitive TDF modules together. TDF signals carry the samples of a signal, while TDF ports determine the direction of the signals from one TDF module to another. TDF signals are declared using the template class **sca_tdf::sca_signal<T>**. The data type of the signal is passed as a template argument to this class. For example, a continuous-value signal can be represented by using the data type *double*:

```
// signal declarations
sca_tdf::sca_signal<double> sig; // continuous-value signal
```

Unlike SystemC signals, the TDF signals of the AMS extensions do not provide member functions to directly read to or write from the channel. Instead, the member functions read and write are defined for TDF input and TDF output ports, respectively, as already described in Section 2.2.2.

As in SystemC, the constructor initialization of the parent module can be used to assign a user-defined name to a signal:

```
// assign the name "sig" to a TDF signal instance called sig in the constructor initialization list
SC_CTOR(my_module) : sig("sig") {}
```

Section 2.3.3 will describe the structural composition of TDF modules in more detail and will show examples of assigning user-defined names to ports and signals.

2.3. Modeling discrete-time and continuous-time behavior

A TDF module is the basic structural building block for describing discrete-time and continuous-time behavior. It is a class that implements a TDF behavioral description, and may not instantiate other modules. TDF modules act as primitive modules.

2.3.1. Discrete-time modeling

Discrete-time behavior can be defined in the member function **processing**. In this member function, a pure algorithmic or procedural description in C++ can be given, which is executed at each module activation. The module activation is defined by the module time step, which can be either user-specified with the member function **set_timestep** or derived by time step propagation (see Section 2.1.3).

In Figure 2.16, an example is given for a 1 kHz sinusoidal source. By defining a module time step of 0.125ms, the actual output signal will be oversampled with a factor of 8.

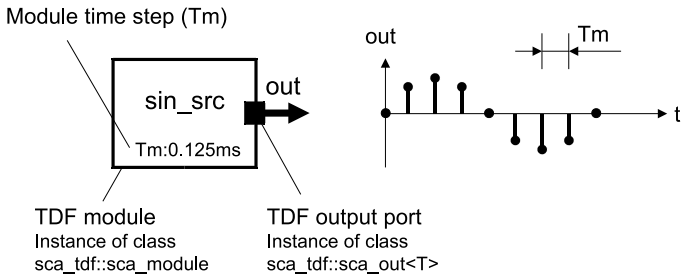


Figure 2.16. TDF primitive module implementing a sinusoidal source

The corresponding C++ source code is given below. The constructor has parameters with default values, which define the amplitude, frequency and sampling period (in this case equal to the module time step) of the sine wave to be generated by the source. The module time step is usually set in the member function **set_attributes**. The sinus function **sin**, which is part of the C++ math library, is used in the member function **processing**. To write the samples to the output port, the port member function **write** is used.

```
SCA_TDF_MODULE(sin_src)
{
  sca_tdf::sca_out<double> out; // output port

  sin_src( sc_core::sc_module_name nm, double ampl_= 1.0, double freq_ = 1.0e3,
           sca_core::sca_time Tm_ = sca_core::sca_time(0.125, sc_core::SC_MS) )
  : out("out"), ampl(ampl_), freq(freq_), Tm(Tm_)
  {}

  void set_attributes()
  {
    set_timestep(Tm);
  }

  void processing()
  {
    double t = get_time().to_seconds(); // actual time
    out.write( ampl * std::sin( 2.0 * M_PI * freq * t ) );
  }

 private:
  double ampl; // amplitude
  double freq; // frequency
  sca_core::sca_time Tm; // module time step
};
```

2.3.2. Continuous-time modeling

A TDF module can be used to embed linear dynamic equations in the form of linear transfer functions in the Laplace domain or state-space equations. Although the TDF model of computation processes the samples at discrete time steps, the equations of these embedded functions will be solved by considering the input samples as continuous-time signals. The result of the embedded linear dynamic equations system, which is also continuous in time and value, is sampled into a signal using a time step which corresponds to the time step of the port, in which the samples are written.

The example below shows the corresponding signal flow when embedding a Laplace transfer function (LTF) in a TDF module. The input signal represents a sampled step function. This discrete-time signal is

interpreted by the LTF function as a continuous-time signal. The filtered, continuous-time signal is written to the output port. During this write operation, the continuous-time signal is being sampled into a discrete-time signal using the output port attributes.

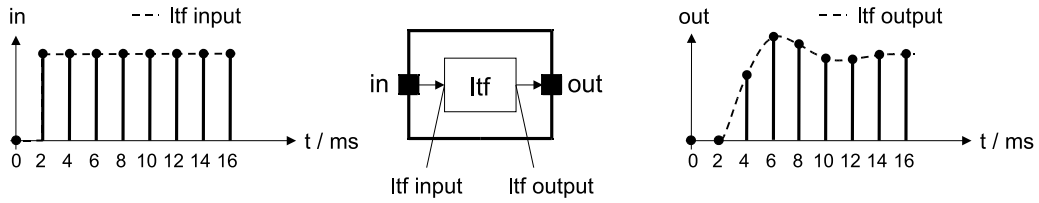


Figure 2.17. TDF primitive module embedding a continuous-time Laplace transfer function (LTF)

Laplace transfer functions

A Laplace transfer function (LTF) can be used in the numerator-denominator or zero-pole form.

The class **sca_tdf::sca_ltf_nd** implements a scaled continuous-time linear transfer function of the Laplace-domain variable *s* in the numerator-denominator form:

$$H(s) = k \cdot \frac{\sum_{i=0}^{M-1} num_i \cdot s^i}{\sum_{i=0}^{N-1} den_i \cdot s^i} \cdot e^{(-s \cdot delay)}$$

where *k* is the constant gain of the transfer function, *M* and *N* are the number of numerator and denominator coefficients, respectively, and num_i and den_i are real-valued coefficients of the numerator and denominator, respectively. The coefficients must be declared as objects of class **sca_util::sca_vector** with data type *double*. The parameter *delay* is the time continuous delay applied to the values available at the input.

The example below shows a first-order low-pass filter using the following Laplace transfer function:

$$H(s) = \frac{H_0}{1 + \frac{1}{2\pi f_c} s}$$

where H_0 is the DC gain and f_c is the filter cut-off frequency in Hz.

The following code implements such a behavior in a TDF module using the class **sca_tdf::sca_ltf_nd**, which instantiates the corresponding equation system. The numerator and denominator coefficients are calculated from the user-specified gain and cut-off frequency.

```
SCA_TDF_MODULE(ltf_nd_filter)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;

  ltf_nd_filter( sc_core::sc_module_name nm, double fc_, double h0_ = 1.0 )
  : in("in"), out("out"), fc(fc_), h0(h0_) {}

  void initialize()
  {
    num(0) = 1.0;
    den(0) = 1.0;
    den(1) = 1.0 /( 2.0 * M_PI * fc );
  }

  void processing()
  {
    out.write( ltf_nd( num, den, in.read(), h0 ) );
  }

 private:
  sca_tdf::sca_ltf_nd ltf_nd;           // Laplace transfer function
```



```
  sca_util::sca_vector<double> num, den; // numerator and denominator coefficients
  double fc; // 3dB cut-off frequency in Hz
  double h0; // DC gain
};
```

The next example shows the same filter, but now implemented as zero-pole description, using the class **sca_tdf::sca_ltf_zp**.

The class **sca_tdf::sca_ltf_zp** implements a scaled continuous-time linear transfer function of the Laplace-domain variable *s* in the zero-pole form:

$$H(s) = k \cdot \frac{\prod_{i=0}^{M-1}(s - zeros_i)}{\prod_{i=0}^{N-1}(s - poles_i)} \cdot e^{(-s \cdot delay)}$$

where *k* is the constant gain of the transfer function, *M* and *N* are the number of zeros and poles, respectively, and $zeros_i$ and $poles_i$ are complex-valued zeros and poles, respectively. If *M* or *N* is zero, the corresponding numerator or denominator term shall be a constant 1. The parameter *delay* is the time continuous delay applied to the values available at the input.

The zeros and poles must be declared as objects of class **sca_util::sca_vector** with a *complex* data type of class **sca_util::sca_complex**.

For a first-order low-pass filter, the zero-pole respresentation becomes:

$$H(s) = \frac{H_0}{1 + \frac{1}{2\pi f_c} s} = \frac{H_0 2\pi f_c}{s + 2\pi f_c}$$

This filter does not require any zeros to be defined. The poles and *k*-value of the filter are calculated from the user-defined DC gain H_0 and cut-off frequency f_c.

```
SCA_TDF_MODULE(ltf_zp_filter)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;

  ltf_zp_filter( sc_core::sc_module_name nm, double fc_, double h0_ = 1.0 )
  : in("in"), out("out"), fc(fc_), h0(h0_) {}

  void initialize()
  {
    // filter requires no zeros to be defined
    poles(0) = sca_util::sca_complex( -2.0 * M_PI * fc, 0.0 );
    k = h0 * 2.0 * M_PI * fc;
  }

  void processing()
  {
    out.write( ltf_zp( zeros, poles, in.read(), k ) );
  }

 private:
  double k; // filter gain
  sca_tdf::sca_ltf_zp ltf_zp; // Laplace transfer function
  sca_util::sca_vector<sca_util::sca_complex > poles, zeros; // poles and zeros as complex values
  double fc; // 3dB cut-off frequency in Hz
  double h0; // DC gain
};
```

The numerator and denominator coefficients or zero-pole values do not need to be static. Their values may change during simulation.

State-space equations

The class **sca_tdf::sca_ss** implements a continuous-time system, which behavior is defined by the following state-space equations:

$$\frac{ds(t)}{dt} = \mathbf{A} \cdot s(t) + \mathbf{B} \cdot x(t - delay)$$
$$y(t) = \mathbf{C} \cdot s(t) + \mathbf{D} \cdot x(t - delay)$$

where $s(t)$ is the state vector, $x(t)$ is the input vector, and $y(t)$ is the output vector. The parameter *delay* is the time continuous delay applied to the values available at the input. **A**, **B**, **C**, and **D** are matrices having the following characteristics:

- **A** is a n-by-n matrix, where n is the number of states.
- **B** is a n-by-m matrix, where m is the number of inputs.
- **C** is a r-by-n matrix, where r is the number of outputs.
- **D** is a r-by-m matrix.

The matrices **A**, **B**, **C**, and **D** must be declared as objects of class **sca_util::sca_matrix** with data type *double*.

The next example shows the same low-pass filter, but now implemented as state-space equation, using the class **sca_tdf::sca_ss**.

```
SCA_TDF_MODULE(statespace_eqn)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;

  statespace_eqn( sc_core::sc_module_name nm, double fc_, double h0_ = 1.0 )
  : in("in"), out("out"), fc(fc_), h0(h0_) {}

  void initialize()
  {
    double r_val = 1e3;
    double c_val = 1.0 / ( 2.0 * M_PI * fc * r_val);

    a(0,0) = -1.0 / ( c_val * r_val );
    b(0,0) = 1.0 / r_val;
    c(0,0) = h0 / c_val;
    d(0,0) = 0.0;
  }

  void processing()
  {
    sca_util::sca_vector<double> x;
    x(0) = in.read();

    sca_util::sca_vector<double> y = state_space1( a, b, c, d, s, x );
    out.write(y(0));
  }

 private:
  sca_tdf::sca_ss state_space1;            // state-space equation
  sca_util::sca_matrix<double> a, b, c, d; // state-space matrices
  sca_util::sca_vector<double> s;          // state vector
  double fc; // 3dB cut-off frequency in Hz
  double h0; // DC gain
};
```

Using the state vector

If a coefficient (thus parameter) in a Laplace transfer function or state-space equation has changed, the corresponding equation system will be reinitialized. A user-defined vector of class **sca_util::sca_vector<double>** can be used to store the state of the equation system. If not specified, an internal state vector is used, which is not accessible to the user. The user-defined state vector is not changed during reinitialization, but only the default internal state is reset to zero. This allows the creation of filters with different parameters, e.g., to realize a switch with different cut-off frequencies, by defining multiple LTF instances using the same state vector. The example below shows how to model such a switch.

```
SCA_TDF_MODULE(ltf_switch)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;
  sca_tdf::sca_de::sca_in<bool> fc_high; // control signal from the discrete-event domain
```



```
  ltf_switch( sc_core::sc_module_name nm, double fc0_, double fc1_, double h0_ = 1.0 )
  : in("in"), out("out"), fc_high("fc_high"), fc0(fc0_), fc1(fc1_), h0(h0_) {}

  void initialize()
  {
    num(0) = 1.0;
    den0(0) = den1(0) = 1.0;
    den0(1) = 1.0 /( 2.0 * M_PI * fc0 );
    den1(1) = 1.0 /( 2.0 * M_PI * fc1 );
  }

  void processing() ❶
  {
    if ( fc_high.read() )
      out.write( ltf1( num, den1, state, in.read(), h0 ) );
    else
      out.write( ltf0( num, den0, state, in.read(), h0 ) );
  }

 private:
  sca_tdf::sca_ltf_nd ltf0, ltf1;
  sca_util::sca_vector<double> num, den0, den1;
  sca_util::sca_vector<double> state; ❷
  double fc0, fc1;
  double h0;
};
```

❶ The user-defined state vector is kept constant during reinitalization of the LTF function.

❷ Declaration of user-defined state vector to store the state of the system during reinitalization of the LTF function.

Using Laplace transfer functions or state-space equations in multirate applications

The Laplace transfer functions or state-space equation examples shown so far use the **read** method of an input port to retrieve a single value, and use the **write** method to write a single value to an output port.

Laplace transfer function or state-space equations can also be embedded in multirate applications, where, for example, the input signal has a higher rate than the output signal, as shown in Figure 2.18. In this example, the TDF module needs to read two input values at each module activation, which then need to be passed to the embedded function.

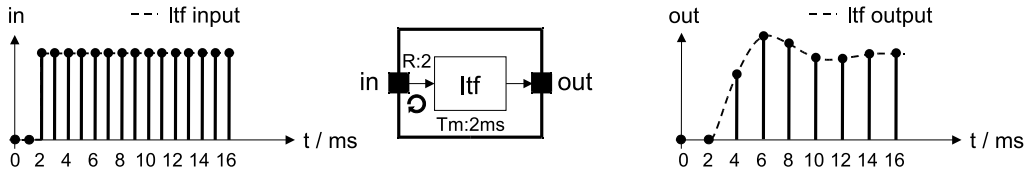


Figure 2.18. Laplace transfer function used for combined filtering and decimation

In order to pass all available samples at the input port directly to the LTF function, not the values, but the *reference* to the port itself is passed as argument to the LTF function, as shown in the example below.

```
SCA_TDF_MODULE(ltf_multirate_filter)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;

  tf_multirate_filter( sc_core::sc_module_name nm, double fc_, double h0_ = 1.0 )
  : in("in"), out("out"), fc(fc_), h0(h0_) {}

  set_attributes()
  {
    in.set_rate(2);
  }

  void initialize()
  {
    num(0) = 1.0;
    den(0) = 1.0;
    den(1) = 1.0 /( 2.0 * M_PI * fc );
```

```
  }

  void processing()
  {
    out.write( filter( num, den, in, h0 ) );  ❶
  }

 private:
  sca_tdf::sca_ltf_nd filter;
  sca_util::sca_vector<double> num, den;
  double fc;
  double h0;
};
```

❶ The argument *in* directly passes the reference to the input port to the LTF function. Note that in the previous cases, the input port member **read** is used, which returns a value of type *double*, which is passed to the LTF function.

In a similar way, TDF modules with embedded Laplace transfer functions or state-space equations can be designed using output ports with a rate higher than 1. Writing multiple samples to an output port is facilitated by the port **write** method, which can access the continuous-time values from a Laplace transfer or state-space function, and write the complete set of output samples to an output port. There is no different language construct needed to make use of this feature.

Special care has to be taken in case the number of output samples is higher than the number of input samples. For example, in a TDF module with an output port rate of 3 and an input port rate of 2, there is 1 sample missing at the first module activation to write the required samples (3) to the output. To resolve this, a *time continuous* delay to the input signal should be specified as additional parameter *delay*, which is one of the function parameters.

2.3.3. Structural composition of TDF modules

The way how TDF modules are instantiated and interconnected to form a TDF cluster does not differ from regular SystemC modules. They can be instantiated as child modules inside a regular SystemC parent module created with the help of the macro **SC_MODULE** or by deriving publicly from **sc_core::sc_module**. This parent module also instantiates all necessary ports to communicate with the outside world and internal signals for the interconnection of the child modules. The parameterization of the instantiated modules as well as the interconnection of the modules should be done in the constructor (e.g., created with the help of the macro **SC_CTOR**) of the parent SystemC module. The instantiation and interconnection of TDF modules on the top-level inside **sc_main** is done in the same way.

Port binding

In order to connect TDF modules in a proper way to other TDF modules and signals, or even with regular SystemC modules and signals, the following specific bindings are possible, as illustrated in Figure 2.19 and Figure 2.20. The port binding rules are compatible and complementary to the SystemC rules.

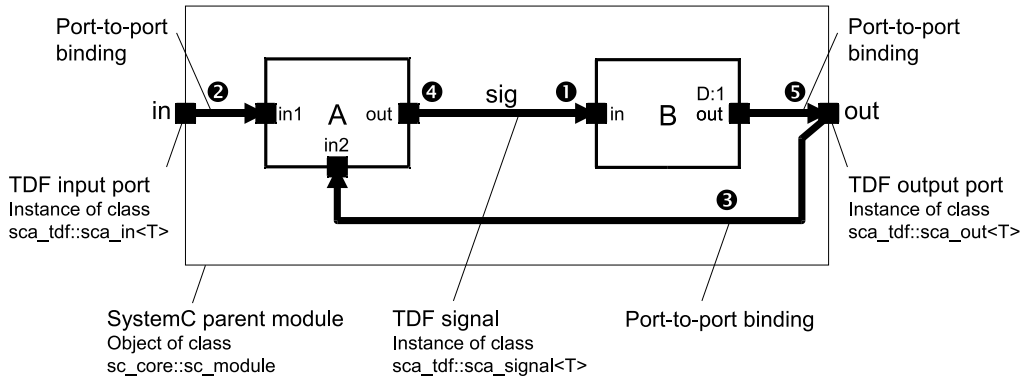


Figure 2.19. Port binding rules for TDF input and output ports

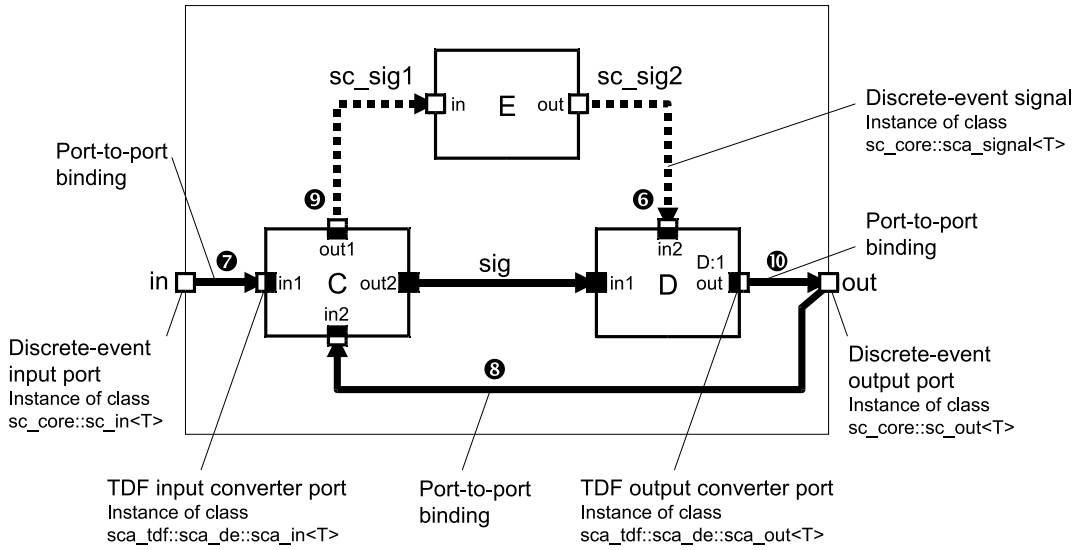


Figure 2.20. Port binding for TDF input and output converter ports

❶ Binding a TDF input port to a TDF signal.

❷ Binding a TDF input port to a TDF input port of the parent module (port-to-port binding).

❸ Binding a TDF input port to a TDF output port of the parent module (port-to-port binding).

❹ Binding a TDF output port to a TDF signal.

❺ Binding a TDF output port to a TDF output port of the parent module (port-to-port binding).

❻ Binding a TDF input converter port to a discrete-event input signal.

❼ Binding a TDF input converter port to a discrete-event input port of the parent module (port-to-port binding).

❽ Binding a TDF input converter port to a discrete-event output port of the parent module (port-to-port binding).

❾ Binding a TDF output converter port to a discrete-event output signal.

❿ Binding a TDF output converter port to a discrete-event output port of the parent module (port-to-port binding).

Furthermore, a TDF input port or TDF output port should be bound to exactly one TDF signal throughout the whole hierarchy. A TDF signal should be bound to exactly one TDF output port of a primitive TDF module, and may be bound to TDF input ports of primitive modules throughout the whole hierarchy.

The example below shows the implementation of the structural composition of Figure 2.19.

```
SC_MODULE(my_structural_module)
{
  sca_tdf::sca_in<double>  in; ❶
  sca_tdf::sca_out<double> out;

  mod_a a; ❷
  mod_b b;

  SC_CTOR(my_structural_module)
  : in("in"), out("out"), a("a"), b("b"), sig("sig") ❸
  {
    a.in1(in); ❹
    a.in2(out);
    a.out(sig);

    b.in(sig);
    b.out(out);
  }

 private:
  sca_tdf::sca_signal<double> sig; ❺
```



```
};
```

❶ The TDF input and output ports declared inside this module of class **sc_core::sc_module** become part of the structural composition.
❷ The child TDF modules are declared within the parent module. The declaration of these child modules should be known prior to the declaration in this context, e.g., by including them via their header files.
❸ The initialization-list in the parent module's constructor propagates the necessary configuration parameters to the TDF ports, TDF signals, and child modules.
❹ Port binding is done inside the constructor.
❺ Internal TDF signals are used to connect the TDF ports and child modules. These signal are declared to be private, as they should not be accessible from outside the module.

The example below shows the implementation of the structural composition of Figure 2.20.

```
SC_MODULE(my_mixed_module)
{
  sc_core::sc_in<double>  in;
  sc_core::sc_out<double> out;

  mod_c c; // TDF primitive module
  mod_d d; // TDF primitive module
  mod_e e; // SystemC module

  SC_CTOR(my_mixed_module)
  : in("in"), out("out"), c("c"), d("d"), e("e"),
    sig("sig"), sc_sig1("sc_sig1"), sc_sig2("sc_sig2")
  {
    c.in1(in);
    c.in2(out);
    c.out1(sc_sig1);
    c.out2(sig);

    d.in1(sig);
    d.in2(sc_sig2);
    d.out(out);

    e.in(sc_sig1);
    e.out(sc_sig2);
  }

 private:
  sca_tdf::sca_signal<double> sig;
  sc_core::sc_signal<bool>    sc_sig1;
  sc_core::sc_signal<bool>    sc_sig2;
};
```

2.3.4. Multirate behavior

To implement multirate behavior in a TDF module, the TDF port member function **set_rate** can be used. Figure 2.21 below shows an example, where the rate of the output port is set to 2. For each module activation, one sample is read from the input port, and two samples are written to the output port. This results in an oversampled signal at the output, with a rate equal to the rate of the output port.

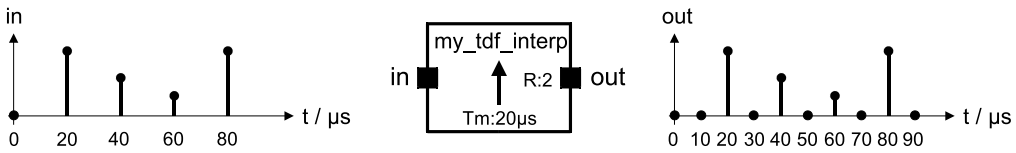


Figure 2.21. Multirate example: 2 times oversampling by inserting zeros

As already discussed in Section 2.1.3, the time step of the TDF input port, output port and module should be consistent. As the module time step is set to 20 µs (Tm:20µs), with an input port rate of 1, the samples at the input port are read each 20 µs. The samples at the output port are written with a time step of 10 µs. This example inserts zeros for the additional samples, but other methods like linear interpolation or sample-and-hold could be implemented as well.

```
SCA_TDF_MODULE(my_tdf_interp) {
```



```
  sca_tdf::sca_in<double> in;
  sca_tdf::sca_out<double> out;

  SCA_CTOR(my_tdf_interp) : in("in"), out("out") {}

  void set_attributes()
  {
    out.set_rate(2);
  }

  void processing()
  {
    out.write( in.read() ); // input sample directly fed to the output
    out.write( 0.0, 1 );    // insert zero as 2nd sample
  }
};
```

Figure 2.22 shows an example, which performs decimation of the input signal, as the rate of the input port is higher than the rate of the output port.

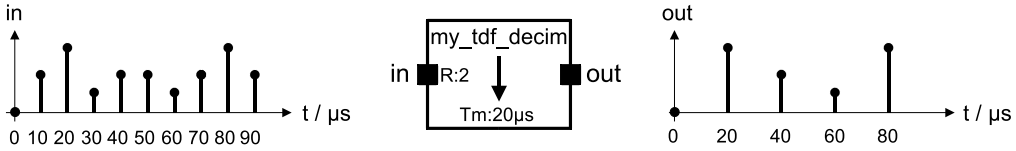


Figure 2.22. Multirate example: Downsampling by a factor of 2

```
SCA_TDF_MODULE(my_tdf_decim)
{
  sca_tdf::sca_in<double> in;
  sca_tdf::sca_out<double> out;

  SCA_CTOR(my_tdf_decim) : in("in"), out("out") {}

  void set_attributes()
  {
    in.set_rate(2);
  }

  void processing()
  {
    out.write( in.read() ); // only write the first sample and neglect the second one
  }
};
```

2.3.5. Introducing delays

Section 2.1.2 explained the cases when delays are essential in a TDF model. The introduction of delays in a TDF cluster will result in inserted samples at the beginning of the sampled TDF signals. The inserted samples are of the same value type as used by the TDF port and signal. As the initial value for a regular C++ data type is undefined, and thus the value of the inserted sample is undefined, it is recommended to initialize these delay samples.

Figure 2.23 shows a basic TDF module, in which a delay of one sample is introduced at the output port.

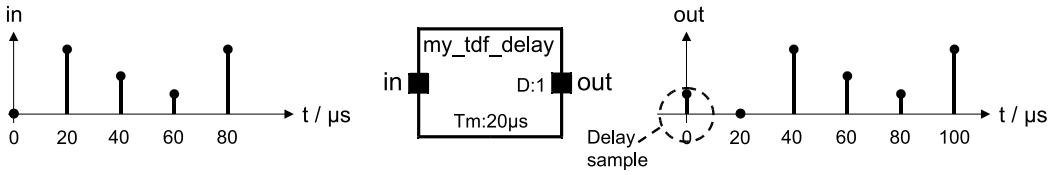


Figure 2.23. TDF module introducing a delay of one sample

The implementation of this delay is given in the next example. It can be seen in the code, that the delay value is also initialized with a default value of 1.1.

```
SCA_TDF_MODULE(my_tdf_delay) {
```



```
  sca_tdf::sca_in<double>   in;
  sca_tdf::sca_out<double> out;

  SCA_CTOR(my_tdf_delay) : in("in"), out("out") {}

  void set_attributes()
  {
    out.set_delay(1);
  }

  void initialize()
  {
    out.initialize(1.1);
  }

  void processing()
  {
    out.write( in.read() ); // directly write the input sample to the output (incl the delay)
  }
};
```

2.4. Interaction between TDF and discrete-event domain

As explained in Section 2.1, the TDF model of computation has its own mechanisms for time annotation, which could result in time differences between the local time of each TDF module and the time in the discrete-event domain (SystemC kernel time). Therefore, special care should be taken in synchronizing TDF signals with the discrete-event domain of SystemC in both directions (i.e., reading from and writing to discrete event signals).

To maintain a high simulation efficiency despite the presence of TDF and discrete-event domain interactions, a loosely-coupled synchronization mechanism is used, which is called *data synchronization*. For TDF modeling this means that discrete events will not influence the activation and execution of TDF modules.

2.4.1. Reading from the discrete-event domain

To read from a channel coming from the discrete-event domain, a TDF input converter port of class **sca_tdf::sca_de::sca_in<T>** has to be used, see Figure 2.24. For convenience, the shorter name **sca_tdf::sc_in<T>** can be used, which class name **sc_in** indicates the interface to the SystemC discrete-event domain. Unlike the regular TDF input ports of class **sca_tdf::sca_in<T>**, the availability of a discrete-event signal at the TDF input converter ports will not activate ("fire") module execution. Instead, the TDF module activation order (schedule) is determined independently at its individual port time step in accordance with the converter port rate and the TDF module time step.

Precondition for correct data synchronization is that the value read from the converter port should be available at the first delta cycle of the corresponding time point in the discrete-event domain. As the TDF cluster runs independently from the discrete-event domain, it could happen that the *previous* discrete-event value is read, indicating that a discrete-event process did not write the value to the channel before the first delta cycle. This would result in a delay in the signal. To overcome this, a small time offset could be introduced using the port member function **set_timeoffset** (see the section called "Port attributes").

The example below shows the use of a TDF module, which reads the values from the discrete-event for further TDF signal processing and writes them to a TDF output port each millisecond.

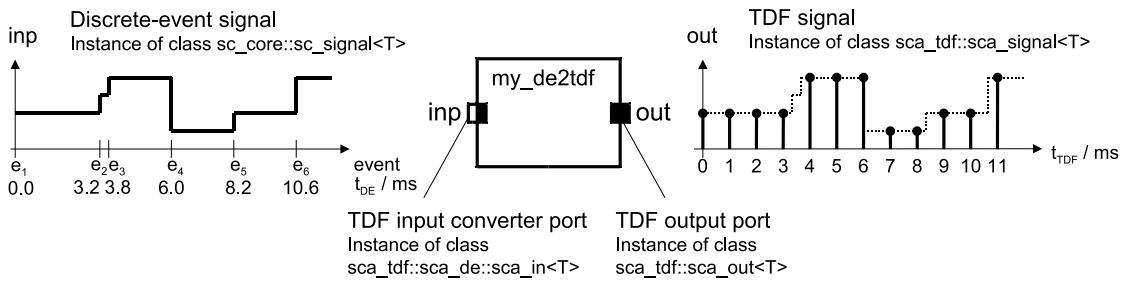


Figure 2.24. TDF module with converter port as input


```
SCA_TDF_MODULE(my_de2tdf)
{
  sca_tdf::sca_de::sca_in<double> inp; // TDF input converter port
  sca_tdf::sca_out<double> out;        // TDF output port

  SCA_CTOR(my_de2tdf) : inp("inp"), out("out") {}

  void set_attributes()
  {
    set_timestep(1.0, sc_core::SC_MS);
  }

  void processing()
  {
    out.write( inp.read() );
  }
};
```

2.4.2. Writing to the discrete-event domain

To write to a channel in the discrete-event domain, a TDF output converter port of class **sca_tdf::sca_de::sca_out<T>** should be used, see Figure 2.25. For convenience, the shorter name **sca_tdf::sc_out<T>** can be used, which class name **sc_out** directly indicates the interface to the SystemC discrete-event domain. The time offset and time step assigned to the output converter port define, at which time point and time interval a value is written to the discrete-event domain.

Precondition for correct data synchronization is that the sample written to the converter port can be written to the associated channel at the first delta cycle of the corresponding discrete-event time point. In case a channel of class **sc_core::sc_signal<T>** is connected to the converter port, there is only a discrete-event generated in case of a signal change, as indicated with the events e_1, e_2, and e_3. In case a channel of class **sc_core::sc_buffer<T>** is connected to the converter port, all samples written to the port will generate an event, which is indicated with the additional samples e_{11}, e_{12}, e_{13}, etc.

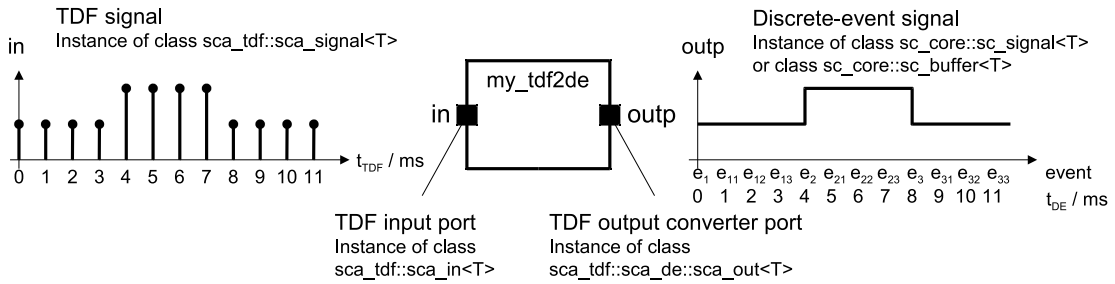


Figure 2.25. TDF module with a converter port as output

The example below shows the implementation of a TDF module, which writes samples to the discrete-event domain.

```
SCA_TDF_MODULE(my_tdf2de)
{
  sca_tdf::sca_in<double> in;            // TDF input port
  sca_tdf::sca_de::sca_out<double> outp; // TDF output converter port

  SCA_CTOR(my_tdf2de) : in("in"), outp("outp") {}

  void set_attributes()
  {
    set_timestep(1.0, sc_core::SC_MS);
  }

  void processing()
  {
    outp.write( in.read() );
  }
```



```
};
```

2.4.3. Using discrete-event control signals

The example below shows a simple digitally controlled gain amplifier, in which the gain is defined by an external control signal from the discrete-event domain. The execution frequency of the member function **processing** is defined by the module time step, which is set to 1 ms. Each time the processing function is called, data from the discrete-event domain is read.

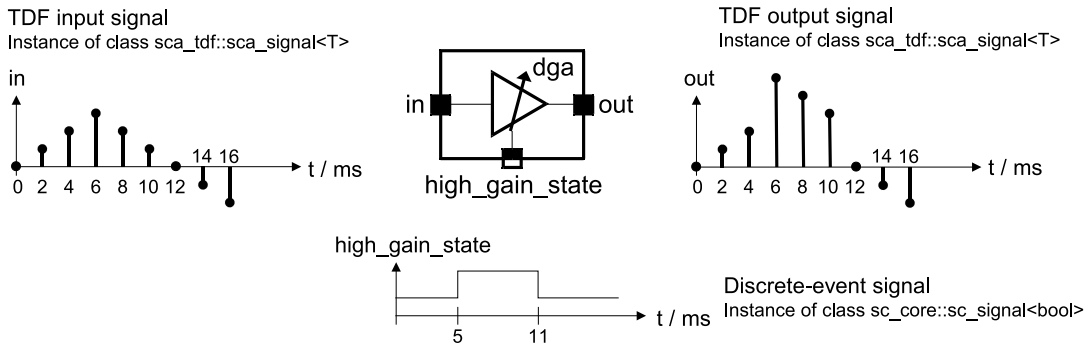


Figure 2.26. TDF module with a converter port used as control input

```
SCA_TDF_MODULE(my_dga)
{
  sca_tdf::sca_in<double>   in; // input port
  sca_tdf::sca_out<double> out; // output port

  // control signal from the discrete-event domain
  sca_tdf::sca_de::sca_in<bool> high_gain_state; // input converter port

  SCA_CTOR(my_dga)
  : in("in"), out("out"), high_gain_state("high_gain_state"),
    high_gain(100.0), low_gain(1.0) {}

  void set_attributes()
  {
    set_timestep(1.0, sc_core::SC_MS);
  }

  void processing()
  {
    double gain = high_gain_state.read() ? high_gain : low_gain;
    out.write( gain * in.read() );
  }

 private:
  double high_gain, low_gain;
};
```

2.5. TDF execution semantics

In addition to the elaboration and simulation phases as defined in SystemC language standard IEEE 1666-2005, specific functionality is implemented for the elaboration and execution of TDF models. The essential TDF module member functions for time-domain simulation are **set_attributes**, **initialize** and **processing**. A user should overload these member functions to implement initialization the initialization and signal processing behavior of his user defined TDF module. It is not allowed to call these member functions directly.

As depicted in Figure 2.27, the elaboration phase includes the following steps:

- *TDF module attribute settings*: Execute the member function **set_attributes** of all TDF modules.
- *TDF time step calculation and propagation*: Propagate and calculate unassigned port and module time steps based on the assigned time steps and port rates. (see Section 2.1.3).

- *TDF cluster computability check*: Define and check the cluster schedule.

The steps for the simulation phase are:

- *TDF module initialization*: Execute the (optional) member function **initialize** of all TDF modules.
- *TDF module activation and processing*: Continuously execute member function **processing** of each TDF module, till all samples have been processed.
- *TDF module post-processing*: Execute the (optional) member function **end_of_simulation** of all TDF modules. Note that this member function is not AMS specific, but is inherited from the SystemC module base class.

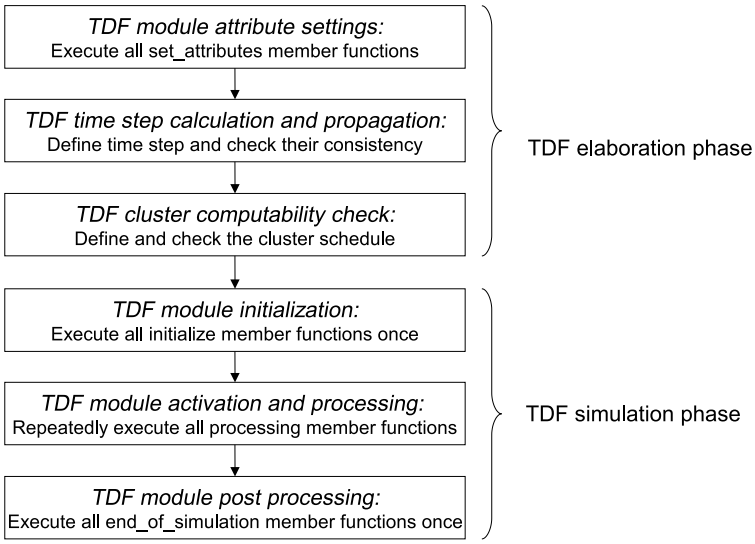


Figure 2.27. TDF elaboration and simulation phases

The elaboration and simulation phases are executed by starting a time-domain simulation using the function **sc_core::sc_start**. This is explained in Section 6.1.1.

2.6. Application examples

This section shows concrete application examples of the Timed Data Flow model of computation and its multirate capabilities. Especially, the interaction of time steps and data rates will play an important role here. The reader is encouraged to reproduce the computations regarding data rates and time steps of the examples in this section in order to grasp the concepts of Timed Data Flow modeling.

2.6.1. BASK modulator

This example considers Binary Amplitude Shift Keying (BASK) modulation, where a sinusoidal carrier is modulated by a binary signal. A BASK modulator consists of the carrier signal source (`sin_src`) and a mixer (`mixer`), which basically multiplies a binary baseband signal (`bit_src`) with segments of the carrier signal. Figure 2.28 shows a structural composition of the BASK modulator. The signals in this figure illustrate the concept of Binary Amplitude Shift Keying.

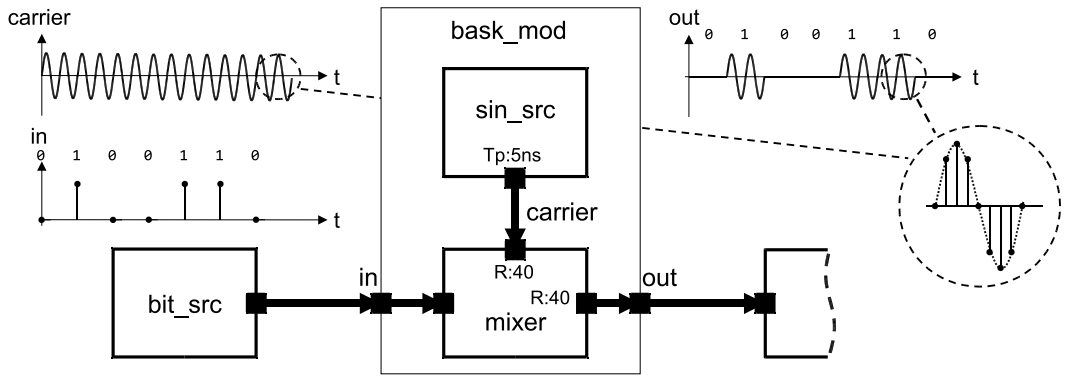


Figure 2.28. BASK modulator

The module `sin_src` is already described in Section 2.3.1. The mixer reads 40 carrier samples per baseband sample. It can be implemented as follows:

```
SCA_TDF_MODULE(mixer)
{
  sca_tdf::sca_in<bool>    in_bin;   // input port baseband signal
  sca_tdf::sca_in<double>  in_wav;   // input port carrier signal
  sca_tdf::sca_out<double> out;      // output port modulated signal

  SCA_CTOR(mixer)
  : in_bin("in_bin"), in_wav("in_wav"), out("out"), rate(40) {}  // use a carrier data rate of 40

  void set_attributes()
  {
    in_wav.set_rate(rate);
    out.set_rate(rate);
  }

  void processing()
  {
    for(unsigned long i = 0; i < rate; i++)
    {
      if ( in_bin.read() )
        out.write( in_wav.read(i), i );
      else
        out.write( 0.0, i );
    }
  }

 private:
  unsigned long rate;
};
```

This is obviously more sensible than up-sampling the binary signal first to a data rate of 40 such that both the carrier signal and the base band signal fit to a mixer with both input ports set to a data rate of 1. The next code snippet shows how the two modules can be combined to form a BASK modulator module. Note that a regular **SC_MODULE** is used in this case, in which the two TDF primitive modules are instantiated.

```
SC_MODULE(bask_mod)
{
  sca_tdf::sca_in<bool>    in;
  sca_tdf::sca_out<double> out;

  sin_src sine;
  mixer   mix;

  SC_CTOR(bask_mod)
  : in("in"), out("out"),
    sine("sine", 1.0, 1.0e7, sca_core::sca_time( 5.0, sc_core::SC_NS ) ),
    mix("mix")
  {
    sine.out(carrier);
    mix.in_wav(carrier);
    mix.in_bin(in);
    mix.out(out);
  }

 private:
```



```
  sca_tdf::sca_signal<double> carrier;
};
```

Note that the carrier frequency of 10 MHz is set by passing a parameter to the module `sin_src`, while the baseband frequency is determined indirectly by the data rate of the module `mixer`, and the time step set at the output of module `sin_src`. The port `in_wav` of the module `mixer` has the same time step as the output of module `sin_src` (namely 5 ns), but a data rate of 40. Therefore, the port `in_bin` of the module `mixer`, which has a data rate of 1, gets a time step of 200 ns. This results in a baseband frequency of 5 MHz, which is exactly the situation depicted in Figure 2.28.

For the sake of completeness, the code of the binary baseband source, which produces a random binary signal is given below.

```
SCA_TDF_MODULE(bit_src)
{
  sca_tdf::sca_out<bool> out; // output port

  SCA_CTOR(bit_src) : out("out") {}

  void processing()
  {
    out.write( (bool)(std::rand()%2) );
  }
};
```

2.6.2. BASK demodulator

The demodulation of a BASK modulated signal is done by first using a rectifier (which takes the absolute value of the signal), followed by a low-pass filter, which can be implemented as described in Section 2.3.2 with the module `ltf_nd_filter`. The rectifier can be implemented as follows:

```
SCA_TDF_MODULE(rectifier)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;

  SCA_CTOR(rectifier) : in("in"), out("out") {}

  void processing()
  {
    out.write( std::abs(in.read()) );
  }
};
```

The output signal of the low-pass filter is a signal of type *double*, which contains 40 samples per 200 ns, and needs to get sampled down to 1 sample per 200 ns (see Figure 2.29).

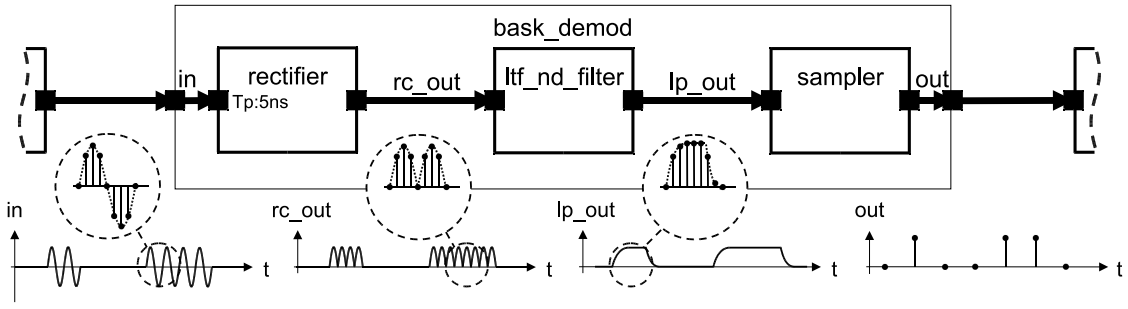


Figure 2.29. BASK demodulator

The next listing shows an implementation of the sampler. It has an input data rate of 40. Therefore, it reads exactly the number of samples, which are associated to one specific bit in the baseband signal. It only uses one sample at a fixed sampling position within the second half of the sample stream read per module execution. The idea behind this is that the output of the low-pass filter can be expected to be settled by that time. This sample is compared with a threshold value: If it is larger, the output of the sampler is *true*, and *false* otherwise. This effectively models a 1-bit A/D converter, which samples its input every 200 ns.


```
SCA_TDF_MODULE(sampler)
{
  sca_tdf::sca_in<double> in;  // input port
  sca_tdf::sca_out<bool>  out; // output port

  SCA_CTOR(sampler) : in("in"), out("out"), rate(40), threshold(0.2) {}

  void set_attributes()
  {
    in.set_rate(rate);
    sample_pos = (unsigned long)std::ceil( 2.0 * (double)rate/3.0 );
  }

  void processing()
  {
    if( in.read(sample_pos) > threshold )
      out.write(true);
    else
      out.write(false);
  }

 private:
  unsigned long rate;
  double threshold;
  unsigned long sample_pos;
};
```

Note that the above code bears a certain causal looseness, which can occur if the rate of the input port is larger than 1: The value of the output sample is computed based on an input sample, which has a time stamp *larger* than the output token. Therefore, regarding the simulation time of the TDF model of computation, effect precedes cause. This irregularity can easily be resolved by introducing a delay, for example with a **set_delay**(1) at the output port. However, this is not really necessary since serious problems (i.e. paradoxes) could occur only if a produced output value would be fed into a feedback loop. But in this case, a delay has to be introduced anyway (see Section 2.1.2), which resolves the problem automatically.

The next listing shows how the three modules are combined for the overall BASK demodulator module. Note that no time step is explicitly set here, since we expect it to be set in the part of the model which provides the modulated signal.

```
SC_MODULE(bask_demod)
{
  sca_tdf::sca_in<double> in;
  sca_tdf::sca_out<bool>  out;

  rectifier     rc;
  ltf_nd_filter lp;
  sampler       sp;

  SC_CTOR(bask_demod)
  : in("in"), out("out"), rc("rc"), lp("lp", 3.3e6), sp("sp"), rc_out("rc_out"), lp_out("lp_out")
  {
    rc.in(in);
    rc.out(rc_out);

    lp.in(rc_out);
    lp.out(lp_out);

    sp.in(lp_out);
    sp.out(out);
  }

 private:
  sca_tdf::sca_signal<double> rc_out, lp_out;
};
```

2.6.3. TDF simulation of the BASK example

The implementation of the complete BASK application is done in the **sc_main** program. Within the program body, the bit source module `bit_src`, BASK modulator module `bask_mod` and BASK demodulator module `bask_demod` are instantiated. These TDF modules are interconnected using TDF signals.

```
int sc_main(int argc, char* argv[])
{
  sc_core::sc_set_time_resolution(1.0, sc_core::SC_FS);
```



```
  sca_tdf::sca_signal<bool>  in_bits, out_bits;
  sca_tdf::sca_signal<double> wave;

  bit_src bs("bs");          // random bit source
    bs.out(in_bits);

  bask_mod mod("mod");       // modulator
    mod.in(in_bits);
    mod.out(wave);

  bask_demod demod("demod"); // demodulator
    demod.in(wave);
    demod.out(out_bits);

  sca_util::sca_trace_file* atf = sca_util::sca_create_vcd_trace_file( "tr.vcd" );

  sca_util::sca_trace( atf, in_bits, "in_bits" );
  sca_util::sca_trace( atf, wave, "wave" );
  sca_util::sca_trace( atf, out_bits, "out_bits" );

  sc_core::sc_start(1, sc_core::SC_US);

  sca_util::sca_close_vcd_trace_file( atf );

  return 0;
}
```

More information on the simulation control and tracing capabilities can be found in Chapter 6.

2.6.4. Interfacing the BASK example with SystemC

As shown by Figure 2.28, the components instantiated in the BASK example are all TDF modules that belong to the same TDF cluster. In particular, the random binary signal at the data input of the mixer is generated by the pure TDF module `bit_src`.

In practice, this binary signal is more likely to be produced by a digital component that follows the discrete-event domain rules, resulting in a true heterogeneous system composed of two digital parts (the random data generator and the data drain) and one AMS TDF part (the BASK modulator and demodulator). Figure 2.30 shows the major modification induced by this design: the data input of the BASK modulator (resp. the data output of the BASK demodulator) should now be a SystemC **sc_core::sc_in<T>** port (resp. **sc_core::sc_out<T>** port) carrying *bool* values. From the TDF perspective, a converter port is thus required to read from the channel (resp. to write to the channel) corresponding to the discrete-event domain port. Such ports are indicated by the symbol ◧ in this Figure.

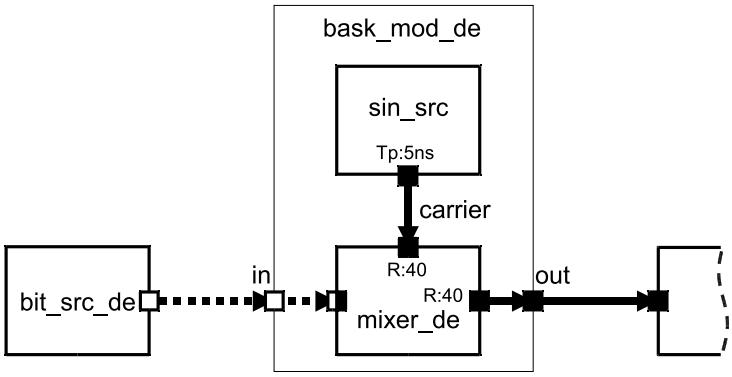


Figure 2.30. BASK modulator, mixing discrete-event and TDF domain

The following code is pure SystemC. Thanks to the infinite loop in a SystemC **SC_THREAD** construct, this new version of the bit source, now called `bit_src_de`, generates a new random bool value on its output port `out` every 200 ns.

```
SC_MODULE(bit_src_de)
{
  sc_core::sc_out<bool> out;

  SC_CTOR(bit_src_de): out("out")
```



```
  {
    SC_THREAD(bit_gen_thread);
  }

  void bit_gen_thread()
  {
    while(true)
    {
      bool var = (bool)(std::rand()%2);
      out.write(var);
      sc_core::wait( 200, sc_core::SC_NS );
    }
  }
};
```

The TDF mixer module has now a digital input `in_bin` connected to the output of the `bit_src_de` SystemC module. The mixer source code does not differ too much from the previous one, the major modification being the introduction of the discrete-event converter port:

```
SCA_TDF_MODULE(mixer_de)
{
  sca_tdf::sca_de::sca_in<bool> in_bin; // TDF converter input port from discrete-event domain
  sca_tdf::sca_in<double>       in_wav;
  sca_tdf::sca_out<double>      out;

  SCA_CTOR(mixer_de)
  : in_bin("in_bin"), in_wav("in_wav"), out("out"), rate(40) {}

  void set_attributes()
  {
    in_wav.set_rate(rate);
    out.set_rate(rate);
  }

  void processing()
  {
    for(unsigned long i = 0; i < rate; i++)
    {
      if(in_bin.read())
        out.write( in_wav.read(i), i );
      else
        out.write( 0.0, i );
    }
  }

 private:
  unsigned long rate;
};
```

Accordingly, the source code for the BASK modulator, shown below, details the slight change needed: the data input is now a discrete-event input port:

```
SC_MODULE(bask_mod_de)
{
  sc_core::sc_in<bool>    in; // data input is now digital
  sca_tdf::sca_out<double> out;

  sin_src  sine;
  mixer_de mix;  // use mixer with discrete-event input

  SC_CTOR(bask_mod_de)
  : in("in"), out("out"),
    sine("sine", 1.0, 1.0e7, sca_core::sca_time( 5.0, sc_core::SC_NS ) ),
    mix("mix"), carrier("carrier")
  {
    sine.out(carrier);
    mix.in_wav(carrier);
    mix.in_bin(in);
    mix.out(out);
  }

 private:
  sca_tdf::sca_signal<double> carrier;
};
```

For completeness, the source code for the BASK sampler in the demodulator is given below. The data output `out` is now a converter output port. The corresponding port in the demodulator which instantiates the sampler is declared as a traditional SystemC output port.


```
SCA_TDF_MODULE(sampler_de)
{
  sca_tdf::sca_in<double> in;  // input port
  sca_tdf::sca_de::sca_out<bool>  out; // TDF converter output port to discrete-event domain

  SCA_CTOR(sampler_de) : in("in"), out("out"), rate(40), threshold(0.2) {}

  void set_attributes()
  {
    in.set_rate(rate);
    sample_pos = (unsigned long)std::ceil( 2.0 * (double)rate/3.0 );
  }

  void processing()
  {
    if( in.read(sample_pos) > threshold )
      out.write(true);
    else
      out.write(false);
  }

 private:
  unsigned long rate;
  double threshold;
  unsigned long sample_pos;
};
```

This page is intentionally left blank.

3. Linear Signal Flow modeling

3.1. Modeling fundamentals

The Linear Signal Flow model of computation allows the modeling of AMS behavior defined as relations between variables of a set of linear algebraic equations. LSF is a continuous-time modeling style using directed real-valued signals, resulting in a non-conservative system description. There is no dependency between flow *and* potential quantities; instead only one real-value quantity is used to represent each signal.

Signal flow models can be described in a *block diagram* notation. The elementary parts or functions are represented by blocks. Signals are used to interconnect these blocks. The resulting relations between the blocks define equivalent mathematical equations. Figure 3.1 shows an example of such a signal flow block diagram, composed of four *LSF modules*, which are interconnected using *LSF signals*. Note that the addition "operator", although having a different graphical representation, is also an LSF module. An *LSF model* is composed of a set of connected LSF modules, which will form together an *LSF equation system* or *LSF cluster*. The resulting LSF model has input and output *LSF ports* to connect it with other modules.

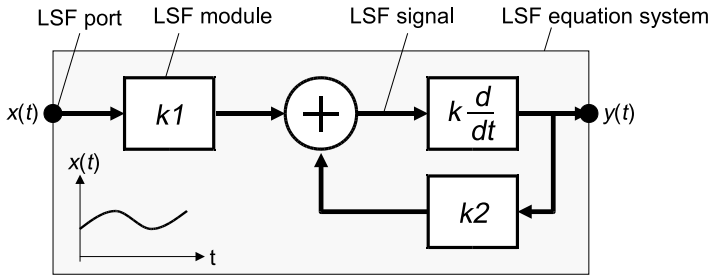


Figure 3.1. Example of a basic LSF model composed of 4 LSF modules

3.1.1. Setup of the LSF equation system

The SystemC AMS extensions offer a finite set of predefined LSF primitive modules implementing functions such as addition, multiplication, integration, etc. Unlike the TDF modeling style, LSF models can only be composed from these primitives. The AMS extensions do not offer the possibility to implement user-defined LSF primitives. Instead, the mathematical equations describing the intended functionality should be created by composing the predefined set of LSF primitive modules. Figure 3.2 shows some basic examples of LSF primitives and their corresponding mathematical equations.

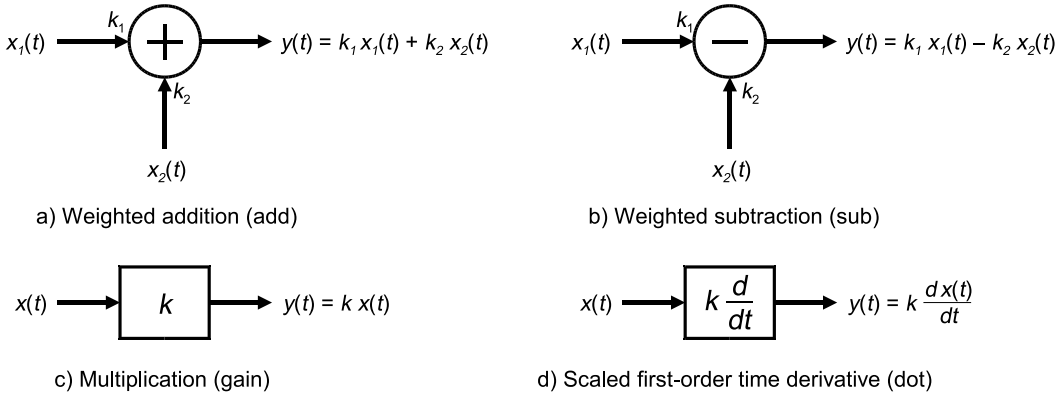


Figure 3.2. Examples of some basic LSF primitives and their corresponding mathematical equations

When creating an LSF model (block diagram), the mathematical equations for each block and their interconnection will be used to compose the overall equation system. For example, the LSF model presented

in Figure 3.1 will result in the following equation system based on the contributed equations of each primitive as shown in Figure 3.2:

$$y(t) = k_1 \cdot \frac{dx(t)}{dt} + k_2 \cdot \frac{dy(t)}{dt}$$

Note that the scale coefficients of the *addition* and the *first-order time derivative* block are set to 1. Instead, additional multiplication blocks *k1* and *k2* are used for this example.

3.1.2. Time step assignment and propagation

Similar as for a TDF module, a time step can be assigned to an LSF module directly or can be assigned automatically using the propagation mechanism of the time step within an LSF cluster. In case an LSF model is connected to a TDF model, the time step from the connected TDF port(s) is propagated to the LSF model. Consistency between locally defined LSF module time step and propagated time step is essential. Otherwise, the time points for the solution of the LSF equation system or communication with the connected TDF model cannot be defined properly (see also Section 2.1.3). The time step should be defined at least at one location in the entire system.

During simulation, the LSF equation system is solved numerically with appropriate time steps, which could be smaller than the assigned time step. The solver will at least provide results at the time points calculated from the assigned time steps.

3.2. Language constructs

3.2.1. LSF modules

A Linear Signal Flow module is a predefined primitive module to represent a particular function or mathematical relation, which will become part of an overall equation system. The available predefined LSF primitive modules are listed in Table 3.1 below. Appendix A gives the details for each LSF module.

LSF module name	Description
sca_lsf::sca_add	Weighted addition of two LSF signals.
sca_lsf::sca_sub	Weighted subtraction of two LSF signals.
sca_lsf::sca_gain	Multiplication of an LSF signal by a constant gain.
sca_lsf::sca_dot	Scaled first-order time derivative of an LSF signal.
sca_lsf::sca_integ	Scaled time-domain integration of an LSF signal.
sca_lsf::sca_delay	Scaled time-delayed version of an LSF signal.
sca_lsf::sca_source	LSF source.
sca_lsf::sca_ltf_nd	Scaled Laplace transfer function in the time-domain in the numerator-denominator form.
sca_lsf::sca_ltf_zp	Scaled Laplace transfer function in the time-domain in the zero-pole form.
sca_lsf::sca_ss	Single-input single-output state-space equation.
sca_lsf::sca_tdf::sca_gain, **sca_lsf::sca_tdf_gain**	Scaled multiplication of a TDF input signal with an LSF input signal.
sca_lsf::sca_tdf::sca_source, **sca_lsf::sca_tdf_source**	Scaled conversion of a TDF input signal to an LSF output signal.
sca_lsf::sca_tdf::sca_sink, **sca_lsf::sca_tdf_sink**	Scaled conversion from an LSF input signal to a TDF output signal.
sca_lsf::sca_tdf::sca_mux, **sca_lsf::sca_tdf_mux**	Selection of one of two LSF input signals by a TDF control signal (multiplexer).

LSF module name	Description
sca_lsf::sca_tdf::sca_demux, **sca_lsf::sca_tdf_demux**	Routing of an LSF input signal to either one of two LSF output signals controlled by a TDF signal (demultiplexer).
sca_lsf::sca_de::sca_gain, **sca_lsf::sca_de_gain**	Scaled multiplication of a discrete-event input signal by an LSF input signal.
sca_lsf::sca_de::sca_source, **sca_lsf::sca_de_source**	Scaled conversion of a discrete-event input signal to an LSF output signal.
sca_lsf::sca_de::sca_sink, **sca_lsf::sca_de_sink**	Scaled conversion from an LSF input signal to a discrete-event output signal.
sca_lsf::sca_de::sca_mux, **sca_lsf::sca_de_mux**	Selection of one of two LSF input signals by a discrete-event control signal (multiplexer).
sca_lsf::sca_de::sca_demux, **sca_lsf::sca_de_demux**	Routing of an LSF input signal to either one of two LSF output signals controlled by a discrete-event signal (demultiplexer).

Table 3.1. LSF primitive modules

Module time step

In order to solve the LSF equation system, a time step has to be associated to the set of connected LSF modules as part of the elaboration phase. This can be done with the LSF module member function **set_timestep**. Alternatively, the LSF model can rely on the time step propagation mechanism, which passes the time step from module to module via its ports across the TDF, LSF, and ELN models of computation. So in cases where an LSF model is connected to a TDF model, the time step from the connected port, if available, is propagated to the LSF model. In case propagated time steps and user-defined time steps are used, consistency between these time steps is compulsory, similar as described in Section 2.1.3.

The module time step can be assigned by calling the member function **set_timestep** of the instantiated object within the constructor of the parent module, and passing a *double* value and the time unit or an object of type **sca_core::sca_time**, as shown in the following example:

```
SC_MODULE(my_lsf_source)
{
  // port declaration
  sca_lsf::sca_out y;

  // child module declaration
  sca_lsf::sca_source src;

  SC_CTOR(my_lsf_source)
  : y("y"),
    src("src", 0.0, 0.0, 1.0e-3, 1.0e3)    // 1 kHz sinusoidal source with an amplitude of 1e-3
  {
    src.set_timestep(0.5, sc_core::SC_MS); // set module timestep of source to 0.5 ms
    src.y(y);
  }
};
```

3.2.2. LSF ports

An LSF port is an object that can be used to connect several LSF models together using LSF signals which are bind to this port. Due to the nature of the LSF modeling formalism, an LSF port can be either an input port or an output port, but not *inout*. LSF ports are used to connect LSF modules using signals of class **sca_lsf::sca_signal**. As LSF ports are always hierarchical ports inside a parent module, they can be used to connect to the LSF child modules directly, following the *port-to-port* binding rule (see Section 3.3.1). LSF ports have a predefined data type, also called *signal flow nature*, which prevents the usage of user-defined data types.

There are currently two classes of LSF ports:

- LSF input ports of class **sca_lsf::sca_in**.
- LSF output ports of class **sca_lsf::sca_out**.

The example below shows how LSF ports are used within an LSF structural model.

```
SC_MODULE(my_lsf_model)
{
  // port declarations
  sca_lsf::sca_in  x; ❶
  sca_lsf::sca_out y; ❷

  SC_CTOR(my_lsf_model) : x("x"), y("y")  ❸
  {
    // model implementation here
  }
};
```

❶ LSF input port that carries a continuous-time and continuous-value signal $x(t)$.
❷ LSF output port that carries a continuous-time and continuous-value signal $y(t)$.
❸ Using the constructor initialization-list to assign the names "x" and "y" to the input and output ports, respectively.

There are no converter ports available for LSF. Instead, specialized converter modules are provided to connect to the TDF or discrete-event domain. This is explained in Section 3.4. Unlike TDF ports, the LSF ports do not provide member functions to directly read to or write from the channel.

3.2.3. LSF signals

LSF signals are used to connect LSF primitive modules together. LSF signals carry the continuous-time and continuous-value of a signal, while LSF ports determine the direction of the signals from one LSF module to another. Similar as for LSF ports, LSF signals use an internal data structure to hold the continuous-time / continuous-value signal. Therefore, the LSF signals are not defined as a template class and should be used according to the example below:

```
  // signal declaration
  sca_lsf::sca_signal sig; // LSF signal
```

As in SystemC, the constructor initialization-list of the parent module can be used to assign a user-defined name to a signal:

```
  // assign the names of LSF signal instance in the constructor initialization-list
  SC_CTOR(my_module) : sig("sig") {}
```

Section 3.3 will describe the creation of structural LSF models and will show examples of assigning user-defined names to ports and signals.

3.3. Modeling continuous-time behavior

LSF models can be used to implement linear dynamic, continuous-time behavior. LSF models can only be composed using LSF primitive modules. Therefore an LSF model is always a structural model.

3.3.1. Structural composition of LSF modules

LSF modules should be instantiated as child modules inside a regular SystemC parent module created with the help of the macro **SC_MODULE** or by deriving publicly from **sc_core::sc_module**. This parent module also instantiates all necessary ports to communicate with the outside world and internal signals for the interconnection of the child modules. The parameterization of the instantiated modules as well as the interconnection of the modules should be done in the constructor (e.g., created with the help of the macro **SC_CTOR**) of the parent SystemC module.

Port binding

In order to connect LSF modules in a proper way to other LSF modules and signals, the following specific bindings are possible, illustrated in Figure 3.3. The port binding rules are compatible and complementary to the SystemC and TDF rules (see also Section 2.3.3).

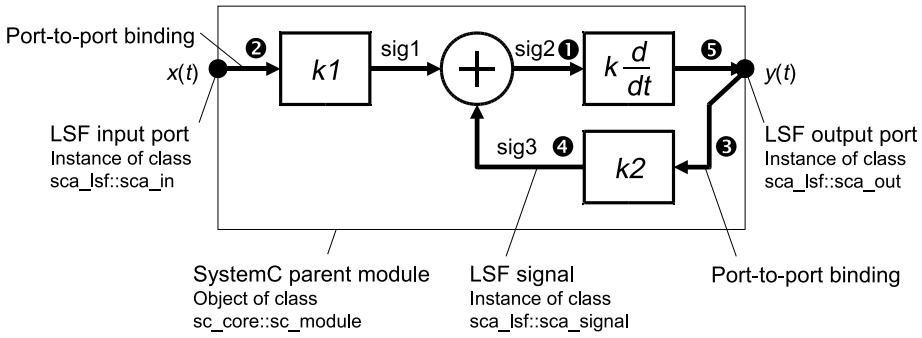


Figure 3.3. Port binding rules for LSF input and output ports

❶ Binding an LSF input port to an LSF signal.

❷ Binding an LSF input port to an LSF input port of the parent module (port-to-port binding).

❸ Binding an LSF input port to an LSF output port of the parent module (port-to-port binding).

❹ Binding an LSF output port to an LSF signal.

❺ Binding an LSF output port to an LSF output port of the parent module (port-to-port binding).

Furthermore, each LSF signal should be bound to exactly one LSF output port of an LSF primitive module, and may be bound to any number of LSF input ports of LSF primitive modules throughout the whole hierarchy.

For LSF primitive modules, which have ports connected to TDF or discrete-event signals or ports, should follow the port binding rules of the corresponding models of computation.

The example below shows the implementation of the structural composition of Figure 3.3.

```
SC_MODULE(my_structural_lsf_model)
{
  sca_lsf::sca_in  x; ❶
  sca_lsf::sca_out y;

  sca_lsf::sca_gain gain1, gain2; ❷
  sca_lsf::sca_dot dot1;
  sca_lsf::sca_add add1;

  my_structural_lsf_model( sc_core::sc_module_name, double k1, double k2 )
  : x("x"), y("y"), gain1("gain1", k1), gain2("gain2", k2), dot1("dot1"), add1("add1"), ❸
    sig1("sig1"), sig2("sig2"), sig3("sig3")
  {
    gain1.x(x);  ❹
    gain1.y(sig1);
    gain1.set_timestep(1,sc_core::SC_MS); ❺

    add1.x1(sig1);
    add1.x2(sig3);
    add1.y(sig2);

    dot1.x(sig2);
    dot1.y(y);

    gain2.x(y);
    gain2.y(sig3);
  }

 private:
  sca_lsf::sca_signal sig1, sig2, sig3; ❻
};
```

❶ The LSF input and output ports declared inside this module of class **sc_core::sc_module** become part of the structural composition.

❷ The LSF primitive modules are declared within the parent module as child modules.

❸ The initialization-list in the parent module's constructor propagates the necessary configuration parameters to the LSF ports, LSF signals, and child modules.
❹ Port binding is done inside the constructor of the parent module.
❺ The time step for LSF primitive modules is done inside the constructor of the parent module. An LSF module could also get its time step via propagation of the time step of its connected modules.
❻ Internal LSF signals are used to connect the LSF ports and child modules. These signals are declared to be private, as they should not be accessible from outside the module.

3.3.2. Continuous-time modeling

The example below shows a first-order low-pass filter, based on the same Laplace transfer function as described in Section 2.3.2:

$$H(s) = \frac{H_0}{1 + \frac{1}{2\pi f_c} s}$$

where H_0 is the DC gain and f_c is the filter cut-off frequency in Hz. The Laplace transfer function can be rewritten for an LSF implementation into:

$$y(t) = H_0 x(t) - \frac{1}{2\pi f_c} \frac{dy(t)}{dt}$$

The corresponding block diagram notation and code implementation is given below, where the scaling coefficients of the LSF primitive modules are used to implement the DC gain H_0 and the filter cut-off frequency f_c :

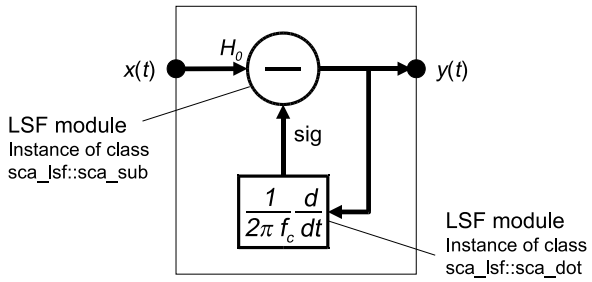


Figure 3.4. Example of an LSF model implementing a first-order low-pass filter

```
SC_MODULE(my_lsf_filter)
{
  sca_lsf::sca_in  x;
  sca_lsf::sca_out y;

  sca_lsf::sca_sub sub1;
  sca_lsf::sca_dot dot1;

  my_lsf_filter( sc_core::sc_module_name, double h0 = 1.0, double fc = 1.0e3 )
  : x("x"), y("y"), sub1("sub1", h0), dot1("dot1", 1.0/(2.0*M_PI*fc) ), sig("sig")
  {
    sub1.x1(x);
    sub1.x2(sig);
    sub1.y(y);

    dot1.x(y);
    dot1.y(sig);
  }

 private:
  sca_lsf::sca_signal sig;
};
```

The gain coefficient *h0* for the input signal is passed via the constructor to the instance **sub1** and the frequency *fc* is passed via the constructor to the instance **dot1**.

3.4. Interaction between LSF and discrete-event or TDF models

The LSF model of computation will setup and solve an equation system to simulate the modeled continuous-time behavior, based on the basic set of LSF primitive modules described in Section 3.2.1. Any "external" input value, e.g., from a discrete-event signal or TDF sample, needs to be contributed to the equation system via one of these LSF primitive modules. Therefore, specialized LSF primitive modules with ports to the discrete-event domain and TDF models of computation are available, which are called *converter modules*. Main purpose of these modules is to establish an interface to convert and transfer data from one model of computation to the other.

3.4.1. Reading from and writing to discrete-event models

In order to connect LSF models with discrete-event models, the LSF converter modules with an internal port of class **sc_core::sc_in** or **sc_core::sc_out** should be used.

Figure 3.5 shows the LSF primitive module **sca_lsf::sca_de::sca_source** reading from a discrete-event signal and writing to an LSF signal. In this example a module time step of 1 ms is assigned to the LSF converter module. The LSF model continuously reads values from the input at the time points, which are calculated from the assigned time steps. The input value is assumed constant until the next value is read. The input values are interpreted to form a continuous-time signal, which is made available at the output of the converter module (read input samples shown as a dotted signal).

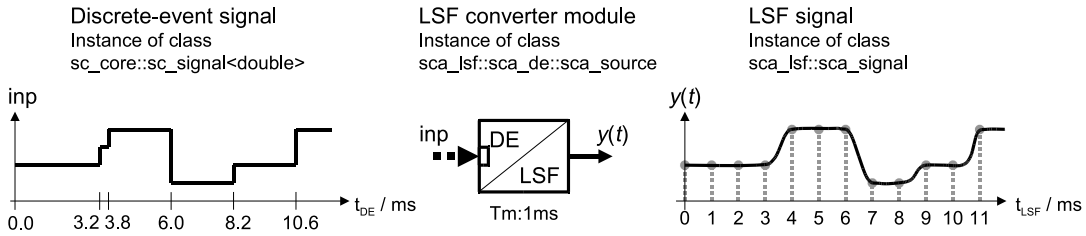


Figure 3.5. LSF converter module reading from a discrete-event input signal and writing to an LSF output signal

Figure 3.6 shows the LSF primitive module **sca_lsf::sca_de::sca_sink**, which reads an LSF signal and writes the equivalent value to the discrete-event signal. The values at the output port are written at the time points, which are calculated from the assigned module time step of 1 ms.

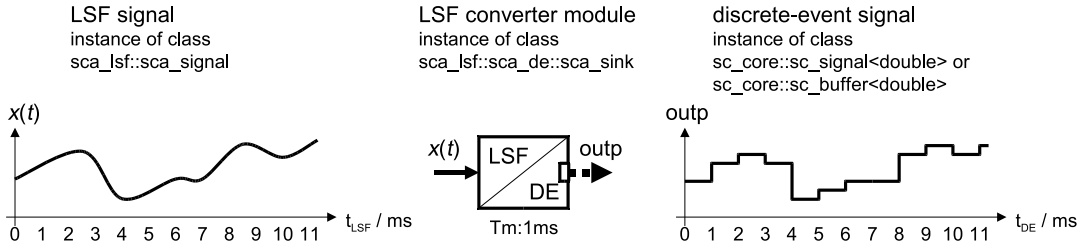


Figure 3.6. LSF converter module reading from an LSF input signal and writing to a discrete-event output signal

3.4.2. Reading from and writing to TDF models

In a similar way, LSF models can be connected to TDF models using converter modules with an internal port of class **sca_tdf::sca_in** or **sca_tdf::sca_out**.

Figure 3.7 shows the LSF primitive module **sca_lsf::sca_tdf::sca_source** reading from a TDF signal and writing to an LSF signal. In this example a module time step of 1 ms is assigned to the LSF converter module. The LSF model continuously reads the samples from the TDF input. The input samples are interpreted to

form a continuous-time signal, available at the output of the converter module (input samples shown as a dotted signal).

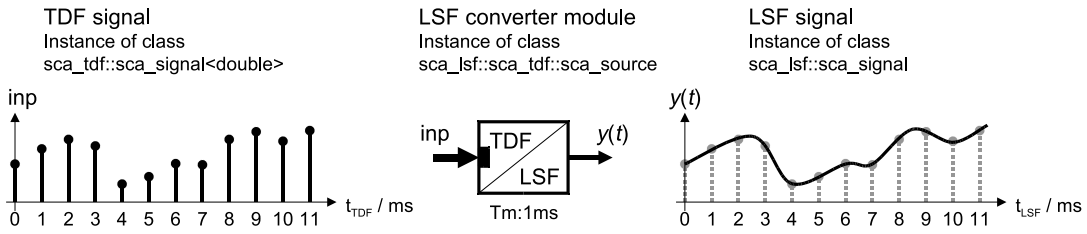


Figure 3.7. LSF converter module reading from a TDF input signal and writing to an LSF output signal

Figure 3.8 shows the LSF primitive module **sca_lsf::sca_tdf::sca_sink** reading an LSF signal and writing the equivalent values to a TDF signal. The samples at the output port are written at the time points, which are calculated from the assigned module time step of 1 ms.

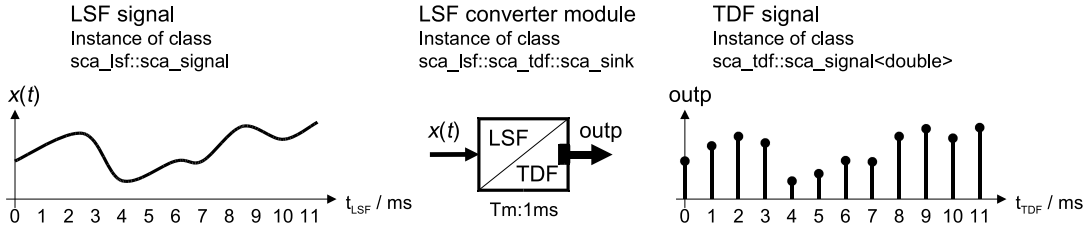


Figure 3.8. LSF converter module reading from an LSF input signal and writing to a TDF output signal

3.4.3. Using discrete-event or TDF control signals

Although not fundamentally different from the LSF converter modules described in the previous two sections, additional LSF primitives are available to control or scale variables or signals within an LSF equation system. The LSF primitives used for control can be identified by having an input port of class **sc_core::sc_in** or **sca_tdf::sca_in** of data type *bool*. Examples are the multiplexers (**sca_lsf::sca_de::sca_mux** and **sca_lsf::sca_tdf::sca_mux**) and demultiplexers (**sca_lsf::sca_de::sca_demux** and **sca_lsf::sca_tdf::sca_demux**). The primitives, which can scale variables or signals make use of the same ports, but using data type *double*. Examples are the multiplication primitives (**sca_lsf::sca_de::sca_gain**, and **sca_lsf::sca_tdf::sca_gain**). Note that if a parameter of an LSF module has changed, the corresponding LSF equation system will be reinitialized.

Figure 3.9 shows an example how LSF primitives can be used in a structural model to control or scale signals.

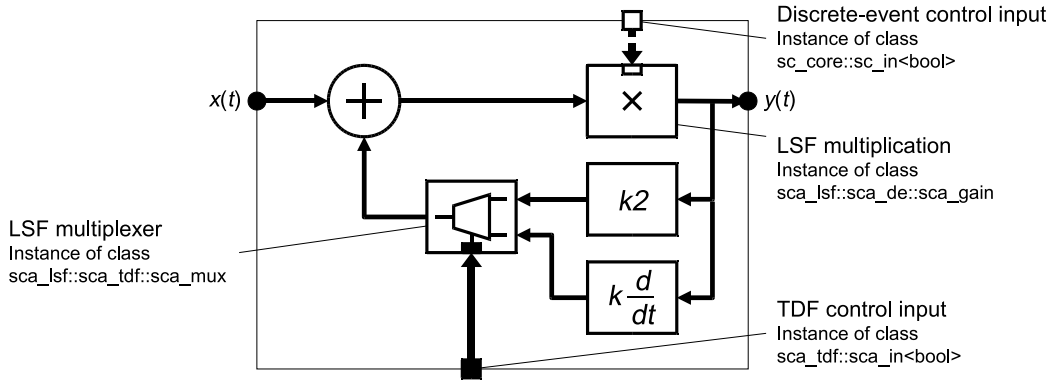


Figure 3.9. LSF model using discrete-event and TDF control signals

Similar as for the LSF converter modules described in Section 3.4, the discrete-event or TDF control signals are read with a fixed time step, which corresponds to the module time step. Only then the LSF equation system will be updated.

3.4.4. LSF model encapsulation

The converter modules described in the previous sections can be used to encapsulate an LSF model within a different model of computation. Figure 3.10 shows an example on how to use converter modules to and from the TDF model of computation to encapsulate LSF behavior. In this case, access to and from the LSF equation system use discrete-time signals following the TDF semantics, whereas the internal LSF signals and computations are continuous-time. This approach gives another possibility to embed continuous-time behavior in the TDF model of computation, besides the embedded linear dynamic equations for TDF modules described in Section 2.3.2.

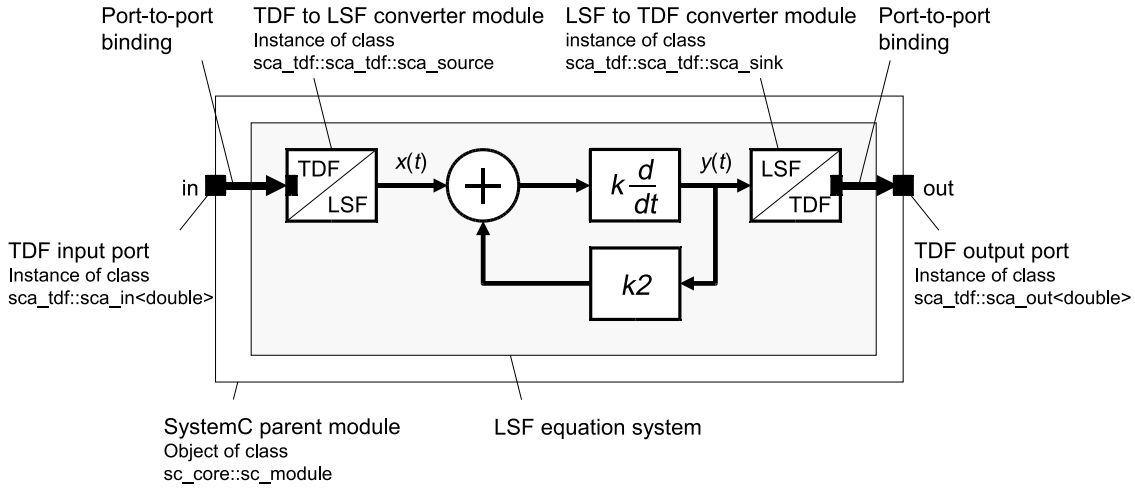


Figure 3.10. LSF equation system encapsulated for inclusion into a structural TDF model description by using converter modules

The example below shows the implementation of Figure 3.10.

```
SC_MODULE(lsf_in_tdf)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;

  sca_lsf::sca_add  add1;
  sca_lsf::sca_dot  dot1;
  sca_lsf::sca_gain gain1;
  sca_lsf::sca_tdf::sca_source tdf2lsf;
  sca_lsf::sca_tdf::sca_sink   lsf2tdf;

  lsf_in_tdf( sc_core::sc_module_name, double k, double k2 )
  : in("in"), out("out"), add1("add1"), dot1("dot1", k), gain1("gain1", k2), tdf2lsf("tdf2lsf"),
    lsf2tdf("lsf2tdf"), sig1("sig1"), sig2("sig2"), sig3("sig3"), sig4("sig4")
  {
    tdf2lsf.inp(in);
    tdf2lsf.y(sig1);

    add1.x1(sig1);
    add1.x2(sig3);
    add1.y(sig2);

    dot1.x(sig2);
    dot1.y(sig4);

    gain1.x(sig4);
    gain1.y(sig3);

    lsf2tdf.x(sig4);
    lsf2tdf.outp(out);
  }

 private:
```



```
  sca_lsf::sca_signal sig1, sig2, sig3, sig4;
};
```

A similar approach can be used to encapsulate an LSF model for inclusion into a structural discrete-event model description, using the converter modules to and from the discrete-event domain as explained in Section 3.4.1.

3.5. LSF execution semantics

In addition to the elaboration and simulation phases as defined in SystemC language standard IEEE 1666-2005, specific functionality is implemented for the elaboration and execution of LSF models.

As depicted in Figure 3.11, the elaboration phase includes the following steps:

- *LSF time step calculation and propagation*: Define the time step and check consistency inside each LSF model (see also Section 3.1.2).
- *LSF equation setup and solvability check*: Compose the LSF equation system from the contributing equations provided by the predefined LSF primitive modules and their relationship defined by the composition. Check whether the resulting equation system can be solved.

The steps for the simulation phase are:

- *LSF initialization*: First set all LSF signals to zero and then set the initial conditions of the system based on the potentially defined initial conditions of the LSF primitives.
- *LSF time-domain simulation*: The LSF equation system is solved numerically using appropriate time steps, which could be smaller than the assigned time step. The solver will at least provide results at the time points, calculated from the assigned time step.

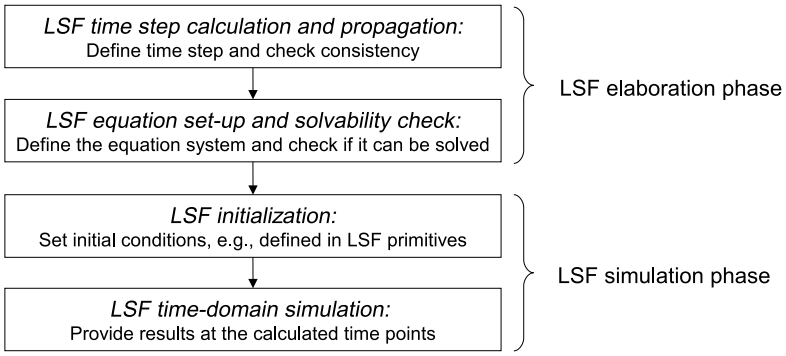


Figure 3.11. LSF elaboration and simulation phases

The elaboration and simulation phase are executed by starting a time-domain simulation using the function **sc_core::sc_start**. This is explained in Section 6.1.1.

3.6. Application examples

This section shows some basic application examples using Linear Signal Flow modeling.

3.6.1. PID controller

The LSF modeling formalism is very suitable to model *control systems*. An example of such a control system is shown in Figure 3.12. This example shows the use of a Proportional–Integral–Derivative (PID) controller, which is part of a control loop. The input of the PID controller is an error signal $e(t)$, which is the difference between a measured output value $y(t)$ of a certain device and the desired reference input y_0. The control output $u(t)$ generated by the PID controller, which regulates the behavior of the device under

control, will be such that the error signal will be minimized. The responsiveness and behavior of the PID controller to an error, either caused by a (sudden) change of the reference input or output value, depends on the PID controller characteristics defined by the parameters K_p, K_i, and K_d.

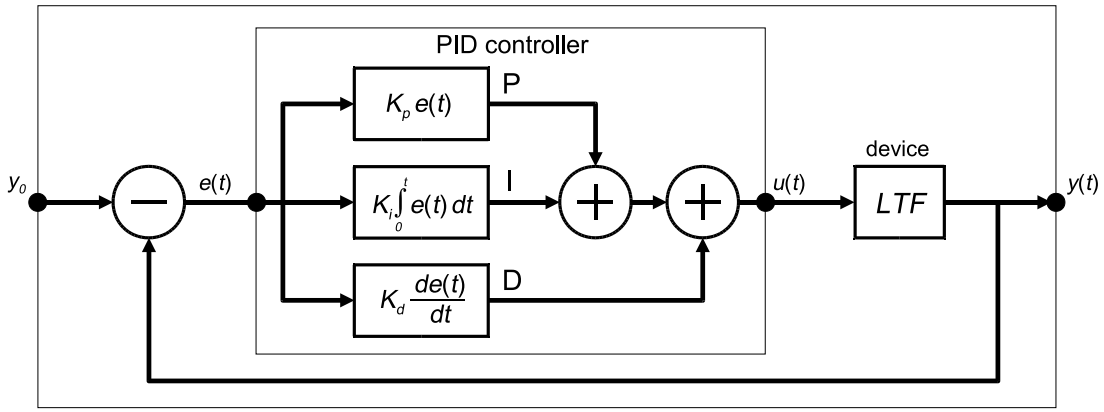


Figure 3.12. Block diagram of a PID controller within a control loop

The parameters K_p, K_i, and K_d are used within the PID controller to set the proportional, integral, and derivative terms, which are then summed to calculate the control output. The equation system of the PID controller, in which $e(t)$ is the error input signal and $u(t)$ is the controller output, then becomes:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_o^t e(t)dt + K_d \cdot \frac{de(t)}{dt}$$

The PID controller can be implemented by using LSF primitive modules in a parent module as shown below:

```
SC_MODULE(pid_controller)
{
  sca_lsf::sca_in  e;
  sca_lsf::sca_out u;

  sca_lsf::sca_gain  gain1;
  sca_lsf::sca_integ integ1;
  sca_lsf::sca_dot   dot1;
  sca_lsf::sca_add   add1, add2; ❶

  pid_controller( sc_core::sc_module_name, double kp, double ki, double kd ) ❷
  : e("e"), u("u"), gain1("gain1", kp), integ1("integ1", ki), dot1("dot1", kd), add1("add1"),
    add2("add2"), sig_p("sig_p"), sig_i("sig_i"), sig_d("sig_d"), sig_pi("sig_pi")
  {
    gain1.x(e);
    gain1.y(sig_p);

    integ1.x(e);
    integ1.y(sig_i);

    dot1.x(e);
    dot1.y(sig_d);

    add1.x1(sig_p);
    add1.x2(sig_i);
    add1.y(sig_pi);

    add2.x1(sig_pi);
    add2.x2(sig_d);
    add2.y(u);
  }

 private:
  sca_lsf::sca_signal sig_p, sig_i, sig_d, sig_pi;
};
```

❶ In order to sum the proportional, integral, and derivative terms, two adders are used, as each primitive adder module has only two inputs.

❷ The parameters for the PID controller can be assigned via the constructor, which allows their setting from the parent module (or **sc_main** function) in which the PID controller is instantiated.

3.6.2. Continuous-time sigma-delta modulator

Figure 3.13 shows the application of a continuous-time sigma-delta (CTSD) modulator architecture, containing a loop filter *H*(*s*), a quantizer and a digital to analog converter (DAC) in the feedback path. The loop filter is implemented using LSF primitives. The quantizer and DAC are implemented as TDF modules. LSF converter modules to and from the TDF model of computation are used, to be able to sample the continuous-time filter output signal *U*(*s*) to a discrete-time domain signal *V*(*z*), and to convert the discrete-time DAC output signal *W*(*z*) to a continuous-time feedback signal *T*(*s*).

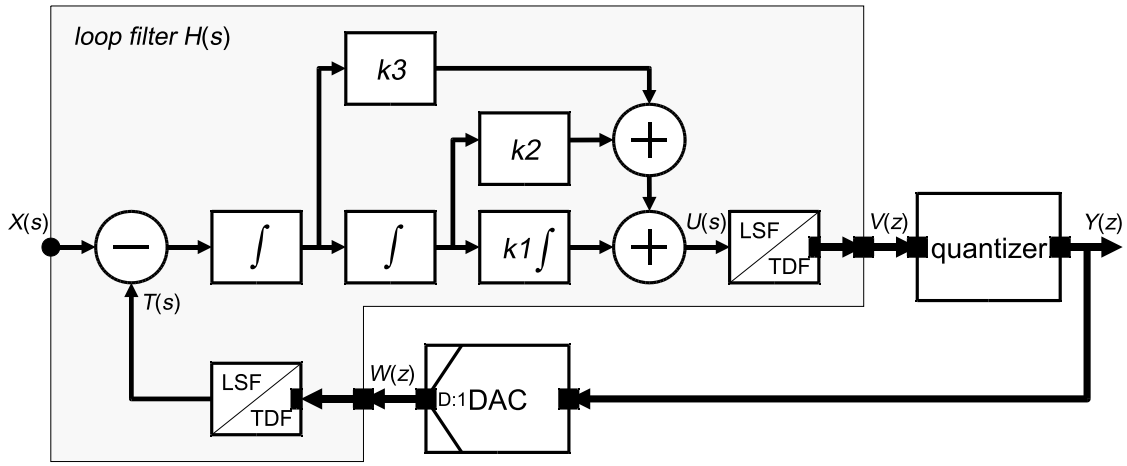


Figure 3.13. Block diagram of a continuous-time sigma-delta (CTSD) modulator

A 3rd-order loop filter is implemented using three integrators, which are cascaded and summed with weightings factors *k1*, *k2*, and *k3*. The corresponding transfer function *H*(*s*) for this loop filter then becomes:

$$H(s) = \frac{k_1 s^2 + k_2 s + k_3}{s^3}$$

The loop filter can be implemented by using LSF primitive modules in a parent module as shown below:

```
SC_MODULE(ctsd_loop_filter)
{
  sca_lsf::sca_in x;
  sca_tdf::sca_out<double> v;
  sca_tdf::sca_in<double>  w;

  sca_lsf::sca_tdf::sca_source tdf2lsf;
  sca_lsf::sca_sub   sub1;
  sca_lsf::sca_integ integ1, integ2, integ3;
  sca_lsf::sca_gain  gain2, gain3;
  sca_lsf::sca_add   add1, add2;
  sca_lsf::sca_tdf::sca_sink lsf2tdf;

  ctsd_loop_filter( sc_core::sc_module_name, double k1, double k2, double k3 )
  : x("x"), v("v"), w("w"), tdf2lsf("tdf2lsf"), sub1("sub1"), integ1("integ1", k1), integ2("integ2"),
    integ3("integ3"), gain2("gain2", k2), gain3("gain3", k3), add1("add1"), add2("add2"),
    lsf2tdf("lsf2tdf"), sig_t("sig_t"), sig_i("sig_1"), sig_i1("sig_i1"), sig_i2("sig_i2"),
    sig_i3("sig_i3"), sig_a1("sig_a1"), sig_a2("sig_a2"), sig_a3("sig_a3"), sig_u("sig_u")
  {
    tdf2lsf.inp(w);
    tdf2lsf.y(sig_t);

    sub1.x1(x);
    sub1.x2(sig_t);
    sub1.y(sig_i);

    integ3.x(sig_i);
    integ3.y(sig_i3);

    integ2.x(sig_i3);
    integ2.y(sig_i2);

    integ1.x(sig_i2);
    integ1.y(sig_i1);
```



```
    gain3.x(sig_i3);
    gain3.y(sig_a1);

    gain2.x(sig_i2);
    gain2.y(sig_a2);

    add1.x1(sig_a1);
    add1.x2(sig_a2);
    add1.y(sig_a3);

    add2.x1(sig_a3);
    add2.x2(sig_i1);
    add2.y(sig_u);

    lsf2tdf.x(sig_u);
    lsf2tdf.outp(v);
  }

 private:
  sca_lsf::sca_signal sig_t, sig_i, sig_i1, sig_i2, sig_i3;
  sca_lsf::sca_signal sig_a1, sig_a2, sig_a3, sig_u;
};
```

This page is intentionally left blank.

4. Electrical Linear Networks modeling

4.1. Modeling fundamentals

The Electrical Linear Networks model of computation introduces the use of electrical primitives and their interconnections to model conservative, continuous-time behavior. The ELN modeling style allows the instantiation of electrical primitives, which can be connected together using *electrical nodes*, to form an *electrical network*. The mathematical relations between the electrical primitives are defined at each node in the network, where both the potential (voltage) and flow (current) quantities are used according to Kirchhoff's voltage law (KVL) and Kirchhoff's current law (KCL). As such, the electrical network is represented by a set of differential algebraic equations, which will be resolved during simulation to determine the actual circuit behavior.
Figure 4.1 shows an example of an electrical network, with two resistors, a capacitor, and a current source. Such a network is called an *ELN model* and is composed of a set of connected *ELN primitive modules*, which will form together an *ELN equation system* or *cluster*. Each ELN primitive module can have one or more *ELN terminals*. The ELN primitive modules are interconnected via their terminals using *ELN nodes*. The reference or ground node, which always has a voltage of zero, is called *ELN reference node*. ELN terminals are also used as an interface to connect the ELN model with other ELN models.

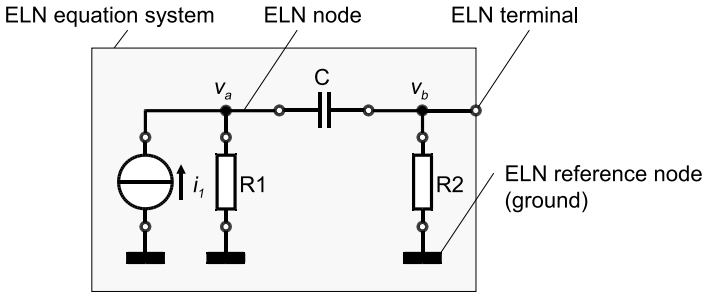


Figure 4.1. Example of a basic ELN model representing an electrical network

4.1.1. Setup of the equation system

The SystemC AMS extensions offer a finite set of ELN primitive modules such as sources (voltage or current), linear lumped elements (resistors, capacitors, inductors), linear distributed elements (transmission lines), ideal amplifier (nullor), ideal transformer, linear gyrator, and ideal switches. Similar to the LSF modeling style, ELN models can only be composed from these primitives, as there is no possibility to implement user-defined electrical primitives. Figure 4.2 shows some ELN lumped elements and their corresponding mathematical equations.

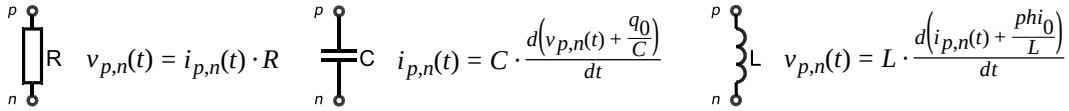


Figure 4.2. Examples of the basic ELN lumped elements: resistor (R), capacitor (C), and inductor (L) with their corresponding mathematical equations

When creating an ELN model (electrical network), the mathematical equations for each primitive and their relationship defined at each node will be used to compose the overall equation system. For example, the ELN model presented in Figure 4.1 will result in an ELN equation system for node A and B by following Kirchhoff's current and voltage laws, and using the contributed equations of each primitive as shown in Figure 4.2.

$$-i_1 + \frac{v_a}{R1} + C \cdot \frac{d\left(v_{a,b} + \frac{q_0}{C}\right)}{dt} = 0$$

$$\frac{v_b}{R2} - C \cdot \frac{d\left(v_{a,b} + \frac{q_0}{C}\right)}{dt} = 0$$

Note that the current through ELN primitives with two terminals is defined as the current flowing from terminal *p* to terminal *n*. This also holds for the current sources.

4.1.2. Time step assignment and propagation

Similar as for a TDF module, a time step can be assigned to an ELN module directly or can be assigned automatically using the propagation mechanism of the time step within an ELN equation system. In case an ELN model is connected to a TDF model, the time step from the connected TDF port(s) is propagated to the ELN model. Consistency between locally defined ELN module time steps and propagated time steps is essential. Otherwise, the time points for the solution of the ELN equation system or communication with the connected TDF model cannot be defined properly (see also Section 2.1.3). The time step should be defined at least at one location in the entire system.

During simulation, this ELN equation system is solved numerically at appropriate time steps, which could be smaller than the assigned time step. The solver will at least provide results at the time points, calculated from the assigned time steps.

4.2. Language constructs

4.2.1. ELN modules

An ELN module is a predefined electrical primitive, which can be used to build an electrical network. The available predefined ELN primitive modules are listed in Table 4.1 below. Appendix A gives the details for each ELN module.

ELN module name	Description
sca_eln::sca_r	Resistor
sca_eln::sca_c	Capacitor
sca_eln::sca_l	Inductor
sca_eln::sca_vcvs	Voltage controlled voltage source
sca_eln::sca_vccs	Voltage controlled current source
sca_eln::sca_ccvs	Current controlled voltage source
>sca_eln::sca_cccs	Current controlled current source
sca_eln::sca_nullor	Nullor (nullator - norator pair), ideal op-amp
sca_eln::sca_gyrator	Gyrator
sca_eln::sca_ideal_transformer	Ideal transformer
sca_eln::sca_transmission_line	Transmission line
sca_eln::sca_vsource	Independent voltage source
sca_eln::sca_isource	Independent current source
sca_eln::sca_tdf::sca_r, **sca_eln::sca_tdf_r**	Variable resistor controlled by a TDF input signal
sca_eln::sca_tdf::sca_c, **sca_eln::sca_tdf_c**	Variable capacitor controlled by a TDF input signal
sca_eln::sca_tdf::sca_l, **sca_eln::sca_tdf_l**	Variable inductor controlled by a TDF input signal
sca_eln::sca_tdf::sca_rswitch, **sca_eln::sca_tdf_rswitch**	Switch controlled by a TDF input signal

ELN module name	Description
sca_eln::sca_tdf::sca_vsource, **sca_eln::sca_tdf_vsource**	Voltage source driven by a TDF input signal
sca_eln::sca_tdf::sca_isource, **sca_eln::sca_tdf_isource**	Current source driven by a TDF input signal
sca_eln::sca_tdf::sca_vsink, **sca_eln::sca_tdf_vsink**	Converts voltage to a TDF output signal
sca_eln::sca_tdf::sca_isink, **sca_eln::sca_tdf_isink**	Converts current to a TDF output signal
sca_eln::sca_de::sca_r, **sca_eln::sca_de_r**	Variable resistor controlled by a discrete-event input signal
sca_eln::sca_de::sca_c, **sca_eln::sca_de_c**	Variable capacitor controlled by a discrete-event input signal
sca_eln::sca_de::sca_l, **sca_eln::sca_de_l**	Variable inductor controlled by a discrete-event input signal
sca_eln::sca_de::sca_rswitch, **sca_eln::sca_de_rswitch**	Switch controlled by a discrete-event input signal
sca_eln::sca_de::sca_vsource, **sca_eln::sca_de_vsource**	Voltage source driven by a discrete-event input signal
sca_eln::sca_de::sca_isource, **sca_eln::sca_de_isource**	Current source driven by a discrete-event input signal
sca_eln::sca_de::sca_vsink, **sca_eln::sca_de_vsink**	Converts voltage to a discrete-event output signal
sca_eln::sca_de::sca_isink, **sca_eln::sca_de_isink**	Converts current to a discrete-event output signal

Table 4.1. ELN primitive modules

Module time step

In order to solve the ELN equation system, a time step should be associated to the set of connected ELN modules as part of the elaboration phase. This can be done with the ELN module member function **set_timestep**. Alternatively, the ELN model can rely on the time step propagation mechanism, which passes the time step from module to module via its ports across the TDF, LSF, and ELN models of computation. So in cases where an ELN model is connected to a TDF model, the time step from the connected port, if available, is propagated to the ELN model. In case propagated time steps and user-defined time steps are used, consistency between these time steps is compulsory, similar as described in Section 2.1.3.

The module time step can be assigned by calling the member function **set_timestep** of the instantiated object within the constructor of the parent module, and passing a *double* value and the time unit or an object of type **sca_core::sca_time**, as shown in the following example:

```
SC_MODULE(my_eln_source)
{
  // terminal declaration
  sca_eln::sca_terminal p;

  // child module declaration
  sca_eln::sca_vsource v_src;

  SC_CTOR(my_eln_source)
  : p("p"),
    v_src("v_src", 0.0, 0.0, 1.0e-3, 1.0e3),  // 1 kHz sinusoidal source with an amplitude of 1 mV
    gnd("gnd")
  {
    v_src.set_timestep(0.25, sc_core::SC_MS); // set module timestep to 0.25 ms
    v_src.p(p);
    v_src.n(gnd);
  }
```



```
 private:
  sca_eln::sca_node_ref gnd;
};
```

4.2.2. ELN terminals

An ELN terminal is an object that can be used to connect several ELN models together, using ELN nodes which are bound to this terminal. Due to the conservative nature of the ELN modeling formalism, an ELN terminal is not defined as an input or output port; instead, these terminal are used to allow making connections with nodes of class **sca_eln::sca_node** or **sca_eln::sca_node_ref** (see Section 4.2.3). As ELN terminals are always used in a structural (parent) module, they can also be used to connect to the ELN child modules directly, following the *port-to-port* binding rule (see Section 4.3.1). ELN terminals make use of an internal data type, also called *electrical nature*, which prevents the usage of user-defined data types.

The example below shows how ELN terminals are used within an ELN structural model.

```
SC_MODULE(my_eln_model)
{
  // terminal declarations
  sca_eln::sca_terminal p; ❶
  sca_eln::sca_terminal n;

  SC_CTOR(my_eln_model) : p("p"), n("n") ❷
  {
    // model implementation here
  }
};
```

❶ ELN positive (p) and negative (n) terminal that carries a continuous-time and -value signal.

❷ Using the constructor initialization-list to assign the names "p" and "n" to the p and n terminals, respectively.

Specialized converter modules are available to connect ELN modules to the TDF or discrete-event domain. This is explained in Section 4.4. ELN terminals do not provide read or write access methods.

4.2.3. ELN nodes

ELN nodes are used to connect ELN primitive modules together. In this case, multiple ELN primitives share the same node (also called *net*). There are two classes of ELN nodes:

- ELN node of class **sca_eln::sca_node**.
- ELN reference node (ground) of class **sca_eln::sca_node_ref**.

The ELN nodes and reference nodes are used to set up the overall equation system. The example below shows how to use ELN nodes and ELN reference nodes.

```
  // node declarations
  sca_eln::sca_node     net1; // ELN node (called "net1")
  sca_eln::sca_node_ref gnd;  // ELN reference node (called ground, "gnd")
```

As in SystemC, the constructor initialization-list of the parent module can be used to assign a user-defined name to a node:

```
  // using the constructor initialization-list to assign the names to the declared ELN nodes
  SC_CTOR(my_eln_module) : net1("net1"), gnd("gnd") {}
```

Section 4.3 will describe the creation of structural ELN models and will show examples of assigning user-defined names to terminals and nodes.

4.3. Modeling continuous-time behavior

ELN models can be used to implement linear dynamic, continuous-time, conservative behavior. ELN models can only be composed using ELN primitive modules. Therefore an ELN model is always a structural model.

4.3.1. Structural composition of ELN modules

ELN modules should be instantiated as child modules inside a regular SystemC parent module created with the help of the macro **SC_MODULE** or by deriving publicly from **sc_core::sc_module**. This parent module also instantiates all necessary terminals to communicate with the outside world and internal nodes for the interconnection of the child modules. The parameterization of the instantiated modules as well as the interconnection of the modules should be done in the constructor (e.g., created with the help of the macro **SC_CTOR**) of the parent SystemC module.

Port (terminal) binding

In order to connect ELN modules in a proper way to other ELN modules and nodes, the following specific bindings are possible, as shown in Figure 3.3. The port binding rules are compatible and complementary to the SystemC rules.

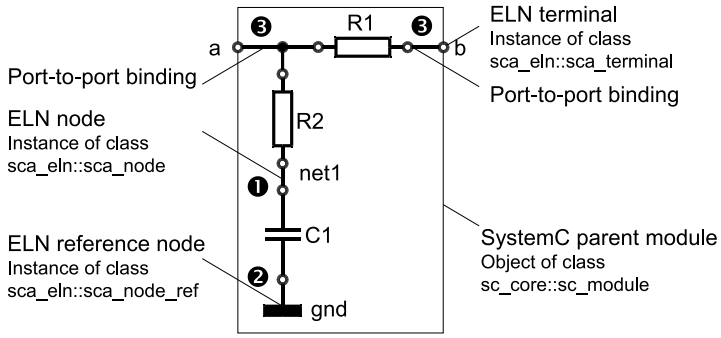


Figure 4.3. Port binding rules for ELN terminals

❶ Binding an ELN terminal to an ELN node.

❷ Binding an ELN terminal to an ELN reference node.

❸ Binding an ELN terminal to an ELN terminal of the parent module (port-to-port binding).

Furthermore, an ELN terminal should be bound to exactly one ELN node or reference node throughout the whole hierarchy. An ELN node or ELN reference node should be bound to one or more ELN terminals throughout the whole hierarchy.

ELN primitive modules, which have ports to connect to TDF or discrete-event signals or ports, should follow the port binding rules of the corresponding models of computation.

The example below shows the implementation of the structural composition of Figure 4.3.

```
SC_MODULE(my_structural_eln_model)
{
  sca_eln::sca_terminal a; ❶
  sca_eln::sca_terminal b;

  sca_eln::sca_r r1, r2; ❷
  sca_eln::sca_c c1;

  SC_CTOR(my_structural_eln_model)
  : a("a"), b("b"), r1("r1", 10e3), r2("r2", 100.0), c1("c1", 100e-6), net1("net1"), gnd("gnd") ❸
  {
    r1.p(a);  ❹
    r1.n(b);

    r2.p(a);
    r2.n(net1);

    c1.p(net1);
```



```
    c1.n(gnd);
  }

 private:
  sca_eln::sca_node net1; ❺
  sca_eln::sca_node_ref gnd;
};
```

❶ The ELN terminals declared inside this module of class **sc_core::sc_module** become part of the structural composition.
❷ The ELN primitive modules are declared within the parent module as child modules.
❸ The initialization-list in the parent module's constructor propagates the necessary configuration parameters to the ELN terminals, ELN nodes, and child modules.
❹ Port (terminal) binding is done inside the constructor of the parent module.
❺ Internal ELN nodes are used to connect the ELN terminals and child modules. These nodes are declared in the private space, as they should not be accessible from outside the module.

4.3.2. Continuous-time modeling

The example below shows a first-order low-pass filter, based on the same Laplace transfer function as described in Section 2.3.2. The cut-off frequency of the filter is defined by the time constant τ of the filter, which is the product of the resistance and capacitance value:

$$f_c = \frac{1}{2\pi\tau} = \frac{1}{2\pi RC}$$

The circuit implementation of this filter is rather simple, as shown in Figure 4.4.

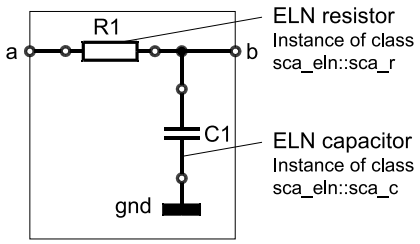


Figure 4.4. ELN circuit implementation of a first-order low-pass filter

The code implementation for the first-order low-pass filter, implemented as RC network is given below:

```
SC_MODULE(my_eln_filter)
{
  sca_eln::sca_terminal a;
  sca_eln::sca_terminal b;

  sca_eln::sca_r r1;
  sca_eln::sca_c c1;

  my_eln_filter( sc_core::sc_module_name, double r1_value, double c1_value )
  : a("a"), b("b"),  r1("r1", r1_value), c1("c1", c1_value), gnd("gnd"),
  {
    r1.n(a);
    r1.p(b);

    c1.n(b);
    c1.p(gnd);
  }

 private:
  sca_eln::sca_node_ref gnd;
};
```

Note that the time step for this network has not been defined in this ELN module. This means that this model relies on the time step propagation mechanism.

4.4. Interaction between ELN and discrete-event or TDF models

The ELN model of computation will setup and solve an equation system to simulate the modeled continuous-time behavior, based on the basic set of ELN primitive modules described in Section 4.2.1. Any "external" input value, e.g., from a discrete-event signal or TDF sample, need to be contributed to the equation system via one of these ELN primitive modules. Therefore, specialized ELN primitive modules with ports to the discrete-event domain and TDF models of computation are available, which are called *converter modules*. Main purpose of these modules is to establish an interface to convert and transfer data from one model of computation to the other.

4.4.1. Reading from and writing to discrete-event models

In order to connect ELN models with discrete-event models, the ELN converter modules with an internal port of class **sc_core::sc_in** or **sc_core::sc_out** should be used.

Figure 4.5 shows the ELN primitive modules **sca_eln::sca_de::sca_vsource** and **sca_eln::sca_de::sca_isource**, which read a discrete-event signal representing a real value and converting this value to an electrical voltage or current respectively. In this example a module time step of 1 ms is assigned to the ELN converter module. The ELN model continuously reads values from the input at the time points, which are calculated from the assigned time steps. The input value is assumed constant until the next value is read. The input values are interpreted to form a continuous-time signal, which is made available at the output of the converter module (read input samples shown as a dotted signal).

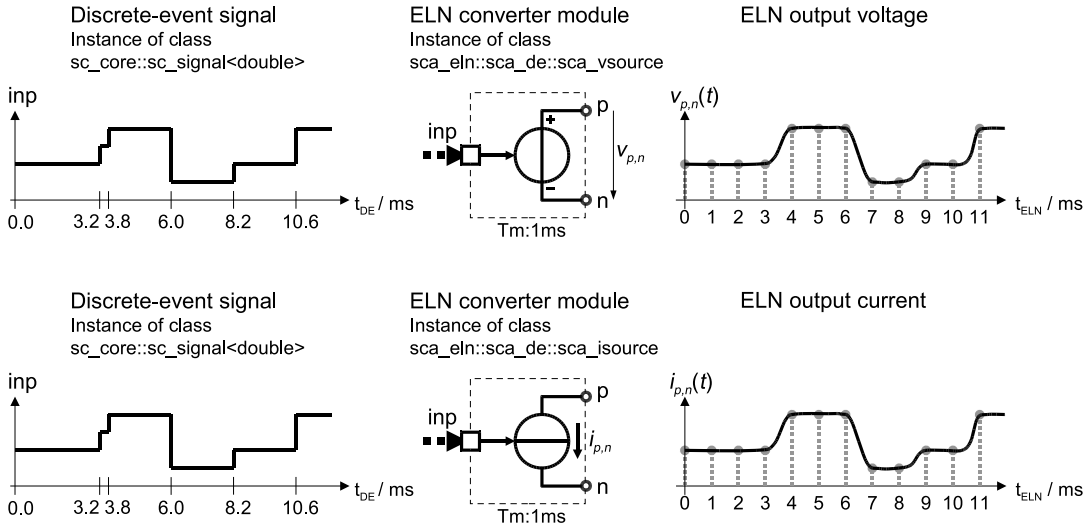


Figure 4.5. ELN converter modules reading double values from a discrete-event input signal and converting them to a continuous-time electrical voltage or current

Figure 4.6 shows the ELN primitive modules **sca_eln::sca_de::sca_vsink** and **sca_eln::sca_de::sca_isink**, which convert an electrical voltage or current to a real value, discrete-event signal. The values at the output port are written at the time points, calculated from to the assigned module time step of 1 ms.

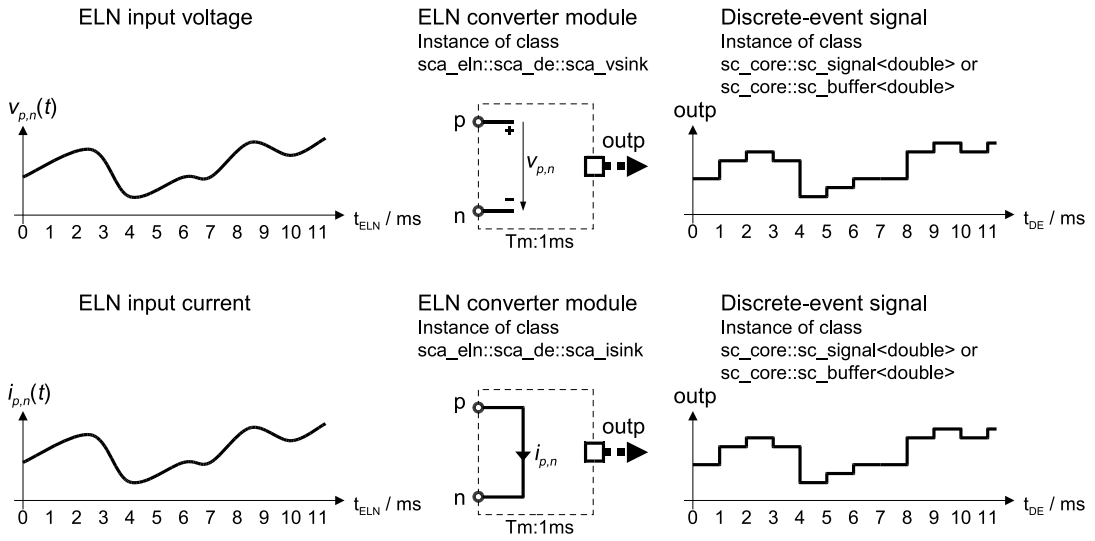


Figure 4.6. ELN converter modules to convert an electrical voltage or current to a real value, discrete-event output signal

4.4.2. Reading from and writing to TDF models

In a similar way, ELN models can be connected to TDF models using converter modules with an internal port of class **sca_tdf::sca_in** or **sca_tdf::sca_out**.

Figure 4.7 shows the ELN primitive modules **sca_eln::sca_tdf::sca_vsource** and **sca_eln::sca_tdf::sca_isource**, which read a value from a TDF signal and convert this value to an electrical voltage or current, respectively. In this example a module time step of 1 ms is assigned to the ELN converter module. The ELN model continuously reads the samples from the TDF input. The input samples are interpreted to form a continuous-time signal, which is made available at the output of the converter module (input samples shown as a dotted signal).

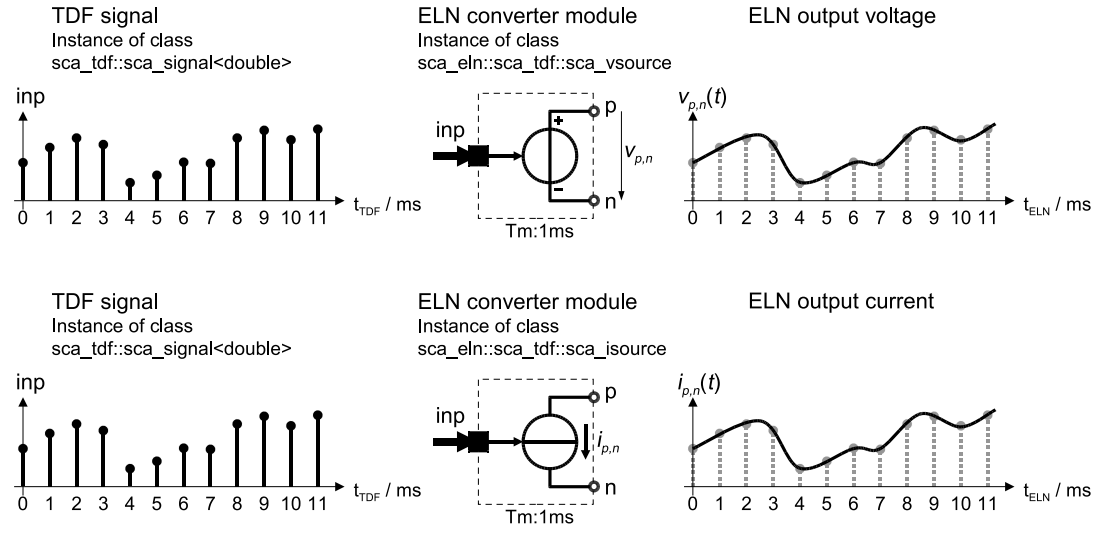


Figure 4.7. ELN converter modules reading double values from a TDF input signal and converting them to a continuous-time electrical voltage or current

Figure 4.8 shows the ELN primitive modules **sca_eln::sca_tdf::sca_vsink** and **sca_eln::sca_tdf::sca_isink**, which will convert an electrical voltage or current to a TDF signal. The samples at the output port are written at the calculated time points, which correspond to the assigned module time step of 1 ms.

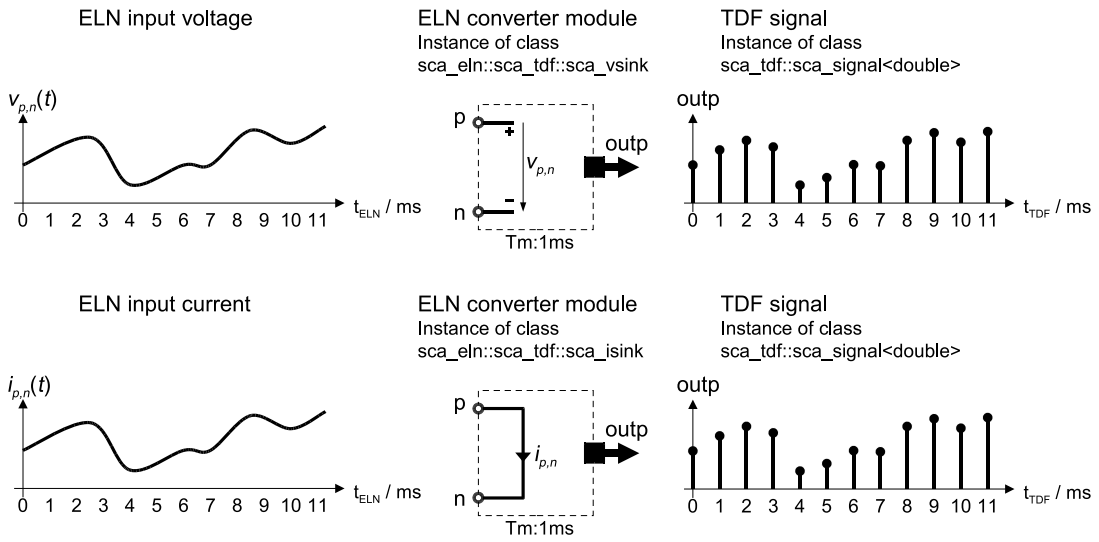


Figure 4.8. ELN converter modules convert an electrical voltage or current to a TDF output signal

4.4.3. ELN model encapsulation

The converter modules described in the previous sections can be used to encapsulate an ELN model within a different model of computation. Figure 4.9 shows an example on how to use converter modules to and from the TDF model of computation to encapsulate ELN behavior. In this case, access to and from the ELN equation system use discrete-time signals following the TDF semantics, whereas the internal ELN signals and computations are continuous-time.

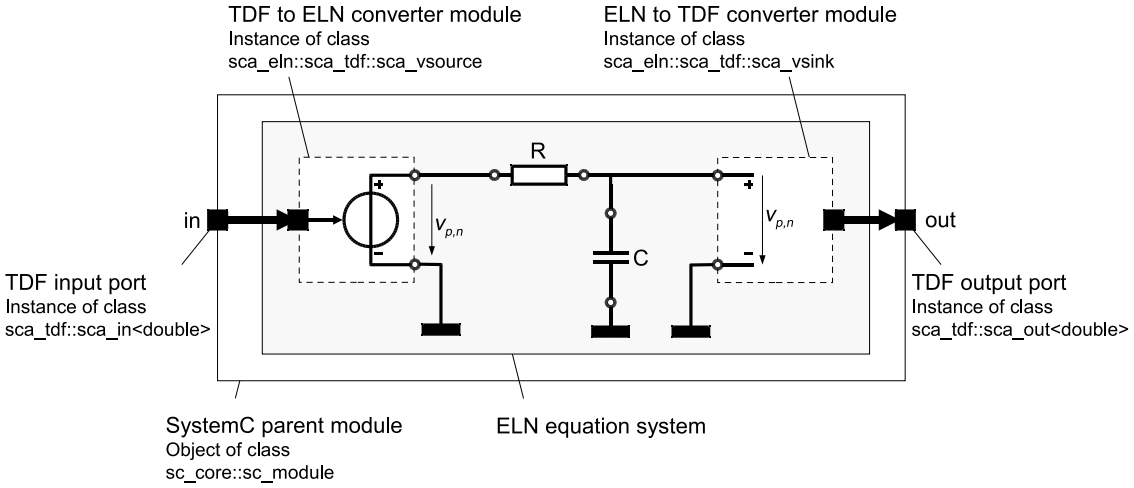


Figure 4.9. ELN equation system encapsulated for inclusion into
a structural TDF model description by using converter modules

The example below shows the implementation of Figure 4.9.

```
SC_MODULE(eln_in_tdf)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;

  sca_eln::sca_tdf::sca_vsource vin;
  sca_eln::sca_tdf::sca_vsink   vout;
  sca_eln::sca_r r;
  sca_eln::sca_c c;

  eln_in_tdf( sc_core::sc_module_name, double r_val, double c_val )
```



```
  : in("in"), out("out"), vin("vin"), vout("vout"), r("r", r_val), c("c", c_val),
    n1("n1"), n2("n2"), gnd("gnd")
  {
    vin.inp(in);
    vin.p(n1);
    vin.n(gnd);

    r.p(n1);
    r.n(n2);

    c.p(n2);
    c.n(gnd);

    vout.p(n2);
    vout.n(gnd);
    vout.outp(out);
  }

 private:
  sca_eln::sca_node n1, n2;
  sca_eln::sca_node_ref gnd;
};
```

A similar approach can be used to encapsulate an ELN model for inclusion into a structural discrete-event model description, using the converter modules to and from the discrete-event domain as explained in Section 4.4.1.

4.5. ELN execution semantics

In addition to the elaboration and simulation phases as defined in SystemC language standard IEEE 1666-2005, specific functionality is implemented for the elaboration and execution of ELN models. These additions are similar to the ones in LSF.

As depicted in Figure 4.10, the elaboration phase includes the following steps:

- *ELN time step calculation and propagation*: Define the time step and check consistency inside each ELN model (see also Section 4.1.2).
- *ELN equation setup and solvability check*: Compose the ELN equation system from the contributing equations provided by the predefined ELN primitive modules and their relationship defined by the composition. Check whether the resulting equation system can be solved.

The steps for the simulation phase are:

- *ELN initialization*: First set all ELN signals to zero and then set the initial conditions of the system based on the potentially defined initial conditions of the ELN primitives.
- *ELN time-domain simulation*: The ELN equation system is solved numerically using appropriate time steps, which could be smaller than the assigned time step. The solver will at least provide results at the time points, calculated from the assigned time step.

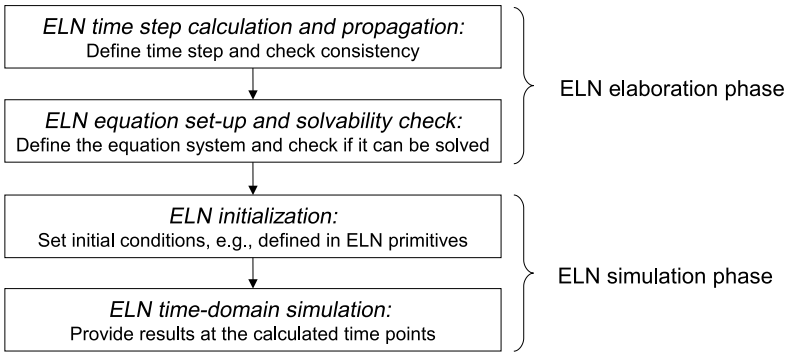


Figure 4.10. ELN elaboration and simulation phases

The elaboration and simulation phase are executed by starting a time-domain simulation using the function **sc_core::sc_start**. This is explained in Section 6.1.1.

4.6. Application examples

This section shows some basic application examples using ELN modeling.

4.6.1. POTS front-end

The Plain Old Telephone System (POTS) front-end is depicted in Figure 4.11. It consists of a phone, transmission line, a protection circuit and a subscriber line interface circuit (SLIC), which can be modeled naturally using ELN primitives. The interface from and to the POTS front-end are based on TDF or discrete-event signals.

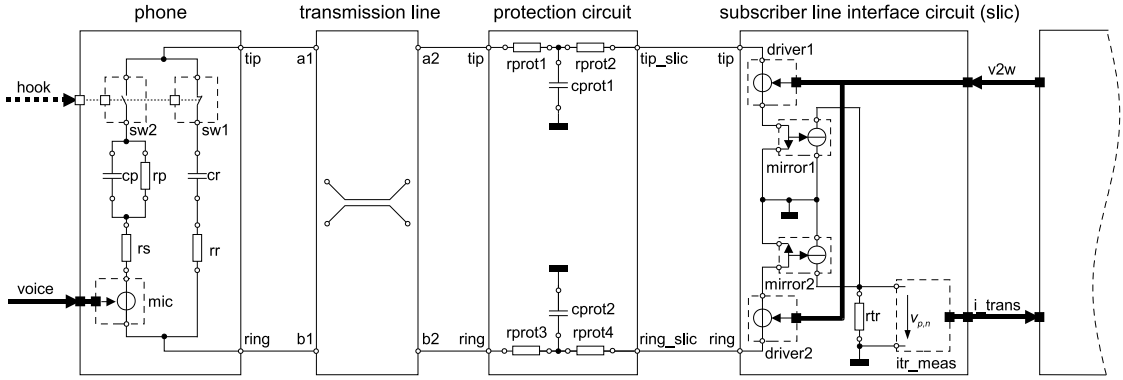


Figure 4.11. The Plain Old Telephone System (POTS) front-end

The implementation of the phone, protection circuit and SLIC are given below.

```
SC_MODULE(phone)
{
  // terminals and ports
  sca_eln::sca_terminal     tip;
  sca_eln::sca_terminal     ring;
  sca_tdf::sca_in<double>   voice;
  sc_core::sc_in<bool>      hook;

  // electrical primitives
  sca_eln::sca_de::sca_rswitch sw1;
  sca_eln::sca_de::sca_rswitch sw2;
  sca_eln::sca_c cr, cp;
  sca_eln::sca_r rr, rs, rp;
  sca_eln::sca_tdf::sca_vsource mic;

  phone( sc_core::sc_module_name nm, double cr_val = 1.0e-6, double rr_val = 1.0e3,
                                     double rs_val = 220.0, double cp_val = 115.0e-9,
                                     double rp_val = 820.0 )
  : tip("tip"), ring("ring"), voice("voice"), hook("hook"),
    sw1("sw1"), sw2("sw2"), cr("cr", cr_val), cp("cp", cp_val),
    rr("rr", rr_val), rs("rs", rs_val), rp("rp", rp_val), mic("mic"),
    w_offhook("w_offhook"), w_onhook("w_onhook"), w1("w1"), w2("w2"), wring("wring")
  {
    // architecture
    sw1.p(tip);
    sw1.n(w_onhook);
    sw1.ctrl(hook);
    sw1.off_state = true;

    sw2.p(tip);
    sw2.n(w_offhook);
    sw2.ctrl(hook);

    cr.p(wring);
    cr.n(w_onhook);

    rr.p(wring);
    rr.n(ring);

    rs.p(w1);
    rs.n(w2);

    cp.p(w1);
```

```
    cp.n(w_offhook);

    rp.p(w_offhook);
    rp.n(w1);

    mic.p(w2);
    mic.n(ring);
    mic.inp(voice);
  }

 private:
  // nodes
  sca_eln::sca_node w_offhook, w_onhook, w1, w2, wring;
};
```

```
SC_MODULE(protection_circuit)
{
  // terminals
  sca_eln::sca_terminal tip_slic;
  sca_eln::sca_terminal ring_slic;
  sca_eln::sca_terminal tip;
  sca_eln::sca_terminal ring;

  // electrical primitives
  sca_eln::sca_r rprot1, rprot2, rprot3, rprot4;
  sca_eln::sca_c cprot1, cprot2;

  protection_circuit( sc_core::sc_module_name, double rprot1_val = 20.0, double rprot2_val = 20.0,
                                               double rprot3_val = 20.0, double rprot4_val = 20.0,
                                               double cprot1_val = 18.0e-9,
                                               double cprot2_val = 18.0e-9 )
  : tip_slic("tip_slic"), ring_slic("ring_slic"), tip("tip"), ring("ring"),
    rprot1("rprot1", rprot1_val), rprot2("rprot2", rprot2_val),
    rprot3("rprot3", rprot3_val), rprot4("rprot4", rprot4_val),
    cprot1("cprot1", cprot1_val), cprot2("cprot2", cprot2_val),
    n_tip("n_tip"), n_ring("n_ring"), gnd("gnd")
  {
    // architecture
    rprot1.p(tip);
    rprot1.n(n_tip);

    rprot2.p(tip_slic);
    rprot2.n(n_tip);

    cprot1.p(n_tip);
    cprot1.n(gnd);

    rprot3.p(ring);
    rprot3.n(n_ring);

    rprot4.p(ring_slic);
    rprot4.n(n_ring);

    cprot2.p(n_ring);
    cprot2.n(gnd);
  }

 private:
  // nodes
  sca_eln::sca_node n_tip, n_ring;
  sca_eln::sca_node_ref gnd;
};
```

```
SC_MODULE(slic)
{
  // terminals and ports
  sca_eln::sca_terminal tip;
  sca_eln::sca_terminal ring;
  sca_tdf::sca_in<double> v2w;
  sca_tdf::sca_out<double> i_trans;

  // electrical primitives
  sca_eln::sca_tdf::sca_vsource driver1, driver2;
  sca_eln::sca_tdf::sca_vsink itr_meas;
  sca_eln::sca_cccs mirror1, mirror2;
  sca_eln::sca_r rtr;

  slic( sc_core::sc_module_name, double scale_v_tr = 1.0, double scale_i_tr = 1.0 )
  : tip("tip"), ring("ring"), v2w("v2w"), i_trans("i_trans"),
    driver1("driver1", scale_v_tr/2.0), driver2("driver2", scale_v_tr/2.0),
    itr_meas("itr_meas", scale_i_tr),
    mirror1("mirror1", 0.5), mirror2("mirror2", -0.5), rtr("rtr", 1.0),
```



```
    n_tr_i("n_tr_i"), n_tip_gnd("n_tip_gnd"), n_ring_gnd("n_ring_gnd"),
    gnd("gnd")
  {
    // architecture
    driver1.inp(v2w);
    driver1.p(tip);
    driver1.n(n_tip_gnd);

    driver2.inp(v2w);
    driver2.p(ring);
    driver2.n(n_ring_gnd);

    mirror1.ncp(n_tip_gnd);
    mirror1.ncn(gnd);
    mirror1.np(n_tr_i);
    mirror1.nn(gnd);

    mirror2.ncp(n_ring_gnd);
    mirror2.ncn(gnd);
    mirror2.np(n_tr_i);
    mirror2.nn(gnd);

    rtr.p(n_tr_i);
    rtr.n(gnd);

    itr_meas.p(n_tr_i);
    itr_meas.n(gnd);
    itr_meas.outp(i_trans);
  }

 private:
  // nodes
  sca_eln::sca_node n_tr_i, n_tip_gnd, n_ring_gnd;
  sca_eln::sca_node_ref gnd;
};
```

The implementation of the POTS front-end is done in the function **sc_main**, which is the main program. Only the instantiation and structural composition is shown here.

```
int sc_main(int argc,char* argv[])
{
  sca_eln::sca_node n_slic_tip, n_slic_ring;
  sca_eln::sca_node n_tip_a1, n_tip_a2, n_ring_b1, n_ring_b2;
   transmission_line;

  sca_tdf::sca_signal<double> s_v_in;
  sca_tdf::sca_signal<double> s_i_trans;

  sca_tdf::sca_signal<double> s_voice;
  sc_core::sc_signal<bool>    s_hook;

  // testbench modules
  ...

  slic i_slic("i_slic");
    i_slic.tip(n_slic_tip);
    i_slic.ring(n_slic_ring);
    i_slic.v2w(s_v_in);
    i_slic.i_trans(s_i_trans);

  protection_circuit i_protection_circuit("i_protection_circuit");
    i_protection_circuit.tip_slic(n_slic_tip);
    i_protection_circuit.ring_slic(n_slic_ring);
    i_protection_circuit.tip(n_tip_a2);
    i_protection_circuit.ring(n_ring_b2);

  sca_eln::sca_transmission_line i_transmission_line("i_transmission_line",
                                                     50.0, sc_core::SC_ZERO_TIME, 0.0);
    i_transmission_line.a1(n_tip_a1);
    i_transmission_line.b1(n_ring_b1);
    i_transmission_line.a2(n_tip_a2);
    i_transmission_line.b2(n_ring_b2);

  phone i_phone("i_phone");
    i_phone.tip(n_tip_a1);
    i_phone.ring(n_ring_b1);
    i_phone.voice(s_voice);
    i_phone.hook(s_hook);

  ...
};
```

This page is intentionally left blank.

5. Small-signal frequency-domain analyses

5.1. Modeling fundamentals

To analyze the frequency-domain behavior of an analog/mixed-signal system, varying small signals, called *alternating-current* (AC) signals, at different frequencies are used to stimulate and analyze the steady-state response of the circuit. Either small-signal sinusoidal sources or noise sources are used, and applied to the circuit, which is being linearized around a given *direct-current* (DC) operating-point. This means that large-signal behavior, such as non-linearities causing distortion, are not captured during small-signal frequency-domain analyses.

These AC-domain analysis methods can compute the small-signal frequency-domain behavior of the entire analog/mixed-signal system, which can be composed of modules from the available models of computation. TDF modules can embed a user-defined small-signal frequency-domain description. For LSF and ELN primitive modules, the small-signal frequency-domain behavior is implicitly part of the primitive's description. Figure 5.1 shows an example of a mixed-signal system containing TDF, LSF and ELN models. The modules labeled with "AC" have, besides their time-domain description, a small-signal frequency-domain representation associated with it. Based on the structural composition, a *linear complex equation* is extracted.

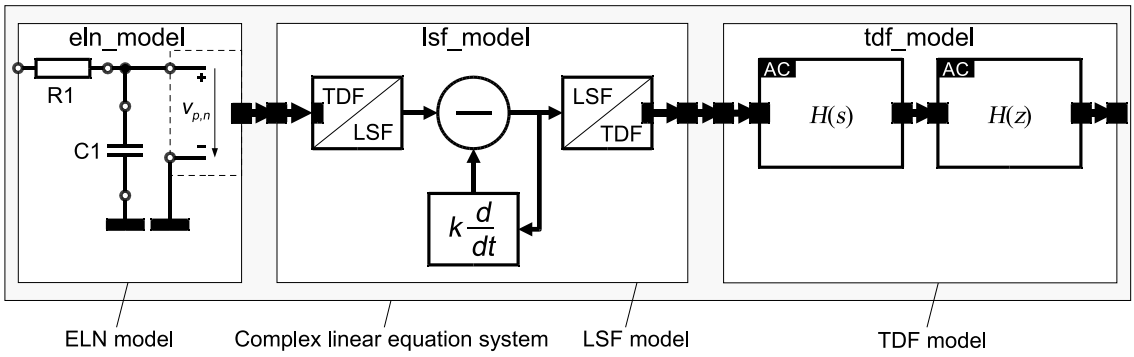


Figure 5.1. Small-signal frequency-domain description using TDF, LSF and ELN modules

5.1.1. Setup of the equation system

The linear complex equation system will make use of the TDF cluster as well as the LSF and ELN equation systems, which are initially defined for time-domain simulation. Transformation of LSF and ELN equation systems from the time-domain representation into the small-signal frequency-domain representation is done using the Laplace transform rules. Generally, for a given function $f(t)$, the following substitutions will be applied to the time-domain-oriented ELN and LSF equation systems:

- A derivation d/dt is substituted by $j\omega$.
- An integration is substituted by $1/j\omega$.
- A delay $f(t\text{-}delay)$ is substituted by $e^{-j\omega \cdot delay}$

Substitution will result in the frequency-domain function $F(j\omega)$ for the LSF and ELN contributions.

TDF modules allow the definition of user-defined small-signal frequency-domain behavior as part of the primitive definition. There is no mechanism available to derive an "AC representation". It is entirely the responsibility of the user to ensure the consistency of the defined frequency-domain and time-domain representations. How to implement small-signal frequency-domain behavior in TDF modules is discussed in Section 5.2.1.

5.1.2. Analysis methods

Two types of analyses are supported:

1. Small-signal frequency-domain analysis: Solves for each frequency point the linear complex equation system, including all small-signal frequency-domain source contributions.

2. Small-signal frequency-domain noise analysis: Solves the linear complex equation system for each frequency point and each small-signal frequency-domain noise source contribution, whereby all contributions of small-signal frequency-domain sources and small-signal frequency-domain noise sources, except the currently activated noise source, are set to zero.

The result of a small-signal frequency-domain or noise analysis is the steady-state response or transfer function of the circuit, described from the input port to the output port of the overall system. During analysis, the resulting linear complex equation system is solved for the given frequency points.

5.2. Language constructs

5.2.1. Small-signal frequency-domain description in TDF modules

The small-signal frequency-domain behavior of a TDF module can be defined in the member function **ac_processing**. The description should be written in the form of a linear complex transfer function, capturing the behavior from the TDF input port to the TDF output port. Different functions are available to define the linear complex transfer function, as presented in the next sections. For these calculations, a data container of type **sca_util::sca_complex** should be used.

The example below shows the implementation of a transfer function $H(s) = 1$. The intermediate result is stored in a variable *res* of type **sca_util::sca_complex**, which is assigned to the TDF output port. More details on the port access methods are given in the next section.

```
SCA_TDF_MODULE(flat_response)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;

  SCA_CTOR(flat_response) {}

  void processing()
  {
    out.write( in.read() );
  }

  void ac_processing()
  {
    double h = 1.0; // flat frequency response H(s) = 1
    sca_util::sca_complex res;

    res = h * sca_ac_analysis::sca_ac(in);
    sca_ac_analysis::sca_ac(out) = res;
  }
};
```

In case a small-signal frequency-domain analysis is performed, but no member function **ac_processing** is defined, or if no complex value is assigned to one or more TDF output ports, all related port values are set to zero, independently from the available value(s) at the input ports.

Note that there is no automatic consistency check between the time- and frequency-domain descriptions, as these definitions are used-defined.

5.2.2. Port access

For small-signal frequency-domain analysis, the complex value of all TDF ports, excluding the converter ports, can be accessed by using the function **sca_ac_analysis::sca_ac** with as argument the port instances, as shown in the previous example. This access method is independent from the port type required in time-domain simulation.

For input ports, the function **sca_ac_analysis::sca_ac** returns a constant reference to a value of type **sca_util::sca_complex**, which means that no value can be assigned to a TDF input port. For output ports, the function returns a reference to a value of type **sca_util::sca_complex**, allowing the assignment of a contribution for small-signal frequency-domain analysis.

For small-signal frequency-domain noise analysis, a noise source independent from the input port values, can be assigned to a TDF output port using the function **sca_ac_analysis::sca_ac_noise**. In this case, a value assigned using the function **sca_ac_analysis::sca_ac** will be ignored.

Note that the values returned from the functions **sca_ac_analysis::sca_ac** and **sca_ac_analysis::sca_ac_noise** are implementation-defined and have no physical interpretation. These values can only be used to describe the complex linear relation between the input and output ports, accessed using these port access functions.

5.3. Utility functions

The SystemC AMS extensions offer a set of utility functions, which can be used within the the member function **ac_processing** to define the small-signal frequency-domain behavior. Note that these functions cannot be used in the time-domain processing method **processing**.

5.3.1. Frequency-domain delay

The function **sca_ac_analysis::sca_ac_delay** can be used to implement a continuous-time delay, defined as $e^{-j\omega \cdot delay}$. The next example is an extension of the TDF delay example presented in Section 2.3.5. The delay is now a module parameter, and used to initialize the delay samples to 0.0 for the time-domain simulation. Note that the delay parameter is an integer value, reflecting the number of samples which will be inserted for time-domain simulation, using a discrete time step. The member function **ac_processing** implements the frequency-domain behavior of this delay. First, the delay is translated in a continuous-time variant, using the member function **get_timestep** multiplied with the number of delayed samples. This value of type **sca_core::sca_time** is passed as argument to the function **sca_ac_analysis::sca_ac_delay**, which defines the delay in the frequency domain.

```
SCA_TDF_MODULE(my_tdf_ac_delay)
{
  sca_tdf::sca_in<double> in;
  sca_tdf::sca_out<double> out;

  my_tdf_ac_delay( sc_core::sc_module_name, unsigned long delay_ )
  : in("in"), out("out"), delay(delay_) {}

  void set_attributes()
  {
    out.set_delay(delay);
  }

  void initialize()    // time-domain initialization
  {
    for( unsigned long i = 0; i < delay; i++ )
      out.initialize( 0.0, i );
  }

  void processing()    // time-domain implementation
  {
    out.write( in.read() );
  }

  void ac_processing() // frequency-domain implementation
  {
    sca_core::sca_time ct_delay = out.get_timestep() * delay; // calculate continuous-time delay

    sca_ac_analysis::sca_ac(out) = sca_ac_analysis::sca_ac(in) *
                                   sca_ac_analysis::sca_ac_delay( ct_delay );
  }

 private:
   unsigned long delay;
};
```

5.3.2. Laplace transfer functions

The frequency-domain descriptions of the Laplace transfer functions in the numerator-denominator and zero-pole form are available, using the utility functions **sca_ac_analysis::sca_ac_ltf_nd** or

sca_ac_analysis::sca_ac_ltf_zp, respectively. They can be used in combination with the time-domain representation, as shown in the example below.

```
SCA_TDF_MODULE(ltf_filter_ac)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;

  ltf_filter_ac( sc_core::sc_module_name nm, double fc_, double h0_ = 1.0 )
  : in("in"), out("out"), fc(fc_), h0(h0_) {}

  void initialize()
  {
    num(0) = 1.0;
    den(0) = 1.0;
    den(1) = 1.0 / ( 2.0 * M_PI * fc );
  }

  void processing()    // time-domain implementation
  {
    out.write( ltf_nd( num, den, in.read(), h0 ) );
  }

  void ac_processing() // frequency-domain implementation
  {
    sca_ac_analysis::sca_ac(out) = sca_ac_analysis::sca_ac_ltf_nd(
                                     num, den, sca_ac_analysis::sca_ac(in), h0 );
  }

 private:
  sca_tdf::sca_ltf_nd ltf_nd;            // Laplace transfer function
  sca_util::sca_vector<double> num, den; // numerator and denominator coefficients
  double fc; // 3dB cutoff frequency in Hz
  double h0; // DC gain
};
```

5.3.3. S-domain definitions

The function **sca_ac_analysis::sca_ac_s** supports frequency-domain representations defined in the *s*-domain, by using the Laplace operator $s^n = (j\omega)^n$.

Figure 5.2 shows the definition and frequency response $H(s)$ and implementation of a second order low-pass filter, implemented in the time- and frequency-domain.

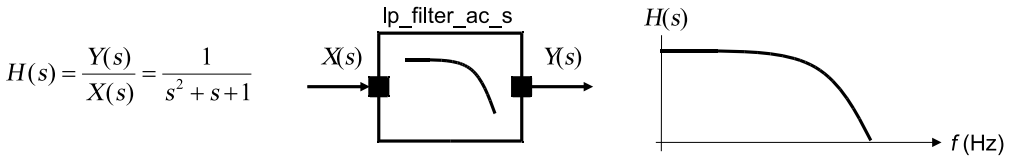


Figure 5.2. Frequency response of second order low-pass filter implemented in the s-domain

```
SCA_TDF_MODULE(lp_filter_ac_s)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;

  SCA_CTOR(lp_filter_ac_s) : in("in"), out("out") {}

  void initialize()
  {
    num(0) = 1.0;
    den(0) = 1.0;
    den(1) = 1.0;
    den(2) = 1.0;
  }

  void processing()    // time-domain implementation
  {
    out.write( ltf_nd( num, den, in.read(), 1.0 ) );
  }

  void ac_processing() // frequency-domain implementation
  {
```



```
    sca_util::sca_complex h = 1.0 / ( sca_ac_analysis::sca_ac_s(2) +
                                      sca_ac_analysis::sca_ac_s(1) + 1.0 );

    sca_ac_analysis::sca_ac(out) = h * sca_ac_analysis::sca_ac(in);
  }

 private:
  sca_tdf::sca_ltf_nd ltf_nd;
  sca_util::sca_vector<double> num, den;
};
```

Alternatively, the frequency-domain behavior can be implemented using the relation $s = j\omega$. The member function **ac_processing** from the previous example can be replaced with an implementation which uses the function **sca_ac_analysis::sca_ac_w**, which returns the current angular frequency in radians/seconds:

```
  void ac_processing() // frequency-domain implementation using s = j*w
  {
    sca_util::sca_complex s = sca_util::SCA_COMPLEX_J * sca_ac_analysis::sca_ac_w();
    sca_util::sca_complex h = 1.0 / ( s * s + s + 1.0 );

    sca_ac_analysis::sca_ac(out) = h * sca_ac_analysis::sca_ac(in);
  }
```

According to the relation $\omega = 2\pi f$, the frequency term can be used as well. The implementation using the function **sca_ac_analysis::sca_ac_f**, which returns the current frequency in Hertz, becomes:

```
  void ac_processing() // frequency-domain implementation using s = j*2*PI*f
  {
    sca_util::sca_complex s = sca_util::SCA_COMPLEX_J * 2.0 * M_PI * sca_ac_analysis::sca_ac_f();
    sca_util::sca_complex h = 1.0 / ( s * s + s + 1.0 );

    sca_ac_analysis::sca_ac(out) = h * sca_ac_analysis::sca_ac(in);
  }
```

5.3.4. Z-domain definitions

The function **sca_ac_analysis::sca_ac_z** supports frequency-domain representations defined in the *z*-domain, by using the the operator z^n ($= e^{j\omega \cdot n \cdot tstep}$). Where *n* is an integer defining the delay, and *tstep* is the timestep between the delays. In case this argument is not used, *tstep* will be defined as the timestep returned by the member function **get_timestep**.

Figure 5.3 shows the definition and frequency response $H(z)$ of a comb-filter.

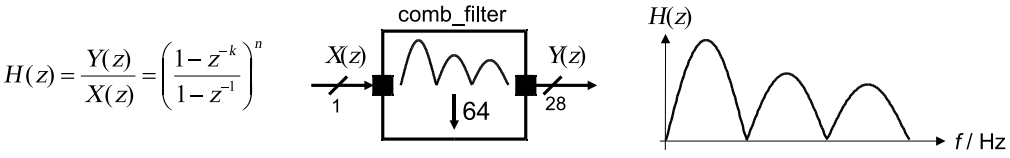


Figure 5.3. Frequency response of a comb-filter implemented in the z-domain

For the frequency-domain implementation, the function **sca_ac_analysis::sca_ac_z** is used, as shown in the example below.

```
SCA_TDF_MODULE(comb_filter)
{
  sca_tdf::sca_in<bool> in;
  sca_tdf::sca_out<sc_dt::sc_int<28> > out;

  comb_filter( sc_core::sc_module_name, int k_ = 64, int n_ = 3 )
  : in("in"), out("out"), k(k_), n(n_) {}

  void set_attributes()
  {
    in.set_rate(k);
    out.set_rate(1);
  }

  void ac_processing() // frequency-domain implementation
```



```
  {
    // complex transfer function
    sca_util::sca_complex h = pow( ( 1.0 - sca_ac_analysis::sca_ac_z(-k) ) /
                                   ( 1.0 - sca_ac_analysis::sca_ac_z(-1) ), n );

    sca_ac_analysis::sca_ac(out) = h * sca_ac_analysis::sca_ac(in) ;
  }

  void processing()    // time-domain implementation
  {
    int x, y, i;
    for( i = 0; i < k; i++) {
      x = in.read(i);
      ...
    }
    out.write(y);

  }

 private:
  int k; // decimation factor
  int n; // order of filter
};
```

5.3.5. Detection of small-signal frequency-domain analyses

The utility functions **sca_ac_analysis::sca_ac_is_running** and **sca_ac_analysis::sca_ac_noise_is_running** can be used within the member function **processing** or **ac_processing** of a TDF module, to implement specific behavior, which depends on the type of analysis running.

The function **sca_ac_analysis::sca_ac_is_running** returns true when a small-signal frequency-domain or noise analysis is running. The function **sca_ac_analysis::sca_ac_noise_is_running** only returns true if a small-signal frequency-domain noise analysis is running.

The example below shows the implementation of a sinusoidal source, which can be used in time-domain and frequency-domain simulations.

```
SCA_TDF_MODULE(sin_src)
{
  sca_tdf::sca_out<double> out;

  sin_src( sc_core::sc_module_name nm, double offset_= 0.0, double ampl_= 1.0,
           double noise_ampl_= 0.1, double freq_ = 1.0e3,
           sca_core::sca_time Tm_ = sca_core::sca_time(0.125, sc_core::SC_MS) )
  : out("out"), offset(offset_), ampl(ampl_), noise_ampl(noise_ampl_), freq(freq_), Tm(Tm_)
  {}

  void set_attributes()
  {
    set_timestep(Tm);
  }

  void processing()
  {
    double t = get_time().to_seconds(); // actual time

    out.write( offset + ampl * std::sin( 2.0*M_PI*freq*t ) );
  }

  void ac_processing()
  {
    if( sca_ac_analysis::sca_ac_noise_is_running() ) ❶
      sca_ac_analysis::sca_ac_noise(out) = noise_ampl;
    else
      sca_ac_analysis::sca_ac(out) = ampl;
  }
 private:
  double offset, ampl, noise_ampl, freq;
  sca_core::sca_time Tm;
};
```

❶ Ony for small-signal frequency-domain noise analysis, the function **sca_ac_analysis::sca_ac_noise_is_running** returns true. In this case, the noise amplitude of the source is set.

5.4. Small-signal frequency-domain analysis with combined TDF, LSF and ELN models

As already stated in the introduction of this chapter, the small-signal frequency-domain analysis is able to extract the frequency behavior of the entire analog/mixed-signal system. The frequency response of the entire system can be analyzed by using TDF modules, which have their frequency-domain behavior defined in their member function **ac_processing**, plus the frequency-domain description of LSF and ELN primitive modules, which is extracted from the LSF and ELN equation system during elaboration.

The implementation shown below is based on the module composition as presented in Figure 5.1. The example shows time-, frequency-domain and noise simulation. The results are written to different trace files.

```
int sc_main(int argc, char* argv[])
{
  sca_eln::sca_node net1;
  sca_tdf::sca_signal<double> sig1, sig2, sig3;

  ...             // source and sink

  eln_model a("a");
    a.p(net1);
    a.outp(sig1);

  lsf_model b("b");
    b.in(sig1);
    b.out(sig2);

  tdf_model c("c");
    c.in(sig2);
    c.out(sig3);

  // tracing
  sca_util::sca_trace_file* tf = sca_util::sca_create_tabular_trace_file("trace.dat");

  sca_util::sca_trace(tf, net1, "net1");
  sca_util::sca_trace(tf, sig1, "sig1");
  sca_util::sca_trace(tf, sig2, "sig2");
  sca_util::sca_trace(tf, sig3, "sig3");

  // start time-domain simulation
  sc_core::sc_start(10, sc_core::SC_MS);

  tf->reopen("ac_trace.dat");
  tf->set_mode(sca_util::sca_ac_format(sca_util::SCA_AC_MAG_RAD));

  // start frequency-domain simulation
  sca_ac_analysis::sca_ac_start(1.0e3, 100.0e4, 4, sca_ac_analysis::SCA_LOG);

  tf->reopen("ac_noise_trace.dat");
  tf->set_mode(sca_util::sca_noise_format(sca_util::SCA_NOISE_ALL));

  // start frequency-domain noise simulation
  sca_ac_analysis::sca_ac_noise_start(1.0e3, 100.0e4, 4, sca_ac_analysis::SCA_LOG);

  sca_util::sca_close_tabular_trace_file(tf);

  return 0;
}
```

More information on the simulation control and tracing capabilities can be found in Chapter 6.

This page is intentionally left blank.

6. Simulation and tracing

The AMS extensions use the SystemC functions to start and stop time-domain simulations. New functions are available for frequency-domain simulation. Advanced tracing mechanism are available to enable or disable time-domain or frequency-domain tracing while running simulations.

6.1. Simulation control

6.1.1. Time-domain simulation

Time-domain (transient) simulation is started by calling **sc_core::sc_start** from within the function **sc_main**, as shown in the example below.

```
#include <systemc-ams>

#include "my_source.h"
#include "my_control.h"
#include "my_dut.h"
#include "my_sink.h"

int sc_main(int argc, char* argv[])
{
  sc_core::sc_set_time_resolution(1.0, sc_core::SC_FS);

  sca_tdf::sca_signal<double> sig1, sig2;
  sc_core::sc_signal<bool> sc_sig;

  my_source i_my_source("i_my_source");
    i_my_source.out(sig1);

  my_control i_my_ctrl("i_my_ctrl");
    i_my_ctrl.out(sc_sig);

  my_dut i_my_dut("i_my_dut");
    i_my_dut.in(sig1);
    i_my_dut.ctrl(sc_sig);
    i_my_dut.out(sig2);

  my_sink i_my_sink("i_my_sink");
    i_my_sink.in(sig2);

  sc_core::sc_start(10.0, sc_core::SC_MS);

  return 0;
}
```

Program arguments

The function **sc_main** acts as main program, and has the same signature of arguments and return value as C++'s usual program entry function `int main(int argc, char* argv[])`. The argument `argc` specifies the number of arguments passed to the main routine. The argument `argv[]` is a field of pointers to the different string arguments.

Note that implementations or simulators, which support SystemC and the AMS extensions may use different mechanisms to define the main program body or even use an alternative approach to **sc_main**.

Time resolution

For AMS simulations, it is recommended to use the smallest time resolution possible covering the required simulation time using the function **sc_core::sc_set_time_resolution**. It is recommended to add this function as first statement in the **sc_main** function. For time-domain simulation, a time resolution of 1 femtosecond (fs) is recommended, which is the smallest time resolution possible allowing a maximum simulation time of 2^{64} fs, which is approximately 5 hours. In case longer simulation times are needed, the time resolution should be increased resulting in a coarser time grid and in possible rounding errors.

Simulation arguments

The function **sc_core::sc_start** without arguments will result in a simulation that runs until the last event has been processed, which might be forever. To simulate for a limited amount of time, the to-be-simulated-

time can be specified as a *double* value together with the time unit, or as an object of class **sc_core::sc_time**. The function **sc_core::sc_start** can be called multiple times, as shown in the example below:

```
int sc_main(int argc, char* argv[])
{
  // instantiate design and testbench, setup tracing, ...
  ...

  sc_core::sc_start(10.0, sc_core::SC_MS); ❶
  ...
  sc_core::sc_time sim_time(10.0, sc_core::SC_MS);
  sc_core::sc_start(sim_time); ❷
  ...
  sc_core::sc_start(); ❸

  return 0;
}
```

❶ Start transient analysis, where the simulation time is specified with two arguments. The first argument is the time of type *double*. The second argument is the time unit, which is an object of class **sc_core::sc_time_unit**.

❷ Start transient analysis, where the simulation time is specified with a single argument, which is an object of class **sc_core::sc_time**.

❸ In this case, no simulation time is specified. Transient analysis will run till the event queue is empty.

6.1.2. Small-signal frequency-domain simulation

Frequency domain simulations are also started from within the function **sc_main**, using **sca_ac_analysis::sca_ac_start** for a small-signal (AC) simulation and **sca_ac_analysis::sca_ac_noise_start** for a small-signal frequency-domain noise simulation. In the case that the model description has not been elaborated, because **sc_core::sc_start** has not yet been called, this will be automatically done before the first frequency-domain simulation starts.

It is possible to succeedingly call the frequency-domain and time-domain analyses start functions in any order inside the function **sc_main**, to analyze the system description under different operating points or digital states.

The example below shows the usage of the functions, which take as arguments the start frequency, stop frequency, number of frequency points, and whether a linear (**sca_ac_analysis::SCA_LIN**) or logarithmic (**sca_ac_analysis::SCA_LOG**) frequency scale should be used.

```
  // frequency-domain simulations from 1kHz to 10kHz with 100 points on a linear scale:
  sca_ac_analysis::sca_ac_start(1.0e3, 10.0e3, 100, sca_ac_analysis::SCA_LIN);
  sca_ac_analysis::sca_ac_noise_start(1.0e3, 10.0e3, 100, sca_ac_analysis::SCA_LIN);

  // frequency-domain simulations from 1Hz to 1MHz with 1001 points on a logarithmic scale:
  sca_ac_analysis::sca_ac_start(1.0, 1.0e6, 1001, sca_ac_analysis::SCA_LOG);
  sca_ac_analysis::sca_ac_noise_start(1.0, 1.0e6, 1001, sca_ac_analysis::SCA_LOG);
```

6.2. Tracing

The SystemC AMS extensions provide utility functions to record the simulation results (waveforms) into trace files, using the Value Change Dump (VCD) format or tabular format. The VCD format has limited capabilities to trace AMS signals, nodes, ports, terminals, or variables. Besides the tracing of regular SystemC variables and signals, it only supports tracing for time-domain simulations. The tabular format can be used to record both time-domain and frequency-domain traces.

The trace file is usually created at the top-level (e.g., inside **sc_main**) after all modules and signals have been instantiated, and just before starting the actual simulation using **sc_core::sc_start**, **sca_ac_analysis::sca_ac_start**, or **sca_ac_analysis::sca_ac_noise_start**.

6.2.1. Trace files and formats

Tracing to a VCD file

For tracing waveforms using the VCD format, the trace file is created by calling the function **sca_util::sca_create_vcd_trace_file** with the name of the file as an argument. This function returns a pointer to a data structure that is used for tracing. Closing the trace file is done using the function **sca_util::sca_close_vcd_trace_file** with as argument the pointer to the same data structure.

```
// open trace file in VCD format
sca_util::sca_trace_file* atf = sca_util::sca_create_vcd_trace_file( "my_trace.vcd" );

...

// close the trace file
sca_util::sca_close_vcd_trace_file( atf );
```

Note: the regular SystemC VCD tracing can be used to trace AMS signals, using functions **sc_core::sc_create_vcd_trace_file**, **sc_core::sc_trace** and **sc_core::sc_close_vcd_trace_file**, but in this case the AMS signals are translated and traced as discrete-event signals, using TDF converter output ports. As such, the synchronization aspects between the TDF and the discrete-event models of computation could play a role in the timing accuracy of the individual samples of these signals (see Section 2.4).

Tracing to a tabular file

For tracing waveforms using the tabular format, the trace file is created by calling the function **sca_util::sca_create_tabular_trace_file** with the name of the file as an argument. The function returns a pointer to a data structure that is used for tracing. Closing the trace file is done using the function **sca_util::sca_close_tabular_trace_file** with as argument the pointer to the same data structure, as shown in the example below.

```
// open trace file in tabular format
sca_util::sca_trace_file* atf = sca_util::sca_create_tabular_trace_file( "my_trace.dat" );

...

// close the trace file
sca_util::sca_close_tabular_trace_file( atf );
```

Tracing to a tabular stream

As tracing of analog signals could result in very big trace files, the AMS tracing functionality has been extended to trace to an output stream, so there is no file generated. This allows direct processing of the AMS signals available in the output stream derived from std::ostream, for example to immediately display the results or to compact the results into an archive file.

For tracing waveforms to an output stream, the trace file is created by calling the function **sca_util::sca_create_tabular_trace_file** with the output stream object as an argument. The function returns a pointer to an object of class **sca_util::sca_trace_file**, which references the stream and is used to manage the signal tracing to it. Closing the trace file is done using the function **sca_util::sca_close_tabular_trace_file** with as argument the pointer to the same output stream, as shown in the example below.

```
// trace in tabular format to the shell
sca_util::sca_trace_file* atfs = sca_util::sca_create_tabular_trace_file(std::cout);

...

// close the trace file handle, the stream is automatically closed once the scope of os is left.
sca_util::sca_close_tabular_trace_file(atfs);
```

Trace file control

As tracing of AMS signals could result in very large and unmanageable waveform files, additional functionality is available to control the recording of trace files. The following trace file control methods are available for class **sca_util::sca_trace_file**:

- The member function **enable** will start tracing at the simulation time where this method is called.
- The member function **disable** will stop tracing at the simulation time where this method is called.
- The member function **reopen** will close the existing trace file (if it was open), and will continue tracing in a new trace file at the simulation time where this method is called.
- The member function **set_mode** will change the mode of the trace file at the simulation time where this method is called. The following modi are available:
  - The time step (sampling) between samples can be set by using the function **sca_util::sca_sampling**, where the first argument is the time step and the second argument is the time offset. Both arguments should be an object of class **sca_core::sca_time**.
  - The function **sca_util::sca_decimation**, with argument *n*, will only write the *n*-th sample to the trace file.
  - The function **sca_util::sca_multirate** defines which signal value should be written to the trace file if no actual value is available. This can occur while tracing signals with different rates and time steps. Available arguments are to interpolate (**sca_util::SCA_INTERPOLATE**), to use the last available value (**sca_util::SCA_HOLD_SAMPLE**), or to not write a value at all (**sca_util::SCA_DONT_INTERPOLATE**). In the latter case, the symbol '*' is written to the trace file.
  - For small-signal frequency domain tracing, the function **sca_util::sca_ac_format** defines the format, in which the signals are written. Available function arguments are: real/ imaginary (**sca_util::SCA_AC_REAL_IMAG**) and amplitude/phase in magnitude/radians (**sca_util::SCA_AC_MAG_RAD**) or dB/degrees (**sca_util::SCA_AC_DB_DEG**).
  - For small-signal frequency domain tracing, the function **sca_noise_format** defines how the noise contribution is written to the trace file. If **sca_util::SCA_NOISE_ALL** is passed, each individual noise contribution is written to the trace file. If **sca_util::SCA_NOISE_SUM** is passed, the sum of all noise contributions is written to the trace file..

The following sections give some examples on how to use trace file control in combination with simulation control.

6.2.2. Tracing signals and comments

Supported AMS signals

The function **sca_util::sca_trace** is used to trace the actual AMS signals. The following elements can be traced:

- For TDF models, tracing is possible for TDF signals, TDF ports, and variables derived from class **sca_tdf::sca_trace_variable**.
- For LSF models, tracing is possible for LSF signals and LSF ports.
- For ELN models, voltage tracing is supported for nodes and terminals. Current tracing through ELN primitive modules having two terminals is supported. Some simulators also support current tracing through ELN primitive modules with more than two terminals.
- SystemC (discrete-event) signals and ports.

The example below shows how to use the function **sca_util::sca_trace** for the tracing of AMS signals of TDF, LSF or ELN models.

```
sca_util::sca_trace( atf, sig1, "sig1" );                    // trace TDF signal sig1
sca_util::sca_trace( atf, sig_de, "sig_de" );                // trace SystemC signal sig_de
sca_util::sca_trace( atf, my_source.out, "out1" );           // trace output of module my_source
sca_util::sca_trace( atf, my_source.i_sin_src->out, "out2" ); // trace output of nested module
sca_util::sca_trace( atf, my_sink.trv, "trv" );              // trace variable in module my_sink
```

Writing comments to a trace file

In order to write some user-specific comments or remarks in a tabular trace file, the function **sca_util::sca_write_comment** can be used, where the first argument is the pointer to the data structure of

the trace file and the second argument is the string containing the comment. The comment, including the preceding character '%', is added to the trace file at the simulation time where this function is called.

```
// open trace file in tabular format
sca_util::sca_trace_file* atf = sca_util::sca_create_tabular_trace_file( "my_trace.dat" );

...

// add comment to trace file
sca_util::sca_write_comment( atf, "user-defined comments" );

...

// close the trace file
sca_util::sca_close_tabular_trace_file( atf );
```

Note that adding user-specific comments could result in incompatibilities when using a specific waveform viewer, depending on file formats supported. It is recommended to check whether a particular waveform viewer supports a format which allows inclusion of user-specific comments.

Trace file example

This section shows some results of tracing time- and frequency signals, based on the following tracing definition in a **sc_main** program:

```
...
sca_util::sca_trace_file* tf = sca_util::sca_create_tabular_trace_file("trace.dat"); ❶

sca_util::sca_trace(tf, sig1, "sig1"); ❷
sca_util::sca_trace(tf, sig2, "sig2");

sc_core::sc_start(2.0, sc_core::SC_MS); ❸

tf->reopen("ac_trace.dat"); ❹

tf->set_mode(sca_util::sca_ac_format(sca_util::SCA_AC_MAG_RAD)); ❺

sca_ac_analysis::sca_ac_start(1.0e3, 1.0e6, 4, sca_ac_analysis::SCA_LOG); ❻

sca_util::sca_close_tabular_trace_file(tf); ❼
...
```

❶ Trace AMS signals to a file in tabular format using the tracing functionality of the AMS extensions.
❷ Define which signals to trace.
❸ Start time-domain simulation. Signals "sig1" and "sig2" will be traced.
❹ Close the current trace file and start tracing to a new file for frequency-domain analysis.
❺ Defintion to trace the amplitude and phase of the signals in magnitude and radians.
❻ Start frequency-domain simulation from 1kHz to 1MHz with 4 points on a logarithmic scale.
❼ Close the trace file.

The file `trace.dat` is shown below. The *%time* in the first line indicates that this file was created during time-domain simulation, and shows the signal names, which are traced. Each line shows the time in seconds and signal values at that point in time. The values are separated by one or more spaces.

```
%time sig1 sig2
0 0 0
0.0005 1 1e-6
0.001 2 1.5e-6
0.0015 3 2e-6
0.002 4 2.5e-5
```

The next example shows the result of the small-signal frequency-domain tracing in `ac_trace.dat`. The file starts with *%frequency* in the header. The format of the AC signals is set to amplitude (the magnitude) and phase (in radians) indicated with *.mag* and *.rad* suffixes to the signal names, respectively.

```
%frequency sig1.mag sig1.rad sig2.mag sig2.rad
1000  1 0  2.53302962314e-08 -3.14143349864
10000  1 0  2.53302959138e-10 -3.1415767381
```

```
100000  1 0  2.53302959106e-12 -3.14159106204
1000000  1 0  2.53302959106e-14 -3.14159249443
```

6.3. Testbenches

Testbenches are used to provide stimulus to a *device under test* (DUT) and check the results or response of the DUT. Very often the DUT is put into a certain state, using an external control. A typical testbench structure is given in Figure 6.1.

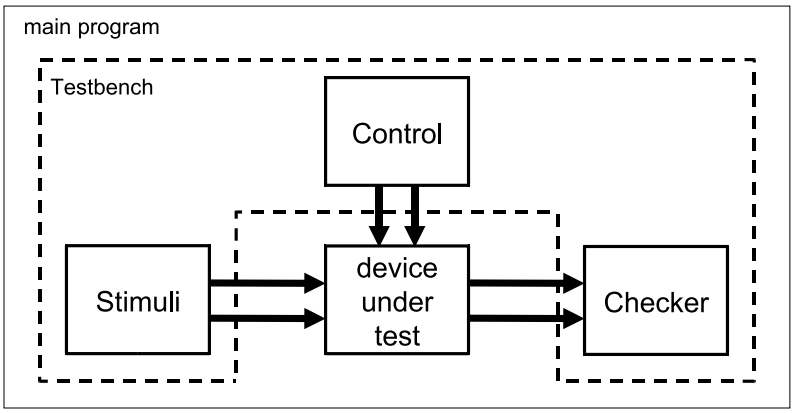


Figure 6.1. Testbench containing stimulus, control, checker, and device under test

A testbench can be implemented in various ways:

- The stimulus and controller can be embedded in the main program and the results is checked in another module. In this way, the main program acts as the testbench.
- The stimulus, controller, and checker are part of a dedicated module, which is instantiated in the main program. Such a module is often called a *verification component*, which basically acts as the testbench.
- The stimulus and controller are separate modules, both instantiated in the main program. The checker is embedded in the main program, which acts as the testbench.

Besides the examples listed above, there are other possibilities to create a testbench. Obviously, there is no single "right" way to create a testbench; it depends on the application.

The example below shows the main program in which the stimuli `my_source`, the control `my_control` and the sink `my_sink` are instantiated as objects. Together with the tracing implemented as inline code, they form the testbench. The device under test `my_dut` is instantiated as a module and is connected to the modules of the testbench.

```
#include <systemc-ams>

#include "my_source.h"
#include "my_control.h"
#include "my_dut.h"
#include "my_sink.h"

int sc_main(int argc, char* argv[])
{
  sc_core::sc_set_time_resolution(1.0, sc_core::SC_FS);

  sca_tdf::sca_signal<double> sig1, sig2;
  sc_core::sc_signal<bool> sc_sig;

  my_source i_my_source("i_my_source");
    i_my_source.out(sig1);

  my_control i_my_ctrl("i_my_ctrl");
    i_my_ctrl.out(sc_sig);

  my_dut i_my_dut("i_my_dut");
    i_my_dut.in(sig1);
    i_my_dut.ctrl(sc_sig);
```



```
    i_my_dut.out(sig2);

  my_sink i_my_sink("i_my_sink");
    i_my_sink.in(sig2);

  sc_core::sc_trace_file* tf = sc_core::sc_create_vcd_trace_file("my_sc_trace"); ❶
  sc_core::sc_trace(tf, sc_sig ,"sc_sig");

  sca_util::sca_trace_file* atf1 = sca_util::sca_create_vcd_trace_file("ams_vcd_trace.vcd"); ❷
  sca_util::sca_trace(atf1, sig1 ,"sig1");

  sca_util::sca_trace_file* atf2 = sca_util::sca_create_tabular_trace_file("ams_trace.dat"); ❸
  sca_util::sca_trace(atf2, sig2 ,"sig2");

  sc_core::sc_start(2.0, sc_core::SC_MS); ❹

  atf2->reopen("ams_trace.dat");  ❺
  sc_core::sc_start(2.0, sc_core::SC_MS);

  atf2->disable(); ❻
  sc_core::sc_start(2.0, sc_core::SC_MS);

  atf2->enable();  ❼
  atf2->set_mode( sca_util::sca_decimation(2) );
  sc_core::sc_start(2.0, sc_core::SC_MS);

  atf2->reopen("ams_trace3.dat"); ❽

  sca_core::sca_time tstep(1.0, sc_core::SC_MS); ❾
  atf2->set_mode( sca_util::sca_sampling( tstep, sc_core::SC_ZERO_TIME ) );
  sc_core::sc_start(10.0, sc_core::SC_MS);

  sc_core::sc_close_vcd_trace_file(tf); ❿
  sca_util::sca_close_vcd_trace_file(atf1);
  sca_util::sca_close_tabular_trace_file(atf2);

  return 0;
}
```

❶ Trace signals using SystemC's standard tracing facility. Be aware that in the case AMS (e.g., TDF) signals are traced, they are automatically converted to discrete event signals using TDF converter ports, which impacts the timing precision of the recorded samples.

❷ Trace AMS signals to a file in VCD format using the tracing functionality of the AMS extensions.

❸ Trace AMS signals to a file in tabular format using the tracing functionality of the AMS extensions.

❹ Start time-domain simulation. Signals "sig1" and "sig2" will be traced.

❺ Close the current trace file and start tracing to a new file (with same name).

❻ Disable tracing to atf2 to not record the next 2 ms.

❼ Re-enable tracing to atf2, but with a different sample period defined by a decimation factor of 2 (skip one sample).

❽ Close the current trace file of atf2 and start tracing to a new file using a different time step.

❾ Define how samples are written to the trace file. Sample every 1 ms starting from 0 ms.

❿ Close all trace files.

This page is intentionally left blank.

7. Modeling strategies

The SystemC AMS extensions provide designers with powerful tools for modeling analog/mixed-signal systems. The extensions cover the use cases described in Chapter 1, by providing the models of computation Timed Data Flow, Linear Signal Flow, and Electrical Linear Networks, in addition to the discrete-event and Transaction Level Modeling approaches of SystemC. This chapter gives additional advice on how to use and combine these models of computation in an efficient way. The presented strategies are not mandatory, and sometimes there might be other or better approaches. They are provided in order to guide an inexperienced user and help him to create his first models, to achieve high simulation performance, and to increase productivity when designing mixed analog/digital systems.

7.1. Behavioral modeling using the available models of computation

The models of computation provided by the SystemC AMS extensions are primarily intended to facilitate the *behavioral modeling* of analog circuits, as well as of signal processing functions (no matter whether they will be implemented in the analog or digital domain). Depending on the abstraction required, a suitable model of computation for behavioral modeling has to be selected. Figure 7.1 gives an overview of the available models of computation and the abstractions imposed by them, considering the aspects behavior, structure, communication, and time/frequency.

Model of Computation	Imposed abstractions			
	Behavior	Structure	Communication	Time/Frequency
Timed Data Flow (TDF)	Algorithmic descriptions, transfer functions	Functional blocks (non-conservative system)	Sequence of samples of arbitrary type	(Over)sampling, baseband modeling
Linear Signal Flow (LSF)	Linear functional descriptions	Structural representation of linear equations (non-conservative system)	Directed signals, continuous value	No abstraction (continuous time)
Electrical Linear Networks (ELN)	Macro modeling with linear primitives and ideal switches	Simplified network (conservative system)	No abstraction (physical quantities)	No abstraction (continuous time)

Figure 7.1. Abstractions imposed by the AMS models of computation

The most important abstractions imposed by the models of computation are:

- *Linearization* of non-linear behavior due to the focus on *linear* behavior in the models of computation that require the solving of equation systems (LSF, ELN).
- *Abstraction to functional blocks* (non-conservative systems) with *directed signals* in the models of computation LSF and TDF. This abstraction replaces the physical quantities ($i(t)$, $u(t)$) with abstract quantities $x(t)$.
- *Sampling* of continuous-time signals to discrete-time signals for the TDF model of computation.

Figure 7.2 shows the impact of abstraction and sampling to non-conservative behavior of a signal in an electrical network.

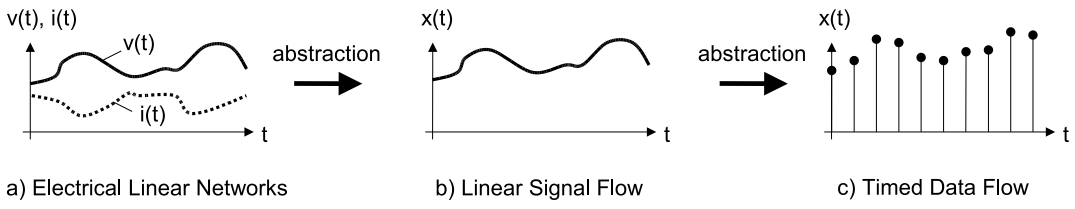


Figure 7.2. Abstraction of communication and time using the AMS models of computation

In the following subsections, a brief and general description of the capabilities of each model of computation is given. When using multiple models of computation, it is important to combine them appropriately. The required partitioning of functionality onto different models of computation is discussed in Section 7.2.1.

7.1.1. Macromodeling with Electrical Linear Networks

The ELN model of computation permits *macromodeling*: Accurate physical devices such as transistors are modeled by simple electrical primitives such as (ideal) switches, voltage sources, and other electrical linear primitives. The objective is to specify an abstract model with simplified behavior. Considering signals and interfaces, no abstraction is required. The ELN model of computation should be used in the following cases:

1. Description of systems where it is not easy or natural to give equations, e.g., transmission-line models, or nearly linear loads that are switched within a duty cycle.
2. Analog interfaces and components, which are relevant for the dimensioning of the system or its overall behavior. Therefore, they must show up at the system level.

In order to setup an ELN macromodel, the electrical circuit behavior must be linearized. The availability of switches in addition to linear components enables to handle the switching between different operating modes or the on/off switching of loads. The following strategy might be useful to get an ELN model from a given circuit:

1. Identify partitions of the circuit with nearly linear behavior, and model them using ELN components. Components that are not required for the overall functionality (e.g. clamping diodes) can be omitted.
2. Identify switching components and replace them with ideal switches.
3. Depending on the intended environment of the model:
   - If applied as part of ELN, model input and output impedances.
   - If applied as part of TDF or discrete-event, use appropriate converter elements.

Note that the ELN model of computation does not support modeling of non-linear limitation or saturation effects. It is recommended to partition a model such that non-linear effects are modeled using the TDF model of computation.

Figure 7.3 shows an example of a power driver using Pulse Width Modulation (PWM). The original circuit is shown in Figure 7.3a. In order to apply ELN macromodeling, the clamping diodes are omitted assuming that the circuit itself has been validated using a circuit simulator. The CMOS transistors that are switching the load, a coil with 10 Ohm resistance, are replaced with ideal switches. The resulting ELN macromodel is shown in Figure 7.3b.

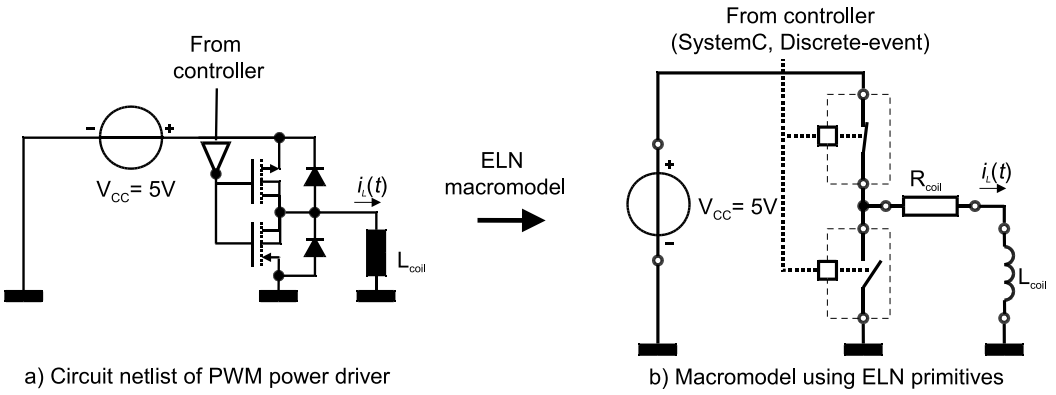


Figure 7.3. Abstraction of PWM power driver into an ELN macromodel

The PWM driver together with its load has the behavior of a low pass filter for the load current $I_L(t)$. As such, it could also be modeled as a functional block. However, the load itself is usually an external part,

and thus might be changed by the user. Therefore, it makes sense to provide an electrical terminal, and a (linear) macromodel of the load. The next code example shows the ELN model of the PWM driver.

```
SC_MODULE(pwm_driver)
{
  sc_core::sc_in<bool> in;
  sca_eln::sca_terminal  out;

  sca_eln::sca_vsource vcc;  // voltage source
  sca_eln::sca_de::sca_rswitch highside, lowside;  // two switches

  pwm_driver( sc_core::sc_module_name nm, double vcc_ = 5.0)
  : in("in"), out("out"),
    vcc("vcc"), highside("highside"), lowside("lowside"), node("node"), gnd("gnd")
  {
    vcc.offset = vcc_;  // usage as constant voltage source
    vcc.p(node);
    vcc.n(gnd);

    highside.ctrl(in);  // 1st switch
    highside.p(node);
    highside.n(out);

    lowside.ctrl(in);   // 2nd switch...
    lowside.p(out);
    lowside.n(gnd);
    lowside.off_state = true; // ...is inverted
  }

 private:
  sca_eln::sca_node node;
  sca_eln::sca_node_ref gnd;
};
```

The load can as well be described easily using linear primitives, in the most simple case, a coil with some resistance might be sufficient:

```
SC_MODULE(load)
{
  sca_eln::sca_terminal p, n;

  sca_eln::sca_r r;
  sca_eln::sca_l l;

  load( sc_core::sc_module_name nm,
        double res_ = 500.0, double ind_ = 0.000001 )
  : p("p"), n("n"), r("r", res_ ), l("l", ind_ ), node("node")
  {
    r.p(p);
    r.n(node);

    l.p(node);
    l.n(n);
  }

 private:
  sca_eln::sca_node node;
};
```

7.1.2. Behavioral modeling with Linear Signal Flow

The LSF model of computation permits the description of *block diagrams* for the computation of linear differential equations. Compared to transfer functions, LSF allows to specify the order of computations and to access intermediate results or coefficients. In particular, LSF is useful to:

1. Model filters with a given structure that has, e.g., impact on noise.
2. Model continuous-time control systems, in particular those that require access to coefficients from other models of computation.

For LSF, an abstraction of physical signals is required as described in Chapter 3. Most notably, this also requires the abstraction of communication towards directed signals. Considering the structure and behavior, functional blocks have to be identified, and their behavior has to be described by instantiating the pre-defined functional primitives. Considering time, no abstraction is required.

Note that LSF does not provide means to specify non-linear limitation or saturation. It is recommended to partition a model in a way that non-linear effects, if needed, are specified using the TDF model of computation. A typical application example where LSF is useful is shown by Figure 7.4. It is a PID controller that can be part of a closed loop control system model. Its coefficients can be adjusted from a TDF model. In order to model a closed-loop control system without delay, the device itself must also be modeled using the LSF model of computation. Using any other model of computation (ELN, TDF) will introduce a delay in the control loop.

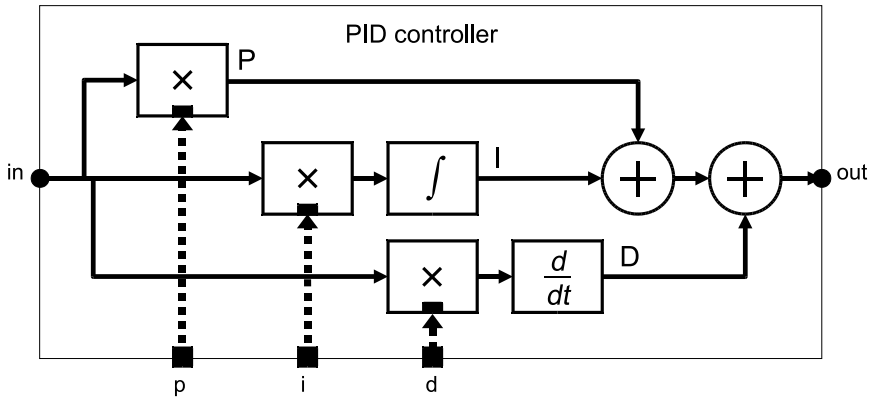


Figure 7.4. LSF model of a PID controller with adjustable coefficients

```
SC_MODULE(lsf_pid_external_control)
{
  sca_lsf::sca_in  in;
  sca_lsf::sca_out out;

  sca_tdf::sca_in<double> p, i, d; // adjustable coefficients

  sca_lsf::sca_tdf::sca_gain  gain_p, gain_i, gain_d; // coefficients used to scale the gain
  sca_lsf::sca_integ integ;
  sca_lsf::sca_dot   dot;
  sca_lsf::sca_add   add1, add2;

  lsf_pid_external_control( sc_core::sc_module_name name )
  : in("in"), out("out"), p("p"), i("i"), d("d"),
    gain_p("gain_p"), gain_i("gain_i"), gain_d("gain_d"),
    integ("integ"), dot("dot"), add1("add1"), add2("add2"),
    sig_gain("sig_gain"), sig_integ1("sig_integ1"), sig_integ2("sig_integ2"),
    sig_dot1("sig_dot1"), sig_dot2("sig_dot2"),
    sig_add("sig_add")
  {
    gain_p.x(in);
    gain_p.y(sig_gain);
    gain_p.inp(p);

    gain_i.x(in);
    gain_i.y(sig_integ1);
    gain_i.inp(i);

    gain_d.x(in);
    gain_d.y(sig_dot1);
    gain_d.inp(d);

    integ.x(sig_integ1);
    integ.y(sig_integ2);

    dot.x(sig_dot1);
    dot.y(sig_dot2);

    add1.x1(sig_gain);
    add1.x2(sig_integ2);
    add1.y(sig_add);

    add2.x1(sig_add);
    add2.x2(sig_dot2);
    add2.y(out);
  }
```



```
 private:
  sca_lsf::sca_signal sig_gain, sig_integ1, sig_integ2, sig_dot1, sig_dot2, sig_add;
};
```

7.1.3. Behavioral and baseband modeling with Timed Data Flow

The TDF model of computation permits the modeling of analog systems at a high level of abstraction, as well as the modeling of signal processing functions.

For modeling analog behavior, the TDF model of computation requires a discrete-time approximation of the continuous-time analog signals. However, TDF permits, in contrast to LSF and ELN, the modeling of *non-linear* behavior. The discrete-time approximation reduces the continuous-time signal to a sequence of discrete samples. This abstraction avoids the need for solving (non-linear) equations and thus improves simulation performance. Besides the discretization, the TDF model of computation also requires the breaking of cyclic dependencies (also known as algebraic loops) by inserting delays (see Section 2.1.2).

In exchange for these abstractions, TDF models permit to describe the processing of streams of samples in an arbitrary, algorithmic way with the help of the member function **processing**. In particular, also non-linear transfer functions (i.e., for modeling limitation) or look-up tables can be implemented easily. Furthermore, the specification of signal processing methods in terms of transfer functions $H(s)$, $H(z)$, or state space representations is supported in TDF (see Section 2.3.2).

The following abstractions are introduced by TDF:

1. Like in LSF, a *block diagram* structure has to be defined. Unlike in LSF, there is virtually no restriction to the behavior of single blocks.
2. The *sampling frequency* must be defined.
3. The TDF model requires *acyclic structures* to ensure schedulability. The acyclic structure can be achieved by introducing a delay into the cycle (Section 2.1.2). Note that most control loops use nowadays digital controllers that anyhow introduce delays. The location of the digital controller might be a good location for introducing such a delay.

Definition and propagation of time steps and rates

The time steps and rates in TDF must be selected carefully to match the modeling problem. It is also recommended to carefully select the places where time steps and rates are defined.

For modeling *analog behavior*, is is recommended to ensure a sufficiently high sample frequency. The sampling frequency must be significantly higher than twice the frequency defined by the lowest time constant in the system. In doubt, a factor 10 is recommended. Selecting a higher rate or smaller time steps results in a higher accuracy at high frequencies at the cost of simulation performance. An appropriate place to define the time step might be the test bench.

Systems with time constants that differ by orders of magnitudes (stiff systems) are a particular problem. We recommend to partition such systems into parts with low time constants, and parts with higher time constants. Then, different *rates* of the TDF model of computation can be used to define different sample frequencies in each partition.

For modeling *digital signal processing (DSP) methods* (e.g. using $H(z)$, or state space representations of digital filters) leads to a dependency between functionality and the selected time step. For DSP methods that are intended for use at a particular sample frequency, it is recommended to define a time step in the module itself (or at its ports respectively). Note that a test bench still can define time steps. However, an error will be reported if the consistency check after propagation of time steps fails (see Section 2.5).

Behavioral modeling with TDF

Section 2.6 gives two application examples introducing behavioral modeling using the TDF model of computation. Note that the SystemC AMS extensions permit to write arbitrary C++ code into the

TDF module member function **processing**. This allows combining ideal signal processing functions (usually found in block libraries) such as amplification, multiplication, or transfer functions in a very easy and effective way with non-ideal behavior. An amplifier, for example, can be modeled by combining the following features:

1. Its behavior in the frequency domain can be modeled using a Laplace transfer function, as discussed in Section 2.3.2. Poles and zeros can be identified easily using circuit simulation, or using the bode plot.

2. Large-signal behavior (e.g., limitation, non-linearity) can be modeled by using C++ code.

```
SCA_TDF_MODULE(amplifier)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;

  amplifier( sc_core::sc_module_name, double gain_     = 100.0,
                                      double dom_pole_ = 5.0e8,
                              double limit_    = 5.0 )
  : in("in"), out("out"), gain(gain_), dom_pole(dom_pole_), limit(limit_) {}

  void initialize()
  {
    // filter requires no zeros to be defined
    poles(0) = sca_util::sca_complex( -2.0 * M_PI * dom_pole, 0.0 );
    k = gain * 2.0 * M_PI * dom_pole;
  }

  void processing()
  {
    // time-domain implementation of amplifier behavior as function of frequency
    double internal = ltf_zp( zeros, poles, state, in.read(), k );

    // limiting the signal
    if (internal > limit) internal = limit;
    else if (internal < -limit) internal = -limit;

    out.write(internal);
  }

 private:
  double gain;     // DC gain
  double dom_pole; // 3dB cutoff frequency in Hz
  double limit;    // limiter value
  double k;                   // filter gain
  sca_tdf::sca_ltf_zp ltf_zp; // Laplace transfer function
  sca_util::sca_vector<sca_util::sca_complex > poles, zeros; // poles and zeros as complex values
  sca_util::sca_vector<double> state; // state vector
};
```

Baseband modeling with TDF

When modeling *radio frequency* (RF) systems with high carrier frequencies, a significant speed-up of simulation can be achieved by applying baseband modeling. This modeling strategy is based on the fact that digital modulation techniques use the amplitude r and the phase ϕ to transmit information. The information itself is then independent from the (usually high) carrier frequency. The idea of baseband modeling is to map the RF carrier frequency to zero, as shown in Figure 7.5. The required sampling rate then only depends on the bandwidth of the modulated signal.

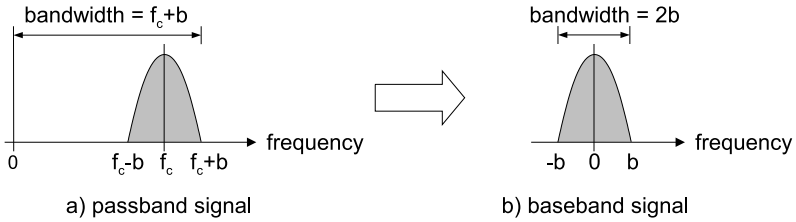


Figure 7.5. Passband (a) and baseband (b) representation of signals in the frequency domain

Formally, the modulated carrier signal $x(t)$ can be described as:

$$\begin{aligned} x(t) &= r(t)\cos(2\pi f_c t + \varphi(t)) \\ &= \mathrm{Re}\{r(t)\, e^{j\,(2\pi f_c t + \varphi(t))}\} \\ &= \mathrm{Re}\{r(t)\, e^{j\,\varphi(t)} e^{j\,2\pi f_c t}\} \end{aligned}$$

where $r(t)$ is the modulating signal, $\phi(t)$ the modulating phase and f_c the carrier frequency. The term, which includes the carrier frequency f_c, can be separated from the signal part, which contains the transmitted information. The signal $v(t)$, which contains the information, is independent from the carrier frequency f_c:

$$v(t) = r(t)\, e^{j\,\varphi(t)}$$

This signal is called the *complex low-pass equivalent* or the *complex envelope*. For the baseband signal, the carrier frequency f_c is set to zero. With $s_i = r\cos(\phi)$ and $s_q = r\sin(\phi)$, the resulting baseband signal becomes:

$$v(t) = s_i(t) + j\, s_q(t)$$

where s_i represents the in-phase term of the baseband signal, and s_q represents the quadrature term. The amplitude and phase of the carrier signal can be computed from these signals at each point in time.

In order to make use of these baseband signals, a specialized data type is needed, which supports the definition of complex values. The SystemC AMS extensions offer the class **sca_util::sca_complex**, which can be used for this purpose. These signals can be used in TDF modules, in which the value type of the TDF ports are changed from scalar values (of type double) to complex values, as shown in the example below. The function `std::pow(c,y)` from the C++ standard library `complex` is used, which computes *c* raised to the power of *y*, where *c* is a complex value.

```
#include <complex>

SCA_TDF_MODULE(baseband_amplifier)
{
  sca_tdf::sca_in< sca_util::sca_complex >  in;
  sca_tdf::sca_out< sca_util::sca_complex > out;

  baseband_amplifier( sc_core::sc_module_name, double gain = 1.0, double iip3 = 1e-3 )
  : in("in"), out("out"), a1( gain ), a3( -4/3 * ( gain / std::pow(iip3,2)) ) {}

  void processing()
  {
    out.write( a1 * in.read() + a3 * std::pow( in.read(),3 ) );
  }

 private:
  double a1, a3;
};
```

The limitation of using **sca_util::sca_complex** as the data type is that it only describes the complex envelope of the modulated signal, and that the carrier frequency information is lost. Due to this, effects like harmonics of the carrier or intermodulation products are not represented, as they fall outside the signal bandwidth. The solution to this is to create a user-defined data type similar to **sca_util::sca_complex**, which supports multi-carrier baseband computations.

7.2. Modeling embedded analog/mixed-signal systems

Behavioral modeling using a single model of computation imposes a number of restrictions as shown in Figure 7.1. They can be overcome by combining (the strengths of) different models of computation. The following subsections describe how to partition the functional behavior onto the different models of computation. Then, a number of simple modeling guidelines is given, how to model architecture-level properties of analog circuits.

7.2.1. Partitioning behavior to different models of computation

A simple, but general strategy that allows beginners to distribute a block diagram like specification to the different models of computation provided by the SystemC AMS extensions is shown by Figure 7.6. It can be applied for each block successively.

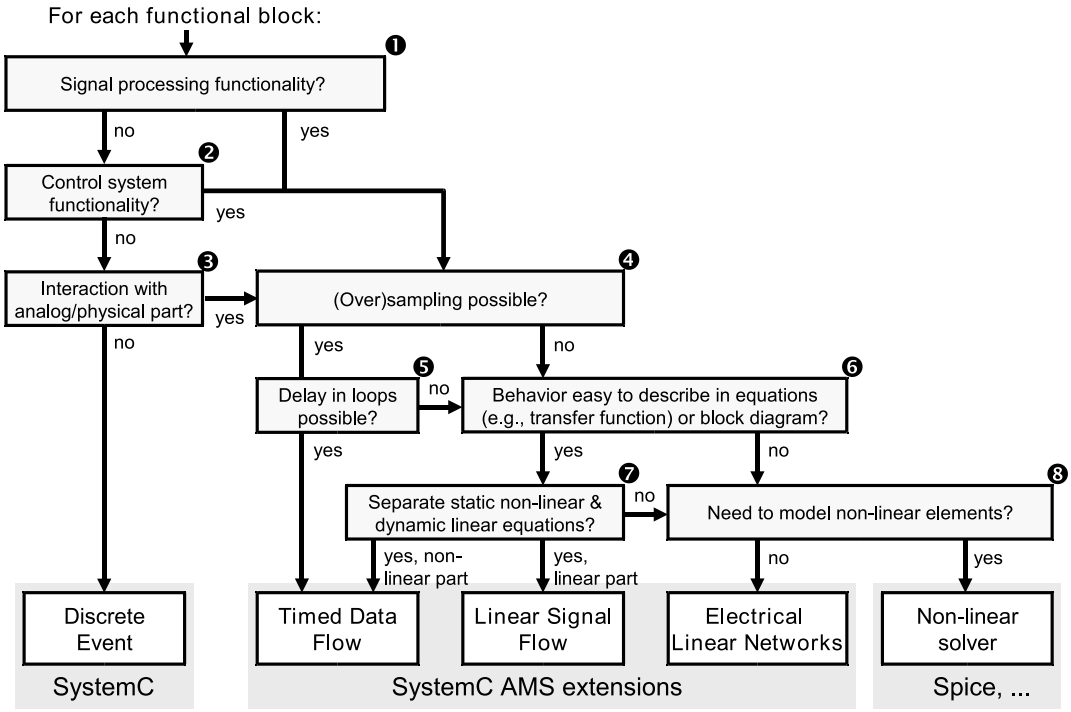


Figure 7.6. Partitioning of behavior to different models of computation

In a first step (labels 1, 2 and 3), it should be investigated whether the discrete-event model of computation is appropriate, or whether the AMS extensions are the better choice. The SystemC AMS extensions are the best choice for modeling signal processing functions. Note that signal processing functions that are implemented in digital or software can also be modeled efficiently using the SystemC AMS extensions at functional level. If a concrete mapping to hardware at architecture level or below shall be modeled, SystemC is more appropriate. Another good reason to use the AMS extensions would be the need for having analog terminals and/or physical quantities such as current available, e.g., for modeling external loads or analog behavior of communication lines.

In a second step (labels 4 and 5), the Timed Data Flow model of computation should be considered to model the AMS subsystem. It requires discrete-time modeling of analog signals, and (in case of cyclic dependencies) the insertion of additional delays. It offers most options for specification of analog and signal processing behavior.

If a discrete approximation is not appropriate (labels 6, 7, and 8), one has to consider the models of computation LSF and ELN. Both rely on a *linear* solver. Therefore, behavior has to be partitioned into linear and non-linear functionality, where the latter can be implemented using TDF. If the accurate modeling of non-linear conservative behavior or electrical networks is required, one should consider using an appropriate non-linear solver or circuit simulator, maybe coupled with SystemC.

7.2.2. Modeling of architecture-level properties

In order to evaluate feasibility and performance of different architectures, the functional model can be used and refined by adding specific properties. These properties include: noise, attenuation, distortions, limitation, jitter, delays, quantization, sampling frequencies, and many other. In the following, some simple guidelines for handling these effects during architecture exploration are given.

Modeling distortions, limitation, and quantization

In order to study the impact of distortions and limitations on the overall system functionality, analog modules should be split into linear dynamic behavior and nonlinear static behavior. Linear dynamic behavior can be specified, e.g., using transfer functions in TDF (see Section 2.3.2). Non-linear behavior such as distortions and limitation can be modeled easily using C++ functions in the TDF module's member function **processing** (see Section 7.1.3).

Modeling noise in time domain

Noise in the TDF model of computation can be modeled by adding gaussian distributed random numbers to a TDF signal. The following example demonstrates a simple model for (white) noise and attenuation in a wireless communication link. For this purpose, a function `gauss_rand` is used that generates gaussian distributed random numbers.

```
// the gauss_rand() function returns a gaussian distributed
// random number with variance "variance", centered around 0, using the Marsaglia polar method

#include <cstdlib> // for std::rand
#include <cmath>   // for std::sqrt and std::log

double gauss_rand(double variance)
{
  double rnd1, rnd2, Q, Q1, Q2;

  do
  {
    rnd1 = ((double)std::rand()) / ((double)RAND_MAX) ;
    rnd2 = ((double)std::rand()) / ((double)RAND_MAX) ;

    Q1 = 2.0 * rnd1 - 1.0 ;
    Q2 = 2.0 * rnd2 - 1.0 ;
    Q = Q1 * Q1 + Q2 * Q2 ;

  } while (Q > 1.0) ;

  return ( std::sqrt(variance) *( std::sqrt( - 2.0 * std::log(Q) / Q) * Q1) );
}

SCA_TDF_MODULE(air_channel_with_noise)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;

  void processing()
  {
    out.write( in.read() * attenuation + gauss_rand(variance) );
  }

  air_channel_with_noise( sc_core::sc_module_name nm,
                          double attenuation_,
                          double variance_ )
  : in("in"), out("out"), attenuation(attenuation_), variance(variance_) {}

 private:
  double attenuation;
  double variance;
};
```

In order to get colored noise, the output of the function `gauss_rand` can be filtered using appropriate transfer functions.

7.3. Design refinement and mixed-level modeling

7.3.1. Mixed-signal, mixed-level simulation

The design of embedded analog/digital systems requires the combination of different models of computation and of different levels of abstraction. This requires the conversion of communication/synchronization at the border between different models of computation. The SystemC AMS extensions provide a basic set of language primitives that enable conversion between SystemC (discrete-event), TDF, ELN, and LSF. In ELN and LSF, converter modules are provided; in TDF, converter ports are available. Note that ELN and LSF can communicate with discrete-event and TDF, but not with each other in a direct way.

It is recommended to model the general signal flow of a system using the TDF model of computation, if possible. This has the following advantages:

1. The TDF model of computation provides conversion to all other models of computation.

2. The TDF model of computation is needed to provide time steps to connected ELN and LSF components.

Figure 7.7 shows a part of a signal processing chain as an example: The LSF controller (shown left) feeds its output via a controlled voltage source into an ELN low-pass filter. In order to connect ELN and LSF, an LSF signal is converted to a TDF signal, for which a time step must be given. The TDF signal controls a (TDF) controlled voltage source that is part of the ELN model.

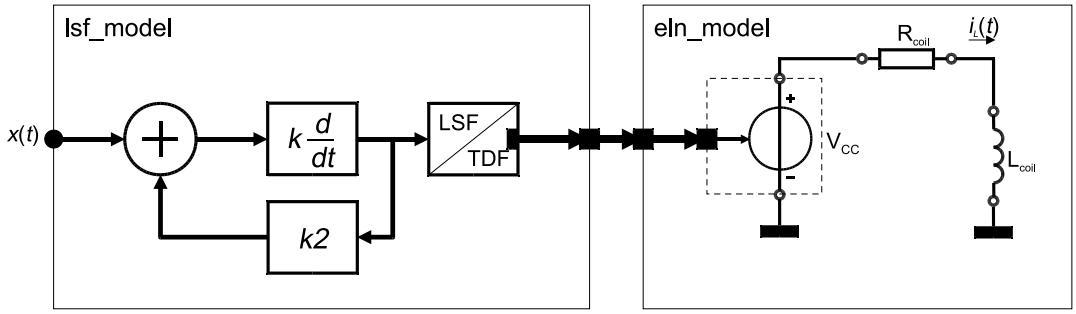


Figure 7.7. Coupling of LSF and ELN via an LSF/TDF converter module

Be aware, that the conversion from LSF (or ELN) to TDF and then to ELN (or LSF) introduces a delay of one time step.

7.3.2. Design refinement and use cases

For the design of digital systems, top-down design is state-of-the-art. The integration of analog/mixed-signal subsystems, which are mostly designed bottom-up, into a digitally dominated top-down flow is still a challenge. In Section 1.2.1, the intended use cases of the AMS extensions have been introduced. This section describes how to apply the SystemC AMS extensions in order to yield higher efficiency and productivity in the design process of embedded analog/digital systems. This complements the refinement approach known from SystemC. Figure 7.8 gives an overview of the application of the SystemC AMS extensions.

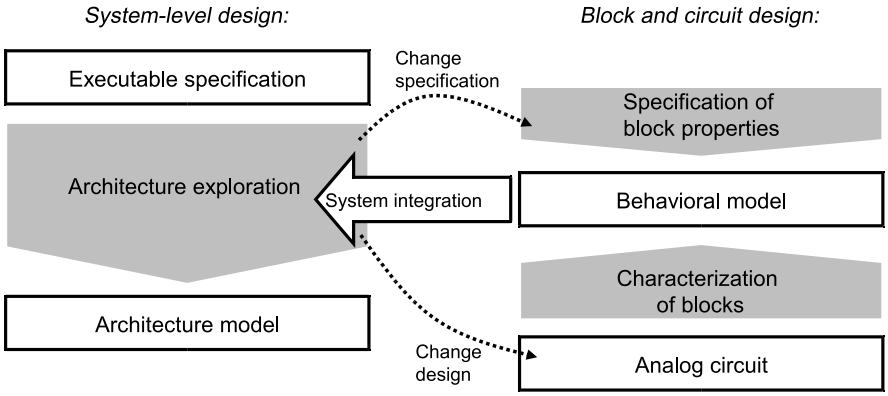


Figure 7.8. Use cases for the SystemC AMS extensions within top-down refinement

In the ideal case, top-down refinement begins with an executable specification of the intended behavior at system level. Usually, the TDF model of computation is suitable to develop a functional model for this purpose. Refinement of the executable specification is part of the architecture exploration use case. The refinement process consists of a stepwise approach of replacing the blocks in the system with more accurate (less abstract) models.

Architecture exploration distinguishes three separate aspects, each one being the opposite of one of the abstractions in Figure 7.1:

- Refinement of behavior
- Refinement of structure
- Refinement of communication/interfaces

Behavioral refinement augments the functional model used for the executable specification with specific properties of an architecture (implementation). This permits the evaluation of the feasibility and performance of different architectures (implementations). Properties that can easily be included in a functional model include: Noise, attenuation, distortions, limitation, jitter, delays, quantization, sampling frequencies, and many other.

As an example, Figure 7.9 shows an ideal (linear) and non-linear amplifier, where the linear gain (a1) and non-linear term (a3) are added with a polynomial representation.

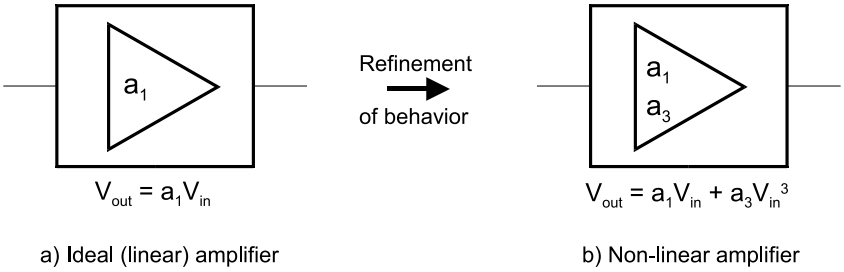


Figure 7.9. Refinement of behavior of an amplifier

The code example below shows how the non-linear behavior can be implemented. The function `std::pow(x,y)` from the C++ standard library `cmath` is used, which computes *x* raised to the power of *y*.

```
#include <cmath>

SCA_TDF_MODULE(non_linear_amplifier)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;

  non_linear_amplifier( sc_core::sc_module_name, double gain = 1.0, double iip3 = 1e-3 )
  : in("in"), out("out"), a1( gain ), a3( -4/3 * ( gain / std::pow(iip3,2)) ) {}

  void processing()
  {
    out.write( a1 * in.read() + a3 * std::pow(in.read(),3) );
  }

 private:
   double a1, a3;
};
```

Refinement of structure repartitions the (usually block-diagram-like) system, used for executable specification, with a structure of functional blocks that each represent a circuit or processor to be designed. Note that also the model of computation changes depending on the intended domain of implementation.

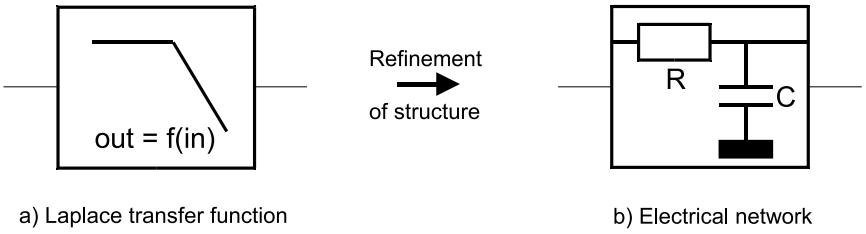


Figure 7.10. Structural refinement of a filter

In order to make the refinement of a model easier, the namespace concept allows to re-use a large part of existing modeling infrastructure such as module and port declarations. However, behavior and (refined) structure have to be written from scratch.

Refinement of communication/interfaces replaces the abstract communication used within the TDF model of computation with concrete signals, e.g., electrical voltages and currents or digital (discrete-event) SystemC signals. This requires to also add converter ports or modules to the models. Conversion between the models

of computation is discussed in Section 7.3.1. In order to support the refinement of communication/interfaces, it is recommended to create adapter/converter classes as known from SystemC TLM extensions.

7.4. Modeling and coding style

7.4.1. Namespaces

The SystemC AMS extensions make extensive use of C++ namespaces to be able to clearly identify the available models of computation and use the available primitive modules within the right context. The namespaces **sca_tdf**, **sca_lsf** and **sca_eln** are reserved names for the language constructs used for the TDF, LSF and ELN model of computation, respectively. Other reserved namespaces are **sca_util** for utility classes and functions, and **sca_ac_analysis** for small-signal frequency-domain analyses. The user should not add new definitions in these namespaces. Instead, it is recommended to declare user-defined modules belonging to the same model of computation to a unique user-defined namespace, as shown in the example below.

```
namespace my_tdf {

  SCA_TDF_MODULE(my_source)
  {
    ...
  }

}; // namespace my_tdf
```

Instantiation of this object will look like this:

```
SC_MODULE(analog_top)
{
  ...
  my_tdf::my_source i_my_source("i_my_source");
  ...
}
```

Header files and naming conventions

The header file **<systemc-ams>** does not import the reserved namespaces **sca_tdf**, **sca_lsf**, **sca_eln**, **sca_util**, and **sca_ac_analysis** into the scope of the program. This means the user has to explicitly add the namespace identifier to each element, when instantiating or declaring such an object. Although the names are a bit longer to write, it will result in a clear naming convention, where the user can recognize immediately whether the object belongs to a particular class library of the SystemC AMS extensions, or whether the object is part of a user-defined library. The example below, and the previous examples given in this user's guide follow this naming convention.

```
#include <systemc-ams>

#include "my_source.h"

int sc_main(int argc, char* argv[])
{
  sc_core::sc_set_time_resolution(1.0, sc_core::SC_FS);

  sca_tdf::sca_signal<double> sig1;

  // instantiate user-defined module from user-defined 'my_tdf' namespace
  my_tdf::my_source i_my_source("i_my_source");
    i_my_source.out(sig1);

  // instantiate other modules
  ...

  // tracing AMS signals
  sca_util::sca_trace_file* tf = sca_util::sca_create_tabular_trace_file("trace.dat");
  sca_util::sca_trace(tf, sig1 ,"sig1");

  sc_core::sc_start(10.0, sc_core::SC_MS);

  tf->set_mode(sca_util::sca_ac_format(sca_util::SCA_AC_MAG_RAD));

  sca_ac_analysis::sca_ac_start(1.0e3, 1.0e6, 4, sca_ac_analysis::SCA_LOG);
```



```
  sca_util::sca_close_tabular_trace_file(tf);

  return 0;
}
```

When using the header file **<systemc-ams.h>**, all elements, which belong to the namespace **sca_core**, **sca_util** and **sca_ac_analysis**, are imported into the scope of the program. This means the user can omit to prefix the elements in these namespaces. Note that the namespace for the different models of computation are not declared, so even in this case, the user has to explicitly use the namespace to create TDF, LSF, and ELN models. The program below shows the same example as given above, but now using the header file **<systemc-ams.h>**.

```
#include <systemc-ams.h>

#include "my_source.h"

int sc_main(int argc, char* argv[])
{
  sc_set_time_resolution(1.0, sc_core::SC_FS);

  sca_tdf::sca_signal<double> sig1;

  // instantiate user-defined module from user-defined 'my_tdf' namespace
  my_tdf::my_source i_my_source("i_my_source");
    i_my_source.out(sig1);

  // instantiate other modules
  ...

  // tracing AMS signals
  sca_trace_file* tf = sca_create_tabular_trace_file("trace.dat");
  sca_trace(tf, sig1 ,"sig1");

  sc_start(10.0, SC_MS);

  tf->reopen("ac_trace.dat");

  tf->set_mode( sca_ac_format(SCA_AC_MAG_RAD) );

  sca_ac_start(1.0e3, 1.0e6, 4, SCA_LOG);

  sca_close_tabular_trace_file(tf);

  return 0;
}
```

It is recommended to use the header file **<systemc-ams>**, resulting in a naming convention reflecting the full names of classes and functions.

Using directive

The *using directive* of C++ allows the elements in a namespace to be used without explicitly adding the namespace identifier to each element. It should only be used in a module implementation, not in the module declaration (e.g. definition in a header file). It is recommended to apply the using directive only within the local scope, e.g., as part of the implementation of a class member function. The example below shows how this concept can be applied for a frequency-domain description as described in Section 5.3.3.

```
  void ac_processing()
  {
    using namespace sca_util;
    using namespace sca_ac_analysis;

    sca_complex s = SCA_COMPLEX_J * sca_ac_w();
    sca_complex h = 1.0 / ( s * s + s + 1.0 );

    sca_ac(out) = h * sca_ac(in);
  }
```

7.4.2. Dynamic memory allocation

Most of the examples shown in this user's guide use objects (e.g., primitive modules), which are directly instantiated in a function body, and thus are allocated automatically on the stack. In case of big designs

using many modules in a complex hierarchy, this approach is not the most efficient way as it can lead to an overflow of the stack for automatic variables. Dynamic memory allocation has the advantage to give the user more direct control, in which order the modules are constructed. The instantiated objects are referenced by pointers so that they do not need to reside anymore in a consecutive memory area, which can lead to resource allocation problems. Furthermore, it allows the instantiation of an arbitrary number of modules determined at runtime, which are referenced from a dynamically created array of module pointers, and which constructors can be called individually to vary the parameterization of each object.

The C++ operator `new` is used to dynamically allocate memory on the heap to store the objects. As allocation returns the address to the newly allocated memory, access to the object's member functions is done using a pointer. Any memory dynamically allocated with the operator `new` must be released (deallocated) using the operator `delete`. This operator is usually called for each dynamically created member object in the destructor of the class.

The example below shows the use of dynamic memory allocation and deallocation for the BASK demodulator similar as described in Section 2.6.2.

```
SC_MODULE(bask_demod)
{
  sca_tdf::sca_in<double> in;
  sca_tdf::sca_out<bool>  out;

  rectifier*     rc;
  ltf_nd_filter* lp;
  sampler*       sp;

  SC_CTOR(bask_demod) : in("in"), out("out"), rc_out("rc_out"), lp_out("lp_out")
  {
    rc = new rectifier("rc");
    rc->in(in);
    rc->out(rc_out);

    lp = new ltf_nd_filter("lp", 3.3e6);
    lp->in(rc_out);
    lp->out(lp_out);

    sp = new sampler("sp");
    sp->in(lp_out);
    sp->out(out);
  }

  ~bask_demod()
  {
    delete(rc);
    delete(lp);
    delete(sp);
  }

 private:
  sca_tdf::sca_signal<double> rc_out, lp_out;
};
```

7.4.3. Module parameters

Modules need to be flexible to be reusable, i.e., their behavior and internal structure must be parameterized to a reasonable degree to allow their adoption to varying specifications. This is especially interesting for the early design stages of architecture exploration and successive refinement of the system structure.

In Section 2.6.1, a BASK modulator model was presented with hard coded design parameters, like the carrier frequency of 70 MHz. With respect to this carrier frequency, the time step values and the data rates where hard coded, such that the resulting signal was sufficiently sampled. Such "magic numbers", hard coded port rates, delays, and time steps, are typical signs of an inflexible implementation. If, for example, the carrier frequency would be increased without changing the time step, the model might not work properly because of undersampling.

A more flexible approach is to derive time step and data rate values from the functional module parameters. In this section it is shown how to make a parameterized version of the BASK-modulator from Section 2.6.1, with adjustable carrier-frequency and baseband frequency, and how to derive data rates and time steps automatically from that. Firstly, a mixer with parameterized data rate is needed:


```
SCA_TDF_MODULE(mixer)
{
  sca_tdf::sca_in<bool>     in_bin;  // input port baseband signal
  sca_tdf::sca_in<double>   in_wav;  // input port carrier signal
  sca_tdf::sca_out<double>  out;     // output port modulated signal

  mixer( sc_core::sc_module_name nm, unsigned long rate_ )
  : in_bin("in_bin"), in_wav("in_wav"), out("out"), rate(rate_)
  {
    using namespace sc_core; // essential for sc_assert to work, when using OSCI systemc-2.2.0
    sc_assert(rate_ > 0);
  }

  void set_attributes()
  {
    in_wav.set_rate(rate);
    out.set_rate(rate);
  }

  void processing()
  {
    for(unsigned long i = 0; i < rate; i++)
    {
      if( in_bin.read() )
        out.write( in_wav.read(i), i );
      else out.write( 0.0 , i );
    }
  }

 private:
  unsigned long rate;
};
```

If parameters are used which are computed elsewhere, it is always a good idea to make plausibility checks. Therefore, the mixers' constructor contains the line **sc_assert**(rate_ > 0) to check if the rate parameter is at least 1. Note that the implementation of **sc_assert** in the OSCI SystemC reference implementation release 2.2 (systemc-2.2.0) is not compliant to the IEEE 1666-2005 standard, and therefore a `using namespace sc_core;` has to be added, before calling **sc_assert**.

Using this mixer, and the parameterized sinusoidal source already used in Section 2.6.1, a parameterized BASK modulator can be implemented as follows:

```
SC_MODULE(bask_mod)
{
  sca_tdf::sca_in<bool>    in;
  sca_tdf::sca_out<double> out;

  sin_src sine;
  mixer   mix;

  bask_mod( sc_core::sc_module_name nm,
            double baseband_freq,
            double carrier_freq,
            double carrier_ampl = 1.0,
            unsigned long samples_per_period = 20 )
  : in("in"), out("out"),
    sine("sine",
       carrier_ampl,
       carrier_freq,
       sca_core::sca_time( (1.0 / (samples_per_period * carrier_freq) ), sc_core::SC_SEC) ),
    mix("mix", (int)ceil( static_cast<double>(samples_per_period) * carrier_freq / baseband_freq ) ),
    carrier("carrier")
  {
    using namespace sc_core; // essential for sc_assert to work, when using OSCI systemc-2.2.0

    // Plausibility checks
    sc_assert(carrier_freq > baseband_freq); // wouldn't make sense otherwise!
    sc_assert(samples_per_period > 2);       // Nyquist criterion satisfied?
    sc_assert(carrier_ampl > 0.0);           // Otherwise the output is 0 all the way!

    sine.out(carrier);
    mix.in_wav(carrier);
    mix.in_bin(in);
    mix.out(out);
  }

 private:
  sca_tdf::sca_signal<double> carrier;
```



```
};
```

The BASK modulator above can be configured with the following parameters:

- `baseband_freq` is the frequency of the binary signal.
- `carrier_freq` is the frequency of the carrier signal.
- `carrier_ampl` is the amplitude of the carrier signal, which defaults to 1.
- `samples_per_period` is the number of samples used for one period of the sinusoidal carrier signal. The default of 20 ensures sufficient sampling.

From these parameters, the appropriate parameters for the constructors of `sin_src` and `mixer` are computed. Again, the constructor contains some plausibility checks using **sc_assert**. The timestep of `sin_src` is the reverse of the product of the carrier frequency and the samples per sinus period used. For example, if the carrier frequency is 10 MHz, and 20 samples per period are used, the overall sampling frequency becomes 200 MHz, which results in a time step of 5 ns. The rate of the port `in_wav` of the mixer has to be the ratio of the product of samples per period and carrier frequency to the baseband frequency. Assuming the latter to be 2 MHz, and again a 10 MHz carrier frequency with 20 samples per period, this would result in a data rate of 100. Note that the ceiling operation in the modulator code might result in a slightly higher samples per period rate than intended.

7.4.4. Separation of module definition and implementation

The condensed examples shown so far have implemented the behavior or structural composition directly inside the class definition. It is recommended to separate the module definition from the actual implementation, into a header file (with .h or .hpp extension) and an implementation file (with .cpp extension), as it is common C++ coding practice. Thus, only the information necessary to use the module is exposed to other files including the header and not its implementation details. Duplicated code generation is avoided reducing overall compilation time. Only for template classes declaration and implementation need to be both kept in the header files, as the C++ compiler needs to be able to specialize the implementation to the passed template parameters.

The example below shows the BASK demodulator example from Section 2.6.2, where the structural composition is implemented in a separate implementation file, as part of the module constructor. The class definition is put in a header file, which allows inclusion in other files. Note that this separation cannot be applied in case a module is created using a class template.

```
// bask_demod.h

#ifndef BASK_DEMOD_H_
#define BASK_DEMOD_H_

#include <systemc-ams>

#include "rectifier.h"
#include "ltf_nd_filter.h"
#include "sampler.h"

SC_MODULE(bask_demod)
{
  sca_tdf::sca_in<double> in;
  sca_tdf::sca_out<bool> out;

  rectifier*     rc;
  ltf_nd_filter* lp;
  sampler*       sp;

  bask_demod( sc_core::sc_module_name nm );

 private:
  sca_tdf::sca_signal<double> rc_out, lp_out;
};

#endif // BASK_DEMOD_H_
```

The class implementation containing the actual structural composition is stored in a separate file:

```
// bask_demod.cpp
```



```
#include "bask_demod.h"

bask_demod::bask_demod(sc_core::sc_module_name nm)
: in("in"), out("out"), rc_out("rc_out"), lp_out("lp_out")
{
  rc = new rectifier("rc");
  rc->in(in);
  rc->out(rc_out);

  lp = new ltf_nd_filter("lp", 3.3e6);
  lp->in(rc_out);
  lp->out(lp_out);

  sp = new sampler("sp");
  sp->in(lp_out);
  sp->out(out);
}
```

7.4.5. Class templates

C++ class templates can be used in case multiple instances using different data types or sizes are needed in a design. For example, if a parallel data stream of width *N* has to be serialized, this can be modeled very naturally with a TDF module having an input data rate of 1 and an output data rate of *N*. Figure 7.11 shows the definition of a serializer, implemented as template class with parameter *N*. For serialization of a 3 bit vector, the template parameter *N* is set to 3.

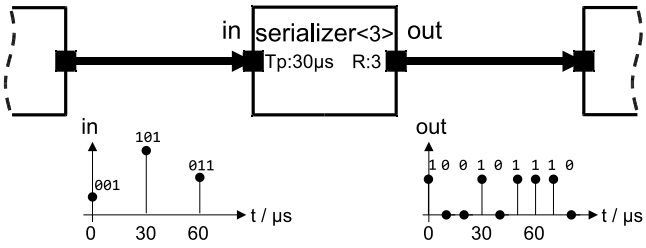


Figure 7.11. Serialization of a 3-bit vector

```
template <int N>
SCA_TDF_MODULE(serializer)
{
  sca_tdf::sca_in<sc_dt::sc_bv<N> > in; // input port
  sca_tdf::sca_out<bool> out; // output port

  SCA_CTOR(serializer) : in("in"), out("out") {}

  void set_attributes()
  {
    out.set_rate(N);
  }

  void processing()
  {
    for(int i = 0; i < N; i++)
    {
      out.write( in.read().get_bit(i), i );
    }
  }
};
```

The example below shows how such a template class can be used within a structural module.

```
SC_MODULE(modulator)
{
  sca_tdf::sca_in<sc_dt::sc_bv<3> > in;
  sca_tdf::sca_out<double> out;

  serializer<3> ser;
  bask_mod      mod;

  SC_CTOR(modulator) : in("in"), out("out"), ser("ser"), mod("mod"), bits("bits")
  {
    ser.in(in);
```

```
    ser.out(bits);

    mod.in(bits);
    mod.out(out);
  }

 private:
  sca_tdf::sca_signal<bool> bits;
};
```

Class templates also facilitate refinement of communication, as discussed in Section 7.3. The example below shows the amplifier module of Section 7.2 implemented as a class template. Depending on the template parameter type, the module can be used either as a passband model, when using type *double*, or as a baseband model using data type **sca_util::sca_complex**.

```
#include <cmath>
#include <complex>

template <class T>
SCA_TDF_MODULE(amplifier)
{
  sca_tdf::sca_in<T>  in;
  sca_tdf::sca_out<T> out;

  amplifier( sc_core::sc_module_name, double gain = 1.0, double iip3 = 1e-3 )
  : in("in"), out("out"), a1( gain ), a3( -4/3 * (gain / std::pow(iip3,2)) ) {}

  void processing()
  {
    out.write( a1 * in.read() + a3 * std::pow(in.read(),3) );
  }

 private:
  double a1, a3;
};
```

7.4.6. Public and private class members

When creating a module using the macro **SC_MODULE** or **SCA_TDF_MODULE**, a class is defined by using the C++ keyword `struct`. In this case, all class members, such as functions and data variables, are public by default. These members can be accessed from outside the class, for example from a function, e.g., the main program **sc_main**, or from another class, e.g., a parent module. Modules which are defined with the keyword `class` have private members by default.

In order to be able to instantiate a module, and connect it with other modules, the constructor and ports have to be declared as public. It is recommended to declare internal signals, nodes, variables, functions and primitive modules as private, unless there is a good reason to access them from outside the scope of the class. For example, signals and nodes could be made public to facilitate debugging.

To facilitate tracing of signals or nodes which are declared private, a helper function **trace_internals** can be defined as public member, which will write the signals to a trace file defined by the argument. The example below extends the BASK demodulator from Section 2.6.2 with tracing of private members. In this case, there is no need to declare the signals itself as public.

```
SC_MODULE(bask_demod)
{
  sca_tdf::sca_in<double> in;
  sca_tdf::sca_out<bool>  out;

  rectifier     rc;
  ltf_nd_filter lp;
  sampler       sp;

  SC_CTOR(bask_demod)
  : in("in"), out("out"), rc("rc"), lp("lp", 3.3e6), sp("sp"), rc_out("rc_out"), lp_out("lp_out")
  {
    rc.in(in);
    rc.out(rc_out);

    lp.in(rc_out);
    lp.out(lp_out);

    sp.in(lp_out);
```



```
    sp.out(out);
  }

  void trace_internals( sca_util::sca_trace_file* tf )
  {
    sca_util::sca_trace(tf, rc_out, rc_out.name() );
    sca_util::sca_trace(tf, lp_out, lp_out.name() );
  }

 private:
  sca_tdf::sca_signal<double> rc_out, lp_out;
};
```

This page is intentionally left blank.

Appendix A. Language reference

Note: This appendix gives only a list of the basic language definitions for TDF, LSF or ELN primitive modules. The complete list of definitions can be found in the Language Reference Manual of the SystemC AMS extensions.

If the default value for a parameter is not given in the tables below, then the value has to be provided by the user and cannot be omitted during construction.

A.1. TDF modules

```
// Name     Type                    Description
// ------------------------------------------------------------------------------------------
//           T                      Arbitrary data type (e.g double, sca_util::sca_vector, ...
// tstep    sca_core::sca_time      Time step as object
// abstime  sca_core::sca_time      Time step as object
// tstepd   double                  Time step in seconds
// tunit    sc_core::sc_time_unit   Time unit (e.g., sc_core::SC_US, sc_core::SC_MS, ...)
// name     const char*             Module name as string
// modname  sc_core::sc_module_name Module name as object
// ------------------------------------------------------------------------------------------

SCA_TDF_MODULE( name )
{
  // port declarations
  sca_tdf::sca_in<T>  in;  // input port
  sca_tdf::sca_out<T> out; // output port

  // Converter ports
  sca_tdf::sca_de::sca_in<T>  inp;  // converter port from discrete-event domain
  sca_tdf::sca_de::sca_out<T> outp; // converter port to discrete-event domain

  // TDF methods, called automatically by the scheduler
  void set_attributes()
  {
    // module and port attributes (optional)
  }

  void initialize()
  {
    // initial values of ports with a delay (optional)
  }

  void processing()
  {
    // time-domain signal processing behavior or algorithm (mandatory)
  }

  void ac_processing()
  {
    // small-signal frequency-domain behavior (optional)
  }

  // module constructor
  SCA_CTOR( name ) {} // macro, or
  name( modname  ) {} // full constructor, can also be used to pass parameters

};
```

A.2. TDF ports

```
// Name        Type                   Description
// ------------------------------------------------------------------------------------------
// value        T                     Value with arbitrary type (double, sca_util::sca_vector, ...)
// sample_id   unsigned long          Sample ID: 0 for single-rate, 0...(rate-1) for multirate
// nsamples    unsigned long          Number of samples
// rate        unsigned long          Rate of the port
// tstep       sca_core::sca_time     Time step as object
// tstepd      double                 Time step in seconds
// tunit       sc_core::sc_time_unit  Time unit (e.g., sc_core::SC_US, sc_core::SC_MS, ...)
// toffset     sca_core::sca_time     Time offset as object
// toffsetd    double                 Time offset in seconds
// ------------------------------------------------------------------------------------------
```

```
sca_tdf::sca_in<T> in;
sca_tdf::sca_out<T> out;

sca_tdf::sca_de::sca_in<T>  inp;
sca_tdf::sca_de::sca_out<T> outp;

out.set_delay( nsamples );
out.set_rate( rate );
out.set_timestep( tstep );
out.set_timestep( tstepd, tunit );

outp.set_timeoffset( toffset );
outp.set_timeoffset( toffsetd, tunit );

nsamples = out.get_delay();
rate     = out.get_rate();

abstime = out.get_time();
abstime = out.get_time( sample_id );
tstep   = out.get_timestep();
tstepd  = out.get_timestep().to_seconds();

toffset = outp.get_timeoffset();

out.initialize( value, sample_id );

value = in.read();
value = in.read( sample_id );

out.write( value );
out.write( value, sample_id );
```

A.3. TDF signals

```
// type T
sca_tdf::sca_signal<T>  // TDF signal
```

A.4. Embedded Laplace transfer functions

A.4.1. sca_tdf::sca_ltf_nd

Description

Scaled Laplace transfer function in the time-domain in the numerator-denominator form.

Definition

```
sca_tdf::sca_ltf_nd( num, den, delay, state, input, k, tstep );
```

Equation

$$H(s) = k \cdot \frac{\sum_{i=0}^{M-1} num_i \cdot s^i}{\sum_{i=0}^{N-1} den_i \cdot s^i} \cdot e^{(-s \cdot delay)}$$

Parameters

Name	Type	Default	Description
num	**sca_util::sca_vector**<double>		Numerator coefficients
den	**sca_util::sca_vector**<double>		Denumerator coefficients
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Time continuous delay (optional)
state	**sca_util::sca_vector**<double>		State vector (optional)
input			Input value, or signal from port

Name	Type	Default	Description
	double, **sca_tdf::sca_in**<double>, **sca_tdf::sca_de::sca_in**<double>, **sca_util::sca_vector**<double>		
k	double	1.0	Gain coefficient (optional)
tstep	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Time step

Constraint of usage

The delay shall be greater or equal to zero.

A.4.2. sca_tdf::sca_ltf_zp

Description

Scaled Laplace transfer function in the time-domain in the zero-pole form.

Definition

```
sca_tdf::sca_ltf_zp( zeros, poles, delay, state, input, k, tstep );
```

Equation

$$H(s) = k \cdot \frac{\prod_{i=0}^{M-1}(s - zeros_i)}{\prod_{i=0}^{N-1}(s - poles_i)} \cdot e^{(-s \cdot delay)}$$

Parameters

Name	Type	Default	Description
zeros	**sca_util::sca_vector**< **sca_util::sca_complex >**		Numerator coefficients
poles	**sca_util::sca_vector**< **sca_util::sca_complex** >		Denumerator coefficients
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Time continuous delay (optional)
state	**sca_util::sca_vector**<double>		State vector (optional)
input	double, **sca_tdf::sca_in**<double>, **sca_tdf::sca_de::sca_in**<double>, **sca_util::sca_vector**<double>		Input value, or signal from port
k	double	1.0	Gain coefficient (optional)
tstep	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Time step

Constraint of usage

The delay shall be greater or equal to zero.

A.4.3. sca_tdf::sca_ss

Description

Single-input single-output state-space equation.

Definition

```
sca_tdf::ss_eqn( a, b, c, d, delay, s, x, tstep );
```

Equation

$$\frac{ds(t)}{dt} = \mathbf{A} \cdot s(t) + \mathbf{B} \cdot x(t - delay)$$
$$y(t) = \mathbf{C} \cdot s(t) + \mathbf{D} \cdot x(t - delay)$$

Parameters

Name	Type	Default	Description
a	**sca_util::sca_matrix**<double>		Matrix A of size n-by-n (n = number of states)
b	**sca_util::sca_matrix**<double>		Matrix B of size n-by-m (m = number of inputs)
c	**sca_util::sca_matrix**<double>		Matrix C of size r-by-n (r = number of outputs)
d	**sca_util::sca_matrix**<double>		Matrix D of size r-by-m
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Time continuous delay (optional)
state	**sca_util::sca_vector<double>**		State vector (optional)
x	**sca_util::sca_vector**<double>, **sca_util::sca_matrix**<double>, **sca_tdf::sca_in**<double>, **sca_tdf::sca_in**< **sca_util::sca_vector**<double>, **sca_tdf::sca_de::sca_in**< **sca_util::sca_vector**<double>		Input vector, matrix or signal from port
tstep	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Time step

Constraint of usage

The delay shall be greater or equal to zero.

A.5. LSF primitive modules

A.5.1. sca_lsf::sca_add

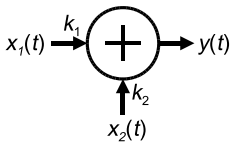
Description

Weighted addition of two LSF signals.

Definition

```
sca_lsf::sca_add( nm, k1, k2 );
```

Symbol



Equation

$$y(t) = k_1 \cdot x_1(t) + k_2 \cdot x_2(t)$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
k1	double	1.0	Weighting coefficient for LSF signal at port x1
k2	double	1.0	Weighting coefficient for LSF signal at port x2

Ports

Name	Interface	Type/Nature	Description
x1	**sca_lsf::sca_in**	Signal flow	LSF input 1
x2	**sca_lsf::sca_in**	Signal flow	LSF input 2
y	**sca_lsf::sca_out**	Signal flow	LSF output

A.5.2. sca_lsf::sca_sub

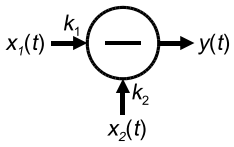
Description

Weighted subtraction of two LSF signals.

Definition

```
sca_lsf::sca_sub( nm, k1, k2 );
```

Symbol



Equation

$$y(t) = k_1 \cdot x_1(t) - k_2 \cdot x_2(t)$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
k1	double	1.0	Weighting coefficient for LSF signal at port x1
k2	double	1.0	Weighting coefficient for LSF signal at port x2

Ports

Name	Interface	Type/Nature	Description
x1	**sca_lsf::sca_in**	Signal flow	LSF input 1
x2	**sca_lsf::sca_in**	Signal flow	LSF input 2
y	**sca_lsf::sca_out**	Signal flow	LSF output

A.5.3. sca_lsf::sca_gain

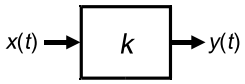
Description

Multiplication of an LSF signal by a constant gain.

Definition

```
sca_lsf::sca_gain( nm, k );
```

Symbol



Equation

$$y(t) = k \cdot x(t)$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
k	double	1.0	Gain coefficient

Ports

Name	Interface	Type/Nature	Description
x	**sca_lsf::sca_in**	Signal flow	LSF input
y	**sca_lsf::sca_out**	Signal flow	LSF output

A.5.4. sca_lsf::sca_dot

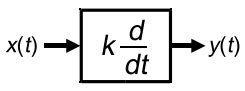
Description

Scaled first-order time derivative of an LSF signal.

Definition

```
sca_lsf::sca_dot( nm, k );
```

Symbol



Equation

$$y(t) = k \cdot \frac{dx(t)}{dt}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
k	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
x	**sca_lsf::sca_in**	Signal flow	LSF input
y	**sca_lsf::sca_out**	Signal flow	LSF output

A.5.5. sca_lsf::sca_integ

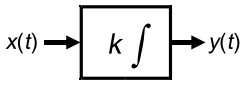
Description

Scaled time-domain integration of an LSF signal.

Definition

```
sca_lsf::sca_integ( nm, k, y0 );
```

Symbol



Equation

$$y(t) = k \cdot \int_o^t x(t)dt + y_0$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
k	double	1.0	Scale coefficient
y0	double	0.0	Initial condition at t=0

Ports

Name	Interface	Type/Nature	Description
x	**sca_lsf::sca_in**	Signal flow	LSF input
y	**sca_lsf::sca_out**	Signal flow	LSF output

A.5.6. sca_lsf::sca_delay

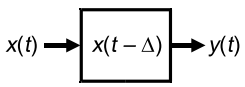
Description

Scaled time-delayed version of an LSF signal.

Definition

```
sca_lsf::sca_delay( nm, delay, k, y0 );
```

Symbol



Equation

$$y(t) = \begin{cases} y_0 & t \le delay \\ k \cdot x(t - delay) & t > delay \end{cases}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Time continuous delay
k	double	1.0	Scale coefficient
y0	double	0.0	Output value before delay is in effect

Ports

Name	Interface	Type/Nature	Description
x	**sca_lsf::sca_in**	Signal flow	LSF input
y	**sca_lsf::sca_out**	Signal flow	LSF output

Constraint of usage

The delay shall be greater or equal to zero.

A.5.7. sca_lsf::sca_source

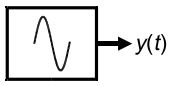
Description

LSF source.

Definition

```
sca_lsf::sca_source( nm, init_value, offset, amplitude, frequency, phase, delay,
                     ac_amplitude, ac_phase, ac_noise_amplitude );
```

Symbol



Equation

For time-domain simulation:

$$y(t) = \begin{cases} init\_value & t < delay \\ offset + amplitude \cdot \sin(2\pi \cdot frequency \cdot (t - delay) + phase) & t \ge delay \end{cases}$$

For small-signal frequency-domain simulation:

$$y(f) = ac\_amplitude \cdot \{\cos(ac\_phase) + j \cdot \sin(ac\_phase)\}$$

For small-signal frequency-domain noise simulation:

$$y(f) = ac\_noise\_amplitude$$

Parameters

Name	Type	Default	Description
nm	**sc_core:: sc_module_name**		Module name
init_value	double	0.0	Initial value
offset	double	0.0	Offset value
amplitude	double	0.0	Source amplitude
frequency	double	0.0	Source frequency in Hertz
phase	double	0.0	Source phase in radian
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Time continuous delay
ac_amplitude	double	0.0	Small-signal amplitude *)
ac_phase	double	0.0	Small-signal phase in radian *)
ac_noise_ amplitude	double	0.0	Small-signal noise amplitude **)

*) for small-signal frequency-domain simulation only.

**) for small-signal frequency-domain noise simulation only.

Ports

Name	Interface	Type/Nature	Description
y	**sca_lsf::sca_out**	Signal flow	LSF output

Constraint of usage

The delay shall be greater or equal to zero.

A.5.8. sca_lsf::sca_ltf_nd

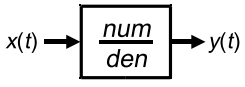
Description

Scaled Laplace transfer function in the time-domain in the numerator-denominator form.

Definition

```
sca_lsf::sca_ltf_nd( nm, num, den, delay, k );
```

Symbol



Equation

$$den_{N-1}\frac{d^{N-1}y(t)}{dt} + den_{N-2}\frac{d^{N-2}y(t)}{dt} + \cdots + den_1\frac{dy(t)}{dt} + den_0 \cdot y(t)$$

$$= k \cdot \left(num_{M-1}\frac{d^{M-1}x(t-delay)}{dt} + num_{M-2}\frac{d^{M-2}x(t-delay)}{dt} + \cdots + num_1\frac{dx(t-delay)}{dt} + num_0 \cdot x(t-delay) \right)$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
num	**sca_util::sca_vector**<double>		Numerator coefficients
den	**sca_util::sca_vector**<double>		Denumerator coefficients
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Time continuous delay
k	double	1.0	Gain coefficient

Ports

Name	Interface	Type/Nature	Description
x	**sca_lsf::sca_in**	Signal flow	LSF output
y	**sca_lsf::sca_out**	Signal flow	LSF output

Constraint of usage

The delay shall be greater or equal to zero.

A.5.9. sca_lsf::sca_ltf_zp

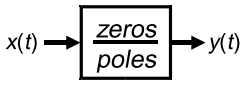
Description

Scaled Laplace transfer function in the time-domain in the zero-pole form.

Definition

```
sca_lsf::sca_ltf_zp( nm, zeros, poles, delay, k );
```

Symbol



Equation

$$\left(\frac{d}{dt} - poles_{N-1}\right)\left(\frac{d}{dt} - poles_{N-2}\right)\cdots\left(\frac{d}{dt} - poles_{1}\right)\left(\frac{d}{dt} - poles_{0}\right)y(t)$$

$$= k \cdot \left\{\left(\frac{d}{dt} - zeros_{M-1}\right)\left(\frac{d}{dt} - zeros_{M-2}\right)\cdots\left(\frac{d}{dt} - zeros_{1}\right)\left(\frac{d}{dt} - zeros_{0}\right)x(t - delay)\right\}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
zeros	**sca_util::sca_vector**< **sca_util::sca_complex**>		Zeros
poles	**sca_util::sca_vector**< **sca_util::sca_complex**>		Poles
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Time continuous delay

Name	Type	Default	Description
k	double	1.0	Gain coefficient

Ports

Name	Interface	Type/Nature	Description
x	**sca_lsf::sca_in**	Signal flow	LSF output
y	**sca_lsf::sca_out**	Signal flow	LSF output

Constraints on usage

The expansion of the numerator and the denominator shall result in a real value, respectively. The delay shall be greater or equal to zero.

A.5.10. sca_lsf::sca_ss

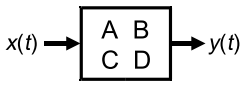
Description

Single-input single-output state-space equation.

Definition

```
sca_lsf::sca_ss( nm, a, b, c, d, delay );
```

Symbol



Equation

$$\frac{ds(t)}{dt} = \mathbf{A} \cdot s(t) + \mathbf{B} \cdot x(t - delay)$$
$$y(t) = \mathbf{C} \cdot s(t) + \mathbf{D} \cdot x(t - delay)$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
a	**sca_util::sca_matrix<double>**		Matrix A of size n-by-n
b	**sca_util::sca_matrix<double>**		Matrix B with one column of size n
c	**sca_util::sca_matrix<double>**		Matrix C with one row of size n
d	**sca_util::sca_matrix<double>**		Matrix D of size 1
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Time continuous delay

Ports

Name	Interface	Type/Nature	Description
x	**sca_lsf::sca_in**	Signal flow	LSF output
y	**sca_lsf::sca_out**	Signal flow	LSF output

Constraint of usage

The delay shall be greater or equal to zero.

A.5.11. sca_lsf::sca_tdf::sca_gain, sca_lsf::sca_tdf_gain

Description

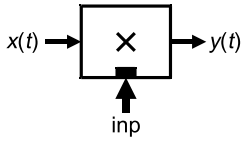
Scaled multiplication of a TDF input signal by an LSF input signal.

Definition

```
sca_lsf::sca_tdf::sca_gain( nm, scale );

sca_lsf::sca_tdf_gain( nm, scale );
```

Symbol



Equation

$$y(t) = scale \cdot inp \cdot x(t)$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
inp	**sca_tdf::sca_in<T>**	double	TDF input
x	**sca_lsf::sca_in**	Signal flow	LSF output
y	**sca_lsf::sca_out**	Signal flow	LSF output

A.5.12. sca_lsf::sca_tdf::sca_source, sca_lsf::sca_tdf_source

Description

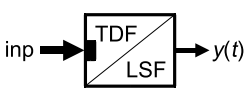
Scaled conversion of a TDF input signal to an LSF output signal.

Definition

```
sca_lsf::sca_tdf::sca_source( nm, scale );

sca_lsf::sca_tdf_source( nm, scale );
```

Symbol



Equation

$y(t) = scale \cdot inp$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
inp	**sca_tdf::sca_in<T>**	double	TDF input
y	**sca_lsf::sca_out**	Signal flow	LSF output

A.5.13. sca_lsf::sca_tdf::sca_sink, sca_lsf::sca_tdf_sink

Description

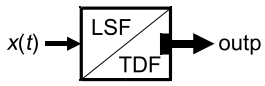
Scaled conversion from an LSF input signal to a TDF output signal.

Definition

```
sca_lsf::sca_tdf::sca_sink( nm, scale );

sca_lsf::sca_tdf_sink( nm, scale );
```

Symbol



Equation

There is no equation contributed to the overall equation system for this module.

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
x	**sca_lsf::sca_in**	Signal flow	LSF input
outp	**sca_tdf::sca_out<T>**	double	TDF output

A.5.14. sca_lsf::sca_tdf::sca_mux, sca_lsf::sca_tdf_mux

Description

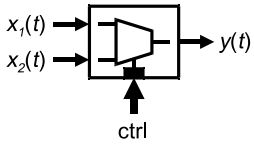
Selection of one of two LSF input signals by a TDF control signal (multiplexer).

Definition

```
sca_lsf::sca_tdf::sca_mux( nm );

sca_lsf::sca_tdf_mux( nm );
```

Symbol



Equation

$$y(t) = \begin{cases} x_1(t) & ctrl = false \\ x_2(t) & ctrl = true \end{cases}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name

Ports

Name	Interface	Type/Nature	Description
x1	**sca_lsf::sca_in**	Signal flow	LSF input 1
x2	**sca_lsf::sca_in**	Signal flow	LSF input 2
ctrl	**sca_tdf::sca_in<T>**	bool	TDF control input
y	**sca_lsf::sca_out**	Signal flow	LSF output

A.5.15. sca_lsf::sca_tdf::sca_demux, sca_lsf::sca_tdf_demux

Description

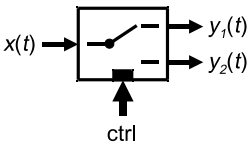
Routing of an LSF input signal to either one of two LSF output signals controlled by a TDF signal (demultiplexer).

Definition

```
sca_lsf::sca_tdf::sca_demux( nm );

sca_lsf::sca_tdf_demux( nm );
```

Symbol



Equation

$$y_1(t) = \begin{cases} x(t) & ctrl = false \\ 0 & ctrl = true \end{cases}$$

$$y_2(t) = \begin{cases} 0 & ctrl = false \\ x(t) & ctrl = true \end{cases}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name

Ports

Name	Interface	Type/Nature	Description
x	**sca_lsf::sca_in**	Signal flow	LSF input
ctrl	**sca_tdf::sca_in<T>**	bool	TDF control input
y1	**sca_lsf::sca_out**	Signal flow	LSF output 1
y2	**sca_lsf::sca_out**	Signal flow	LSF output 2

A.5.16. sca_lsf::sca_de::sca_gain, sca_lsf::sca_de_gain

Description

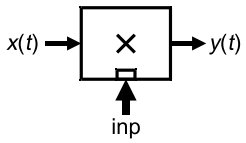
Scaled multiplication of a discrete-event input signal by an LSF input signal.

Definition

```
sca_lsf::sca_de::sca_gain( nm, scale );

sca_lsf::sca_de_gain( nm, scale );
```

Symbol



Equation

$$y(t) = scale \cdot inp \cdot x(t)$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
inp	**sc_core::sc_in<T>**	double	Discrete-event input
x	**sca_lsf::sca_in**	Signal flow	LSF input
y	**sca_lsf::sca_out**	Signal flow	LSF output

A.5.17. sca_lsf::sca_de::sca_source, sca_lsf::sca_de_source

Description

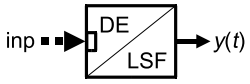
Scaled conversion of a discrete-event input signal to an LSF output signal.

Definition

```
sca_lsf::sca_de::sca_source( nm, scale );

sca_lsf::sca_de_source( nm, scale );
```

Symbol



Equation

$y(t) = scale \cdot inp$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
inp	**sc_core::sc_in<T>**	double	Discrete-event input
y	**sca_lsf::sca_out**	Signal flow	LSF output

A.5.18. sca_lsf::sca_de::sca_sink, sca_lsf::sca_de_sink

Description

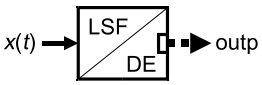
Scaled conversion from an LSF input signal to a discrete-event output signal.

Definition

```
sca_lsf::sca_de::sca_sink( nm, scale );

sca_lsf::sca_de_sink( nm, scale );
```

Symbol



Equation

There is no equation contributed to the overall equation system for this module.

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name

Name	Type	Default	Description
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
x	**sca_lsf::sca_in**	Signal flow	LSF input
outp	**sc_core::sc_out<T>**	double	Discrete-event output

A.5.19. sca_lsf::sca_de::sca_mux, sca_lsf::sca_de_mux

Description

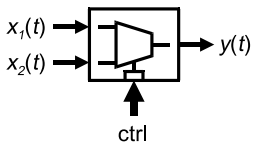
Selection of one of two LSF input signals by a discrete-event control signal (multiplexer).

Definition

```
sca_lsf::sca_de::sca_mux( nm );

sca_lsf::sca_de_mux( nm );
```

Symbol



Equation

$$y(t) = \begin{cases} x_1(t) & ctrl = false \\ x_2(t) & ctrl = true \end{cases}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name

Ports

Name	Interface	Type/Nature	Description
x1	**sca_lsf::sca_in**	Signal flow	LSF input 1
x2	**sca_lsf::sca_in**	Signal flow	LSF input 2
ctrl	**sc_core::sc_in<T>**	bool	Discrete-event control input
y	**sca_lsf::sca_out**	Signal flow	LSF output

A.5.20. sca_lsf::sca_de::sca_demux, sca_lsf::sca_de_demux

Description

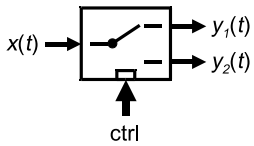
Routing of an LSF input signal to either one of two LSF output signals controlled by a discrete-event control signal (demultiplexer).

Definition

```
sca_lsf::sca_de::sca_demux( nm );
```

```
sca_lsf::sca_de_demux( nm );
```

Symbol



Equation

$$y_1(t) = \begin{cases} x(t) & ctrl = false \\ 0 & ctrl = true \end{cases}$$

$$y_2(t) = \begin{cases} 0 & ctrl = false \\ x(t) & ctrl = true \end{cases}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name

Ports

Name	Interface	Type/Nature	Description
x	**sca_lsf::sca_in**	Signal flow	LSF input
ctrl	**sc_core::sc_in<T>**	bool	Discrete-event control input
y1	**sca_lsf::sca_out**	Signal flow	LSF output 1
y2	**sca_lsf::sca_out**	Signal flow	LSF output 2

A.6. ELN primitive modules

A.6.1. sca_eln::sca_r

Description

Resistor.

Definition

```
sca_eln::sca_r( nm, value  );
```

Symbol

Equation

$$v_{p,n}(t) = i_{p,n}(t) \cdot value$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
value	double	1.0	Resistance in Ohm

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal

A.6.2. sca_eln::sca_c

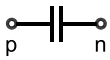
Description

Capacitor.

Definition

```
sca_eln::sca_c( nm, value, q0 );
```

Symbol



Equation

$$i_{p,n}(t) = \frac{d\left(value \cdot v_{p,n}(t) + q_0\right)}{dt}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
value	double	1.0	Capacitance in Farad
q0	double	0.0	Initial charge in Coulomb

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal

Constraint of usage

The parameter *value* shall not be numerically zero.

A.6.3. sca_eln::sca_l

Description

Inductor.

Definition

```
sca_eln::sca_l( nm, value, phi0 );
```

Symbol

Equation

$$v_{p,n}(t) = \frac{d(value \cdot i_{p,n}(t) + phi_0)}{dt}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
value	double	1.0	Inductance in Henry
phi0	double	0.0	Initial magnetic flux in Weber

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal

Constraint of usage

The parameter *value* shall not be numerically zero.

A.6.4. sca_eln::sca_vcvs

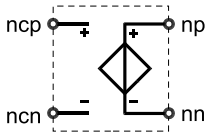
Description

Voltage controlled voltage source.

Definition

```
sca_eln::sca_vcvs( nm, value );
```

Symbol



Equation

$$v_{np,nn}(t) = value \cdot v_{ncp,ncn}(t)$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
value	double	1.0	Scale coefficient of the control voltage

Ports

Name	Interface	Type/Nature	Description
ncp	**sca_eln::sca_terminal**	Electrical	Positive control terminal
ncn	**sca_eln::sca_terminal**	Electrical	Negative control terminal

Name	Interface	Type/Nature	Description
np	**sca_eln::sca_terminal**	Electrical	Positive terminal of source
nn	**sca_eln::sca_terminal**	Electrical	Negative terminal of source

A.6.5. sca_eln::sca_vccs

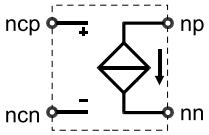
Description

Voltage controlled current source.

Definition

```
sca_eln::sca_vccs( nm, value );
```

Symbol



Equation

$$i_{np,nn}(t) = value \cdot v_{ncp,ncn}(t)$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
value	double	1.0	Scale coefficient in Siemens of the control voltage

Ports

Name	Interface	Type/Nature	Description
ncp	**sca_eln::sca_terminal**	Electrical	Positive control terminal
ncn	**sca_eln::sca_terminal**	Electrical	Negative control terminal
np	**sca_eln::sca_terminal**	Electrical	Positive terminal of source
nn	**sca_eln::sca_terminal**	Electrical	Negative terminal of source

A.6.6. sca_eln::sca_ccvs

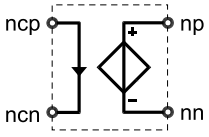
Description

Current controlled voltage source.

Definition

```
sca_eln::sca_ccvs( nm, value );
```

Symbol



Equation

$v_{np,nn}(t) = value \cdot i_{ncp,ncn}(t)$

$v_{ncp,ncn}(t) = 0$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
value	double	1.0	Scale coefficient in Ohm of the control current

Ports

Name	Interface	Type/Nature	Description
ncp	**sca_eln::sca_terminal**	Electrical	Positive control terminal
ncn	**sca_eln::sca_terminal**	Electrical	Negative control terminal
np	**sca_eln::sca_terminal**	Electrical	Positive terminal of source
nn	**sca_eln::sca_terminal**	Electrical	Negative terminal of source

A.6.7. sca_eln::sca_cccs

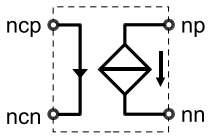
Description

Current controlled current source.

Definition

```
sca_eln::sca_cccs( nm, value );
```

Symbol



Equation

$i_{np,nn}(t) = value \cdot i_{ncp,ncn}(t)$

$v_{ncp,ncn}(t) = 0$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
value	double	1.0	Scale coefficient of the control current

Ports

Name	Interface	Type/Nature	Description
ncp	**sca_eln::sca_terminal**	Electrical	Positive control terminal
ncn	**sca_eln::sca_terminal**	Electrical	Negative control terminal

Name	Interface	Type/Nature	Description
np	**sca_eln::sca_terminal**	Electrical	Positive terminal of source
nn	**sca_eln::sca_terminal**	Electrical	Negative terminal of source

A.6.8. sca_eln::sca_nullor

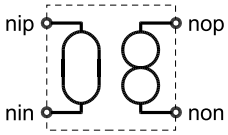
Description

Nullor (nullator - norator pair), ideal Opamp.

Definition

```
sca_eln::sca_nullor( nm );
```

Symbol



Equation

$v_{nip,nin}(t) = 0$

$i_{nip,nin}(t) = 0$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name

Ports

Name	Interface	Type/Nature	Description
nip	**sca_eln::sca_terminal**	Electrical	Positive terminal of nullator
nin	**sca_eln::sca_terminal**	Electrical	Negative terminal of nullator
nop	**sca_eln::sca_terminal**	Electrical	Positive terminal of norator
non	**sca_eln::sca_terminal**	Electrical	Negative terminal of norator

A.6.9. sca_eln::sca_gyrator

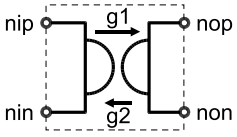
Description

Gyrator.

Definition

```
sca_eln::sca_gyrator( nm, g1, g2 );
```

Symbol



Equation

$i_{p_1,n_1}(t) = g_2 \cdot v_{p_2,n_2}(t)$

$i_{p_2,n_2}(t) = -g_1 \cdot v_{p_1,n_1}(t)$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
g1	double	1.0	Gyration conductance in Siemens
g2	double	1.0	Gyration conductance in Siemens

Ports

Name	Interface	Type/Nature	Description
p1	**sca_eln::sca_terminal**	Electrical	Positive terminal of primary port
n1	**sca_eln::sca_terminal**	Electrical	Negative terminal of primary port
p2	**sca_eln::sca_terminal**	Electrical	Positive terminal of secondary port
n2	**sca_eln::sca_terminal**	Electrical	Negative terminal of secondary port

A.6.10. sca_eln::sca_ideal_transformer

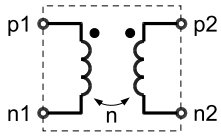
Description

Ideal transformer.

Definition

```
sca_eln::sca_ideal_transformer( nm, ratio );
```

Symbol



Equation

$v_{p1,n1}(t) = ratio \cdot v_{p2,n2}(t)$

$i_{p2,n2}(t) = ratio \cdot i_{p1,n1}(t)$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
ratio	double	1.0	Transformation ratio

Ports

Name	Interface	Type/Nature	Description
p1	**sca_eln::sca_terminal**	Electrical	Positive terminal of primary port

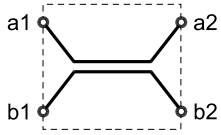
Name	Interface	Type/Nature	Description
n1	**sca_eln::sca_terminal**	Electrical	Negative terminal of primary port
p2	**sca_eln::sca_terminal**	Electrical	Positive terminal of secondary port
n2	**sca_eln::sca_terminal**	Electrical	Negative terminal of secondary port

A.6.11. sca_eln::sca_transmission_line

Description

Transmission line.

Symbol



Definition

```
sca_eln::sca_transmission_line( nm, z0, delay, delta0 );
```

Equation

$$v_{a_1,b_1}(t) = \begin{cases} z_0 \cdot i_{a_1,b_1}(t) & t < delay \\ e^{-delta_0 \cdot delay}\left(v_{a_2,b_2}(t-delay) + z_0 \cdot i_{a_2,b_2}(t-delay)\right) + z_0 \cdot i_{a_1,b_1}(t) & t \geq delay \end{cases}$$

$$v_{a_2,b_2}(t) = \begin{cases} z_0 \cdot i_{a_2,b_2}(t) & t < delay \\ e^{-delta_0 \cdot delay}\left(v_{a_1,b_1}(t-delay) + z_0 \cdot i_{a_1,b_1}(t-delay)\right) + z_0 \cdot i_{a_2,b_2}(t) & t \geq delay \end{cases}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
z0	double	100.0	Characteristic impedance of the transmission line in Ohm
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Transmission delay
delta0	double	0.0	Dissipation factor in 1/ seconds.

Ports

Name	Interface	Type/Nature	Description
a1	**sca_eln::sca_terminal**	Electrical	Wire A at primary side
b1	**sca_eln::sca_terminal**	Electrical	Wire B at primary side
a2	**sca_eln::sca_terminal**	Electrical	Wire A at secondary side
b2	**sca_eln::sca_terminal**	Electrical	Wire B at secondary side

Constraint of usage

The delay shall be greater or equal to zero.

A.6.12. sca_eln::sca_vsource

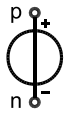
Description

Independent voltage source.

Definition

```
sca_eln::sca_vsource( nm, init_value, offset, amplitude, frequency, phase, delay,
                      ac_amplitude, ac_phase, ac_noise_amplitude );
```

Symbol



Equation

For time-domain simulation:

$$v_{p,n}(t) = \begin{cases} init\_value & t < delay \\ offset + amplitude \cdot \sin(2\pi \cdot frequency \cdot (t - delay) + phase) & t \geq delay \end{cases}$$

For small-signal frequency-domain simulation:

$$v_{p,n}(f) = ac\_amplitude \cdot \{\cos(ac\_phase) + j \cdot \sin(ac\_phase)\}$$

For small-signal frequency-domain noise simulation:

$$v_{p,n}(f) = ac\_noise\_amplitude$$

Parameters

Name	Type	Default	Description
nm	**sc_core:: sc_module_name**		Module name
init_value	double	0.0	Initial value
offset	double	0.0	Offset value
amplitude	double	0.0	Source amplitude
frequency	double	0.0	Source frequency in Hertz
phase	double	0.0	Source phase in radian
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Time continuous delay
ac_amplitude	double	0.0	Small-signal amplitude *)
ac_phase	double	0.0	Small-signal phase in radian *)
ac_noise_ amplitude	double	0.0	Small-signal noise amplitude **)

*) for small-signal frequency-domain simulation only.

**) for small-signal frequency-domain noise simulation only.

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal

Name	Interface	Type/Nature	Description
n	**sca_eln::sca_terminal**	Electrical	Negative terminal

Constraint of usage

The delay shall be greater or equal to zero.

A.6.13. sca_eln::sca_isource

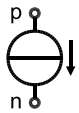
Description

Independent current source.

Definition

```
sca_eln::sca_isource( nm, init_value, offset, amplitude, frequency, phase, delay,
                      ac_amplitude, ac_phase, ac_noise_amplitude );
```

Symbol



Equation

For time-domain simulation:

$$i_{p,n}(t) = \begin{cases} init\_value & t < delay \\ offset + amplitude \cdot \sin(2\pi \cdot frequency \cdot (t - delay) + phase) & t \geq delay \end{cases}$$

For small-signal frequency-domain simulation:

$$i_{p,n}(f) = ac\_amplitude \cdot \{\cos(ac\_phase) + j \cdot \sin(ac\_phase)\}$$

For small-signal frequency-domain noise simulation:

$$i_{p,n}(f) = ac\_noise\_amplitude$$

Parameters

Name	Type	Default	Description
nm	**sc_core:: sc_module_name**		Module name
init_value	double	0.0	Initial value
offset	double	0.0	Offset value
amplitude	double	0.0	Source amplitude
frequency	double	0.0	Source frequency in Hertz
phase	double	0.0	Source phase in radian
delay	**sca_core::sca_time**	**sc_core::SC_ZERO_TIME**	Time continuous delay
ac_amplitude	double	0.0	Small-signal amplitude *)
ac_phase	double	0.0	Small-signal phase in radian *)
ac_noise_ amplitude	double	0.0	Small-signal noise amplitude **)

*) for small-signal frequency-domain simulation only.

**) for small-signal frequency-domain noise simulation only.

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal

Constraint of usage

The delay shall be greater or equal to zero.

A.6.14. sca_eln::sca_tdf::sca_r, sca_eln::sca_tdf_r

Description

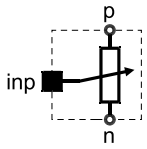
Variable resistor controlled by a TDF input signal.

Definition

```
sca_eln::sca_tdf::sca_r( nm, scale );

sca_eln::sca_tdf_r( nm, scale );
```

Symbol



Equation

$$v_{p,n}(t) = scale \cdot inp \cdot i_{p,n}(t)$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
inp	**sca_tdf::sca_in<T>**	double	TDF control input

A.6.15. sca_eln::sca_tdf::sca_c, sca_eln::sca_tdf_c

Description

Variable capacitor controlled by a TDF input signal.

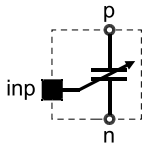
Definition

```
sca_eln::sca_tdf::sca_c( nm, scale, q0 );
```



```
sca_eln::sca_tdf_c( nm, scale, q0 );
```

Symbol



Equation

$$i_{p,n}(t) = scale \cdot \frac{d(inp \cdot v_{p,n}(t) + q_0)}{dt}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient
q0	double	0.0	Initial charge in Coulomb

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
inp	**sca_tdf::sca_in<T>**	double	TDF control input

A.6.16. sca_eln::sca_tdf::sca_l, sca_eln::sca_tdf_l

Description

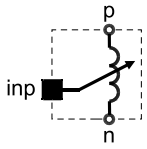
Variable inductor controlled by a TDF input signal.

Definition

```
sca_eln::sca_tdf::sca_l( nm, scale, phi0 );

sca_eln::sca_tdf_l( nm, scale, phi0 );
```

Symbol



Equation

$$v_{p,n}(t) = scale \cdot \frac{d(inp \cdot i_{p,n}(t) + phi_0)}{dt}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient

Name	Type	Default	Description
phi0	double	0.0	Initial magnetic flux in Weber

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
inp	**sca_tdf::sca_in<T>**	double	TDF control input

A.6.17. sca_eln::sca_tdf::sca_rswitch, sca_eln::sca_tdf_rswitch

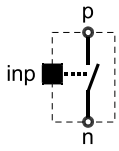
Description

Switch controlled by a TDF input signal.

Definition

```
sca_eln::sca_tdf::sca_rswitch( nm, ron, roff, off_state );
sca_eln::sca_tdf_rswitch( nm, ron, roff, off_state );
```

Symbol



Equation

$$v_{p,n}(t) = \begin{cases} r_{on} \cdot i_{p,n}(t) & ctrl \neq off\_state \\ r_{off} \cdot i_{p,n}(t) & ctrl = off\_state \end{cases}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
ron	double	0.0	On resistance in Ohm
roff	double	**sca_util::SCA_INFINITY**	Off resistance in Ohm
off_state	bool	false	Define which position is the off-position

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
ctrl	**sca_tdf::sca_in<T>**	bool	TDF control input

A.6.18. sca_eln::sca_tdf::sca_vsource, sca_eln::sca_tdf_vsource

Description

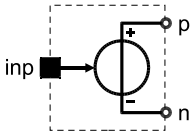
Voltage source driven by a TDF input signal.

Definition

```
sca_eln::sca_tdf::sca_vsource( nm, scale );

sca_eln::sca_tdf_vsource( nm, scale );
```

Symbol



Equation

$v_{p,n}(t) = scale \cdot inp$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
inp	**sca_tdf::sca_in<T>**	double	TDF input

A.6.19. sca_eln::sca_tdf::sca_isource, sca_eln::sca_tdf_isource

Description

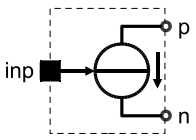
Current source driven by a TDF input signal.

Definition

```
sca_eln::sca_tdf::sca_isource( nm, scale );

sca_eln::sca_tdf_isource( nm, scale );
```

Symbol



Equation

$i_{p,n}(t) = scale \cdot inp$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name

Name	Type	Default	Description
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
inp	**sca_tdf::sca_in<T>**	double	TDF input

A.6.20. sca_eln::sca_tdf::sca_vsink, sca_eln::sca_tdf_vsink

Description

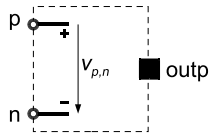
Converts voltage to a TDF output signal.

Definition

```
sca_eln::sca_tdf::sca_vsink( nm, scale );

sca_eln::sca_tdf_vsink( nm, scale );
```

Symbol



Equation

No equation added to the equation system.

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
outp	**sca_tdf::sca_out<T>**	double	TDF output

A.6.21. sca_eln::sca_tdf::sca_isink, sca_eln::sca_tdf_isink

Description

Converts current to a TDF output signal.

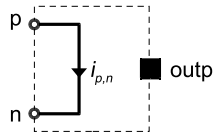
Definition

```
sca_eln::sca_tdf::sca_isink( nm, scale );
```



```
sca_eln::sca_tdf_isink( nm, scale );
```

Symbol



Equation

$v_{p,n}(t) = 0$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
outp	**sca_tdf::sca_out<T>**	double	TDF output

A.6.22. sca_eln::sca_de::sca_r, sca_eln::sca_de_r

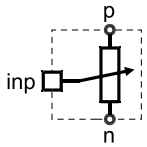
Description

Variable resistor controlled by a discrete-event input signal.

Definition

```
sca_eln::sca_de::sca_r( nm, scale );
sca_eln::sca_de_r( nm, scale );
```

Symbol



Equation

$v_{p,n}(t) = scale \cdot inp \cdot i_{p,n}(t)$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
inp	**sc_core::sc_in<T>**	double	Discrete-event control input

A.6.23. sca_eln::sca_de::sca_c, sca_eln::sca_de_c

Description

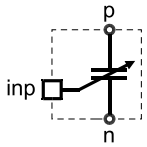
Variable capacitor controlled by a discrete-event input signal.

Definition

```
sca_eln::sca_de::sca_c( nm, scale, q0 );

sca_eln::sca_de_c( nm, scale, q0);
```

Symbol



Equation

$$i_{p,n}(t) = scale \cdot \frac{d\left(inp \cdot v_{p,n}(t) + q_0\right)}{dt}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient
q0	double	0.0	Initial charge in Coulomb

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
inp	**sc_core::sc_in<T>**	double	Discrete-event control input

A.6.24. sca_eln::sca_de::sca_l, sca_eln::sca_de_l

Description

Variable inductor controlled by a discrete-event input signal.

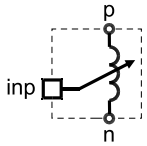
Definition

```
sca_eln::sca_de::sca_l( nm, scale, phi0 );
```



```
sca_eln::sca_de_l( nm, scale, phi0 );
```

Symbol



Equation

$$v_{p,n}(t) = scale \cdot \frac{d(inp \cdot i_{p,n}(t) + phi_0)}{dt}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient
phi0	double	0.0	Initial magnetic flux in Weber

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
inp	**sc_core::sc_in<T>**	double	Discrete-event control input

A.6.25. sca_eln::sca_de::sca_rswitch, sca_eln::sca_de_rswitch

Description

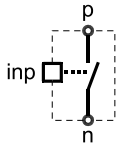
Switch controlled by a discrete-event input signal.

Definition

```
sca_eln::sca_de::sca_rswitch( nm, ron, roff, off_state );

sca_eln::sca_de_rswitch( nm, ron, roff, off_state );
```

Symbol



Equation

$$v_{p,n}(t) = \begin{cases} r_{on} \cdot i_{p,n}(t) & ctrl \neq off\_state \\ r_{off} \cdot i_{p,n}(t) & ctrl = off\_state \end{cases}$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name

Name	Type	Default	Description
ron	double	0.0	On resistance in Ohm
roff	double	**sca_util::SCA_INFINITY**	Off resistance in Ohm
off_state	bool	false	Define which position is the off-position

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
ctrl	**sc_core::sc_in<T>**	bool	Discrete-event control input

A.6.26. sca_eln::sca_de::sca_vsource, sca_eln::sca_de_vsource

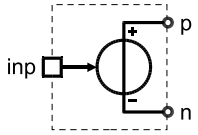
Description

Voltage source driven by a discrete-event input signal.

Definition

```
sca_eln::sca_de::sca_vsource( nm, scale );
sca_eln::sca_de_vsource( nm, scale );
```

Symbol



Equation

$v_{p,n}(t) = scale \cdot inp$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
inp	**sc_core::sc_in<T>**	double	Discrete-event input

A.6.27. sca_eln::sca_de::sca_isource, sca_eln::sca_de_isource

Description

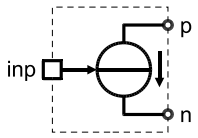
Current source driven by a discrete-event input signal.

Definition

```
sca_eln::sca_de::sca_isource( nm, scale );

sca_eln::sca_de_isource( nm, scale );
```

Symbol



Equation

$i_{p,n}(t) = scale \cdot inp$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
inp	**sc_core::sc_in<T>**	double	Discrete-event input

A.6.28. sca_eln::sca_de::sca_vsink, sca_eln::sca_de_vsink

Description

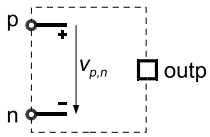
Converts voltage to a discrete-event output signal.

Definition

```
sca_eln::sca_de::sca_vsink( nm, scale );

sca_eln::sca_de_vsink( nm, scale );
```

Symbol



Equation

No equation added to the equation system.

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
outp	**sc_core::sc_out<T>**	double	Discrete-event output

A.6.29. sca_eln::sca_de::sca_isink, sca_eln::sca_de_isink

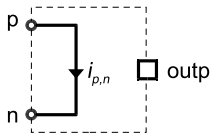
Description

Converts current to a discrete-event output signal.

Definition

```
sca_eln::sca_de::sca_isink( nm, scale );
sca_eln::sca_de_isink( nm, scale );
```

Symbol



Equation

$$v_{p,n}(t) = 0$$

Parameters

Name	Type	Default	Description
nm	**sc_core::sc_module_name**		Module name
scale	double	1.0	Scale coefficient

Ports

Name	Interface	Type/Nature	Description
p	**sca_eln::sca_terminal**	Electrical	Positive terminal
n	**sca_eln::sca_terminal**	Electrical	Negative terminal
outp	**sc_core::sc_out<T>**	double	Discrete-event output

Appendix B. Symbols and graphical representations

This appendix gives an overview of the symbols and graphical representations used in this user's guide. In case derivative block diagrams or electrical networks are extracted from this user's guide, it is strongly recommended to use these symbols and graphical representations in a consistent manner.

The symbols for the individual LSF and ELN primitives are given in Appendix A.

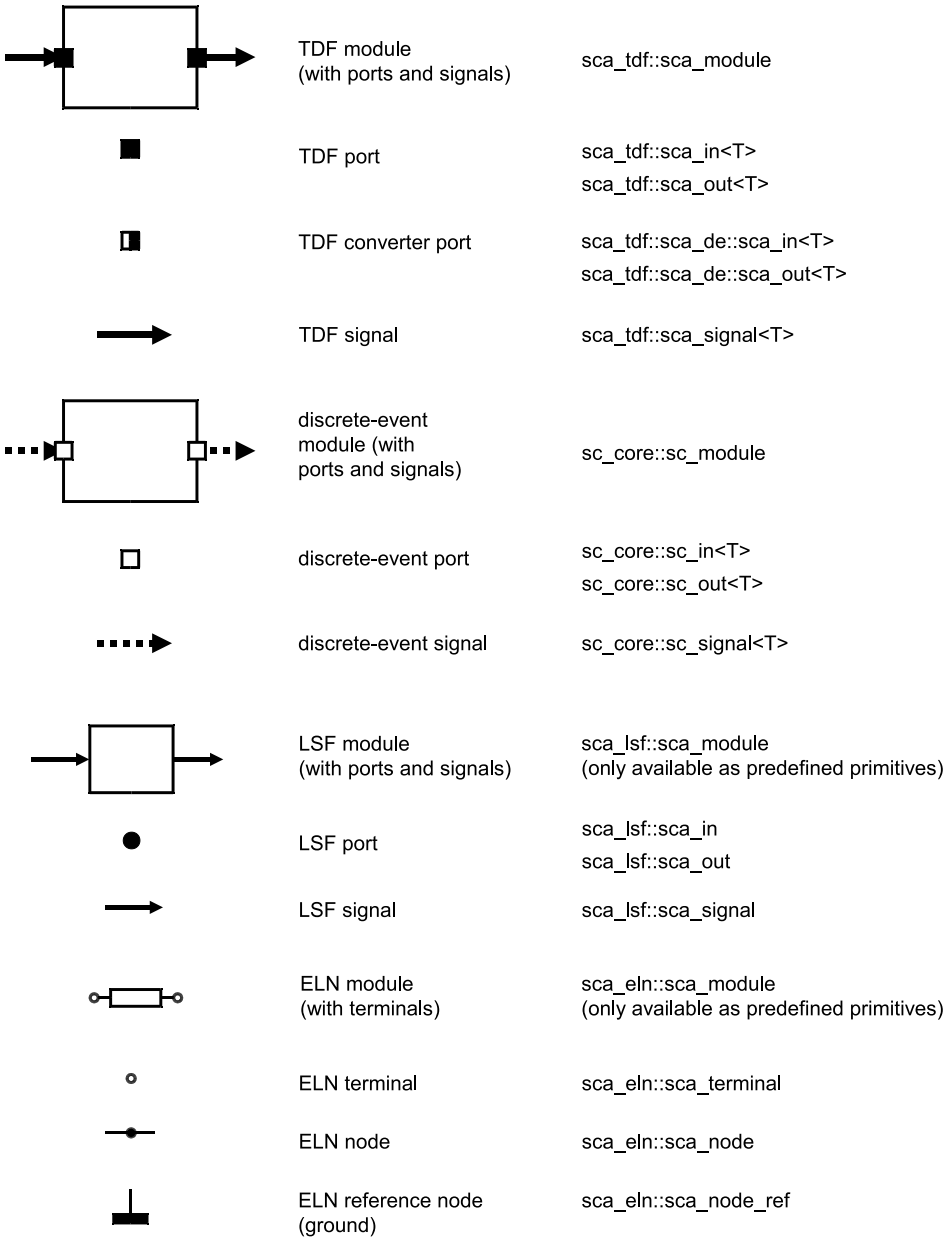


Figure B.1. Symbols and graphical representations of TDF, LSF, ELN and discrete-event elements.

This page is intentionally left blank.

Appendix C. Glossary

This glossary contains brief descriptions for a number of terms used in this user's guide.

C.1. application

A C++ program, written by an end user, that makes use of the classes, functions, macros, and so forth provided by SystemC and the AMS extensions. An application may use as few or as many features of C++ as is seen fit and as few or as many features of SystemC and the AMS extensions as is seen fit.

C.2. cluster

A cluster is a set of connected modules sharing the same model of computation.

C.3. continuous-time signal

A continuous-time signal is a piecewise contiguous and differentiable signal, which may be represented in approximation by a set of samples at discrete time points. Values between the samples can be estimated by different interpolation techniques.

C.4. discrete-time signal

A discrete-time signal is a signal that has been sampled from a continuous-time signal resulting in a sequence of values at discrete time points. Each value in the sequence is called a *sample*.

C.5. electrical linear networks, ELN

A model of computation that uses the electrical linear networks formalism for calculations. (See Chapter 4.)

C.6. frequency-domain processing

Frequency-domain processing can be embedded in timed data flow descriptions for analysis of small-signal frequency-domain behavior. The frequency-domain behavior of a module instance derived from **sca_tdf::sca_module** has to be implemented either by overloading its member function **sca_tdf::sca_module::ac_processing** or by registering an application-defined member function using **sca_tdf::sca_module::register_ac_processing**.

C.7. hierarchical port

A port of a parent module.

C.8. implementation

A specific concrete implementation of the full SystemC AMS extensions, of which only the public shell needs to be exposed to the application (i.e., parts may be pre-compiled and distributed as object code by a tool vendor).

C.9. linear signal flow, LSF

A model of computation that uses the linear signal flow formalism for calculations and signal processing. (See Chapter 3.)

C.10. model of computation, MoC

A *model of computation* implements a modeling formalism, which is a set of rules defining the behavior (computation) and interaction (communication) between AMS primitive modules instantiated within a module. (See Section 1.2.3.)

C.11. numerically singular

Numerically singular describes a situation, in which the solution of an equation system cannot be calculated.

C.12. primitive module

A class that is derived from class **sca_core::sca_module** and complies to a particular model of computation. A primitive module cannot be hierarchically decomposed and contains no child modules or channels.

C.13. primitive port

A port of a primitive module.

C.14. proxy class

A class, which only purpose is to extend the readability of certain statements that otherwise would be restricted by the semantics of C++. An example is to use the proxy class to represent a *continuous-time signal* and to map it to *discrete-time signal*. Proxy classes are only intended to be used for the temporary value returned by a function. A proxy class constructor shall not be called explicitly by an application to create a named object.

C.15. rate

The rate defines the number of samples that have to be read or written at a port of type **sca_tdf::sca_in**, **sca_tdf::sca_out**, **sca_tdf::sca_de::sca_in**, and **sca_tdf::sca_de::sca_out** during each execution of the time-domain and frequency domain processing function of its parent module derived from **sca_tdf::sca_module**. The rate of such a port shall have a positive, nonzero value.

C.16. sample

A sample refers to a value at a certain point in time or refers to a set of values with a certain start and end time. *sample_id* denotes the index of the (data) sample, *nsample* denotes the number of samples in a set of values.

C.17. solver

A solver computes the solution of an equation system (e.g., a set of differential and algebraic equations).

C.18. terminal

A terminal is a class derived from the class **sca_core::sca_port** and is associated with the electrical linear networks model of computation. For electrical primitives with 2 terminals, the terminal names *p* and *n* are defined. Multi-port primitives may use different terminal names.

C.19. timed data flow, TDF

A model of computation that uses the timed data flow formalism for scheduling and signal processing. (See Chapter 2.)

C.20. time-domain processing

Time-domain processing is done through the repetitive activation of the time-domain processing member functions as part of the timed data flow model of computation. The time-domain processing member function can be either the member function **sca_tdf::sca_module::processing** or an application-defined member function, which shall be registered using the member function **sca_tdf::sca_module::register_processing**.

C.21. untimed model of computation

In an untimed model of computation, the behavioral description (computation) and interaction with other modules and processes (communication) does not have a notion of time. Only the order of computations or events, and cause and effect of computations or events are relevant.

Index

A

abstraction, 85
ac_processing, member function
  class sca_tdf::sca_module, 105
application
  glossary, 145
attribute settings
  set_attributes, member function, 15
  timed data flow, 15

B

baseband modeling
  timed data flow, 89
behavioral modeling
  linear signal flow, 87
  timed data flow, 89

C

classes
  public and private members, 102
  sca_eln::sca_c, 56, 123
  sca_eln::sca_cccs, 56, 126
  sca_eln::sca_ccvs, 56, 125
  sca_eln::sca_de::sca_c, 138
  sca_eln::sca_de::sca_c, class, 56
  sca_eln::sca_de::sca_isink, 56, 142
  sca_eln::sca_de::sca_isource, 56, 140
  sca_eln::sca_de::sca_l, 56, 138
  sca_eln::sca_de::sca_r, 56, 137
  sca_eln::sca_de::sca_rswitch, 56, 139
  sca_eln::sca_de::sca_vsink, 56, 141
  sca_eln::sca_de::sca_vsource, 56, 140
  sca_eln::sca_gyrator, 56, 127
  sca_eln::sca_ideal_transformer, 56, 128
  sca_eln::sca_isource, 56, 131
  sca_eln::sca_l, 56, 123
  sca_eln::sca_node, 58
  sca_eln::sca_node_ref, 58
  sca_eln::sca_nullor, 56, 127
  sca_eln::sca_r, 56, 122
  sca_eln::sca_tdf::sca_c, 56, 132
  sca_eln::sca_tdf::sca_isink, 56, 136
  sca_eln::sca_tdf::sca_isource, 56, 135
  sca_eln::sca_tdf::sca_l, 56, 133
  sca_eln::sca_tdf::sca_r, 56, 132
  sca_eln::sca_tdf::sca_rswitch, 56, 134
  sca_eln::sca_tdf::sca_vsink, 56, 136
  sca_eln::sca_tdf::sca_vsource, 56, 134
  sca_eln::sca_terminal, 58
  sca_eln::sca_transmission_line, 56, 129
  sca_eln::sca_vccs, 56, 125
  sca_eln::sca_vcvs, 56, 124
  sca_eln::sca_vsource, 56, 130
  sca_lsf::sca_add, 42, 108
  sca_lsf::sca_de::sca_demux, 42, 121
  sca_lsf::sca_de::sca_gain, 42, 119
  sca_lsf::sca_de::sca_mux, 42, 121
  sca_lsf::sca_de::sca_sink, 42, 120
  sca_lsf::sca_de::sca_source, 42, 119
  sca_lsf::sca_delay, 42, 111
  sca_lsf::sca_dot, 42, 110
  sca_lsf::sca_gain, 42, 110
  sca_lsf::sca_in, 43
  sca_lsf::sca_integ, 42, 111
  sca_lsf::sca_ltf_nd, 42, 113
  sca_lsf::sca_ltf_zp, 42, 114
  sca_lsf::sca_out, 43
  sca_lsf::sca_signal, 44
  sca_lsf::sca_source, 42, 112
  sca_lsf::sca_ss, 42, 115
  sca_lsf::sca_sub, 42, 109
  sca_lsf::sca_tdf::sca_demux, 42, 118
  sca_lsf::sca_tdf::sca_gain, 42, 116
  sca_lsf::sca_tdf::sca_mux, 42, 117
  sca_lsf::sca_tdf::sca_sink, 42, 117
  sca_lsf::sca_tdf::sca_source, 42, 116
  sca_tdf::sca_de::sca_in, 17, 105
  sca_tdf::sca_de::sca_out, 17, 105
  sca_tdf::sca_in, 17, 105
  sca_tdf::sca_ltf_nd, 22, 106
  sca_tdf::sca_ltf_zp, 22, 107
  sca_tdf::sca_module, 105
  sca_tdf::sca_out, 17, 105
  sca_tdf::sca_signal, 20, 106
  sca_tdf::sca_ss, 23, 107
  sca_tdf::sca_trace_variable, 80
  sca_util::sca_trace_file, 79
cluster, glossary, 145
coding style, 96
complex
  envelope, 91
  low-pass equivalent, 91
  value, sca_util::SCA_COMPLEX_J, 73
constants
  sca_util::SCA_COMPLEX_J, 73
constructor
  SCA_CTOR, macro, 16
continuous-time modeling
  electrical linear networks, 60
  linear signal flow, 46
  timed data flow, 21
continuous-time signal, glossary, 145
converter modules
  electrical linear networks, 61
  linear signal flow, 47
converter ports
  timed data flow, 17, 30

D

delay
  frequency-domain delay, 71
delays
  set_delay, member function, 17, 29
  timed data flow, 29

design refinement, 94
disable, member function
  class sca_util::sca_trace_file, 79
discrete-time modeling
  timed data flow, 21
discrete-time signal, glossary, 145
dynamic memory allocation, 97

E

electrical linear networks
  continuous-time modeling, 60
  glossary, 145
  language constructs, 56
  macromodeling, 86
  modeling fundamentals, 55
  module time step, 57
  port binding, 59
  set_timestep, member function, 57
  setup equation system , 55
  structural composition of modules, 59
  time step assignment, 56
  time step propagation, 56
ELN (see electrical linear networks)
embedded equations
  laplace transfer functions, 22
  state-space equations, 23
enable, member function
  class sca_util::sca_trace_file, 79
execution semantics
  electrical linear networks, 64
  linear signal flow, 50
  timed data flow, 32

F

frequency-domain processing, glossary, 145
frequency-domain simulation, 78
functions
  sc_core::sc_ac_noise_start, 78
  sc_core::sc_ac_start, 78
  sc_core::sc_start, 77
  sca_ac_analysis::sca_ac, 70
  sca_ac_analysis::sca_ac_delay, 71
  sca_ac_analysis::sca_ac_f, 73
  sca_ac_analysis::sca_ac_is_running, 74
  sca_ac_analysis::sca_ac_ltf_nd, 71
  sca_ac_analysis::sca_ac_ltf_zp, 71
  sca_ac_analysis::sca_ac_noise, 70
  sca_ac_analysis::sca_ac_noise_is_running, 74
  sca_ac_analysis::sca_ac_s, 72
  sca_ac_analysis::sca_ac_w, 73
  sca_ac_analysis::sca_ac_z, 73
  sca_util::sca_close_tabular_trace_file, 79
  sca_util::sca_close_vcd_trace_file, 79
  sca_util::sca_create_tabular_trace_file, 79
  sca_util::sca_create_vcd_trace_file, 79
  sca_util::sca_decimation, 79
  sca_util::sca_sampling, 79
  sca_util::sca_trace, 80
  sca_util::sca_write_comment, 80
  small-signal frequency-domain, 71

G

get_delay, member function
  class sca_tdf::sca_de::sca_in, 17, 105
  class sca_tdf::sca_de::sca_out, 17, 105
  class sca_tdf::sca_in, 17, 105
  class sca_tdf::sca_out, 17, 105
get_rate, member function
  class sca_tdf::sca_de::sca_in, 17, 105
  class sca_tdf::sca_de::sca_out, 17, 105
  class sca_tdf::sca_in, 17, 105
  class sca_tdf::sca_out, 17, 105
get_time, member function
  class sca_tdf::sca_module, 105
get_timeoffset, member function
  class sca_tdf::sca_de::sca_in, 17, 105
  class sca_tdf::sca_de::sca_out, 17, 105
get_timestep, member function
  class sca_tdf::sca_de::sca_in, 17, 105
  class sca_tdf::sca_de::sca_out, 17, 105
  class sca_tdf::sca_in, 17, 105
  class sca_tdf::sca_module, 105
  class sca_tdf::sca_out, 17, 105

H

header files, 96
hierarchical port, glossary, 145

I

implementation
  glossary, 145
initialization
  initialize, member function, 15
  timed data flow, 15
initialize, member function
  class sca_tdf::sca_de::sca_in, 18, 105
  class sca_tdf::sca_de::sca_out, 18, 105
  class sca_tdf::sca_in, 18, 105
  class sca_tdf::sca_module, 105
  class sca_tdf::sca_out, 18, 105
interaction between models of computation
  electrical linear networks, 61
  linear signal flow, 47
  timed data flow, 30
introduction
  motivation, 1

L

language architecture, 4
language constructs
  electrical linear networks, 56
  linear signal flow, 42
  small-signal frequency-domain, 70
  time data flow, 14
laplace transfer functions
  class sca_lsf::sca_ltf_nd, 113

class sca_lsf::sca_ltf_zp, 114
class sca_tdf::sca_ltf_nd, 22, 106
class sca_tdf::sca_ltf_zp, 22, 107
function sca_ac_analysis::sca_ac_ltf_nd, 71
function sca_ac_analysis::sca_ac_ltf_zp, 71
linear signal flow
continuous-time modeling, 46
glossary, 145
language constructs, 42
modeling fundamentals, 41
module time step, 43
port binding, 44
set_timestep, member function, 43
setup equation system , 41
structural composition of modules, 44
time step assignment, 42
time step propagation, 42
LSF (see linear signal flow)
LTF (see laplace transfer functions)

M

macromodeling, 86
model abstractions
conservative descriptions, 3
continuous-time descriptions, 3
discrete-time descriptions, 3
non-conservative descriptions, 3
modeling formalisms
electrical linear networks, 4
linear signal flow, 4
timed data flow, 4
modeling style, 96
model of computation
electrical linear networks, 55,
glossary, 145
linear signal flow, 41,
partitioning behavior, 91
timed data flow, 7,
untimed, glossary, 146
module activation
processing, member function, 16, 105
module local time
get_time, member function, 16
modules
class sca_eln::sca_module, 56
class sca_lsf::sca_module, 42
class sca_tdf::sca_module, 14, 105
definition and implementation, 100
electrical linear networks, 56
linear signal flow, 42
parameters, 98
time data flow, 14
module time step
get_timestep, member function, 105
set_timestep, member function, 15, 43, 57, 105
multirate behavior
get_rate, member function, 17, 105
set_rate, member function, 17, 28, 105
timed data flow, 28

N

namespaces, 96
naming conventions, 96
nodes
class sca_eln::sca_node, 58
class sca_eln::sca_node_ref, 58
electrical linear networks, 58
noise
modeling in time-domain, 93
numerically singular, glossary, 145

P

PID controller
with adjustable coefficients, 88
port attributes
get_delay, member function, 17, 105
get_rate, member function, 17, 105
get_timeoffset, member function, 17, 105
get_timestep, member function, 17, 105
set_delay, member function, 17, 105
set_rate, member function, 17, 105
set_timeoffset, member function, 17, 105
set_timestep, member function, 17, 105
timed data flow, 17, 105
port binding
electrical linear networks, 59
linear signal flow, 44
timed data flow, 26
port initialization
initialize, member function, 18, 105
port read and write access
read, member function, 18, 105
write, member function, 18, 105
ports
class sca_eln::sca_terminal, 58
class sca_lsf::sca_in, 43
class sca_lsf::sca_out, 43
class sca_tdf::sca_de::sca_in, 17, 105
class sca_tdf::sca_de::sca_out, 17, 105
class sca_tdf::sca_in, 17, 105
class sca_tdf::sca_out, 17, 105
electrical linear networks terminals, 58
linear signal flow, 43
timed data flow, 17
port time
get_time, member function, 20, 105
get_timeoffset, member function, 17
set_timeoffset, member function, 17, 105
port time step
get_timestep, member function, 17, 105
set_timestep, member function, 17, 105
primitive modules
class sca_eln::sca_c, 56, 123
class sca_eln::sca_cccs, 56, 126
class sca_eln::sca_ccvs, 56, 125
class sca_eln::sca_de::sca_c, 138

class sca_eln::sca_de::sca_c, class, 56
class sca_eln::sca_de::sca_isink, 56, 142
class sca_eln::sca_de::sca_isource, 56, 140
class sca_eln::sca_de::sca_l, 56, 138
class sca_eln::sca_de::sca_r, 56, 137
class sca_eln::sca_de::sca_rswitch, 56, 139
class sca_eln::sca_de::sca_vsink, 56, 141
class sca_eln::sca_de::sca_vsource, 56, 140
class sca_eln::sca_gyrator, 56, 127
class sca_eln::sca_ideal_transformer, 56, 128
class sca_eln::sca_isource, 56, 131
class sca_eln::sca_l, 56, 123
class sca_eln::sca_nullor, 56, 127
class sca_eln::sca_r, 56, 122
class sca_eln::sca_tdf::sca_c, 56, 132
class sca_eln::sca_tdf::sca_isink, 56, 136
class sca_eln::sca_tdf::sca_isource, 56, 135
class sca_eln::sca_tdf::sca_l, 56, 133
class sca_eln::sca_tdf::sca_r, 56, 132
class sca_eln::sca_tdf::sca_rswitch, 56, 134
class sca_eln::sca_tdf::sca_vsink, 56, 136
class sca_eln::sca_tdf::sca_vsource, 56, 134
class sca_eln::sca_transmission_line, 56, 129
class sca_eln::sca_vccs, 56, 125
class sca_eln::sca_vcvs, 56, 124
class sca_eln::sca_vsource, 56, 130
class sca_lsf::sca_add, 42, 108
class sca_lsf::sca_de::sca_demux, 42, 121
class sca_lsf::sca_de::sca_gain, 42, 119
class sca_lsf::sca_de::sca_mux, 42, 121
class sca_lsf::sca_de::sca_sink, 42, 120
class sca_lsf::sca_de::sca_source, 42, 119
class sca_lsf::sca_delay, 42, 111
class sca_lsf::sca_dot, 42, 110
class sca_lsf::sca_gain, 42, 110
class sca_lsf::sca_integ, 42, 111
class sca_lsf::sca_ltf_nd, 42, 113
class sca_lsf::sca_ltf_zp, 42, 114
class sca_lsf::sca_source, 42, 112
class sca_lsf::sca_ss, 42, 115
class sca_lsf::sca_sub, 42, 109
class sca_lsf::sca_tdf::sca_demux, 42, 118
class sca_lsf::sca_tdf::sca_gain, 42, 116
class sca_lsf::sca_tdf::sca_mux, 42, 117
class sca_lsf::sca_tdf::sca_sink, 42, 117
class sca_lsf::sca_tdf::sca_source, 42, 116
class sca_tdf::sca_module, 14
class sca_tdf::sca_ss, 107
electrical linear networks, 56
glossary, 146
linear signal flow, 42
time data flow, 14
primitive port, glossary, 146
processing, member function
class sca_tdf::sca_module, 105
proxy class, glossary, 146

R

rate, glossary, 146
read, member function
class sca_tdf::sca_de::sca_in, 18, 105
class sca_tdf::sca_in, 18, 105
reopen, member function
class sca_util::sca_trace_file, 79

S

sample, glossary, 146
sc_core::sc_ac_noise_start, function, 78
sc_core::sc_ac_start, function, 78
sc_core::sc_start, function, 77
sca_ac_analysis::sca_ac_delay, function, 71
sca_ac_analysis::sca_ac_f, function, 73
sca_ac_analysis::sca_ac_is_running, function, 74
sca_ac_analysis::sca_ac_ltf_nd, function, 71
sca_ac_analysis::sca_ac_ltf_zp, function, 71
sca_ac_analysis::sca_ac_noise_is_running, function, 74
sca_ac_analysis::sca_ac_noise, function, 70
sca_ac_analysis::sca_ac_s, function, 72
sca_ac_analysis::sca_ac_w, function, 73
sca_ac_analysis::sca_ac_z, function, 73
sca_ac_analysis::sca_ac, function, 70
SCA_CTOR, macro
class sca_tdf::sca_module, 105
sca_eln::sca_c, class, 123
sca_eln::sca_cccs, class, 126
sca_eln::sca_ccvs, class, 125
sca_eln::sca_de_c, typedef
class sca_eln::sca_de::sca_c, 138
sca_eln::sca_de_isink, typedef
class sca_eln::sca_de::sca_isink, 142
sca_eln::sca_de_isource, typedef
class sca_eln::sca_de::sca_isource, 140
sca_eln::sca_de_l, typedef
class sca_eln::sca_de::sca_l, 138
sca_eln::sca_de_r, typedef
class sca_eln::sca_de::sca_r, 137
sca_eln::sca_de_rswitch, typedef
class sca_eln::sca_de::sca_rswitch, 139
sca_eln::sca_de_vsink, typedef
class sca_eln::sca_de::sca_vsink, 141
sca_eln::sca_de_vsource, typedef
class sca_eln::sca_de::sca_vsource, 140
sca_eln::sca_de::sca_c, class, 138
sca_eln::sca_de::sca_isink, class, 142
sca_eln::sca_de::sca_isource, class, 140
sca_eln::sca_de::sca_l, class, 138
sca_eln::sca_de::sca_r, class, 137
sca_eln::sca_de::sca_rswitch, class, 139
sca_eln::sca_de::sca_vsink, class, 141
sca_eln::sca_de::sca_vsource, class, 140
sca_eln::sca_gyrator, class, 127
sca_eln::sca_ideal_transformer, class, 128
sca_eln::sca_isource, class, 131
sca_eln::sca_l, class, 123

sca_eln::sca_nullor, class, 127
sca_eln::sca_r, class, 122
sca_eln::sca_tdf_c, typedef
class sca_eln::sca_tdf::sca_c, 132
sca_eln::sca_tdf_isink, typedef
class sca_eln::sca_tdf::sca_isink, 136
sca_eln::sca_tdf_isource, typedef
class sca_eln::sca_tdf::sca_isource, 135
sca_eln::sca_tdf_l, typedef
class sca_eln::sca_tdf::sca_l, 133
sca_eln::sca_tdf_r, typedef
class sca_eln::sca_tdf::sca_r, 132
sca_eln::sca_tdf_rswitch, typedef
class sca_eln::sca_tdf::sca_rswitch, 134
sca_eln::sca_tdf_vsink, typedef
class sca_eln::sca_tdf::sca_vsink, 136
sca_eln::sca_tdf_vsource, typedef
class sca_eln::sca_tdf::sca_vsource, 134
sca_eln::sca_tdf::sca_c, class, 132
sca_eln::sca_tdf::sca_isink, class, 136
sca_eln::sca_tdf::sca_isource, class, 135
sca_eln::sca_tdf::sca_l, class, 133
sca_eln::sca_tdf::sca_r, class, 132
sca_eln::sca_tdf::sca_rswitch, class, 134
sca_eln::sca_tdf::sca_vsink, class, 136
sca_eln::sca_tdf::sca_vsource, class, 134
sca_eln::sca_transmission_line, class, 129
sca_eln::sca_vccs, class, 125
sca_eln::sca_vcvs, class, 124
sca_eln::sca_vsource, class, 130
sca_lsf::sc_de_gain, typedef
class sca_lsf::sca_de::sca_gain, 119
sca_lsf::sca_add, class, 108
sca_lsf::sca_de_demux, typedef
class sca_lsf::sca_de::sca_demux, 121
sca_lsf::sca_de_mux, typedef
class sca_lsf::sca_de::sca_mux, 121
sca_lsf::sca_de_sink, typedef
class sca_lsf::sca_de::sca_sink, 120
sca_lsf::sca_de_source, typedef
class sca_lsf::sca_de::sca_source, 119
sca_lsf::sca_de::sca_demux, class, 121
sca_lsf::sca_de::sca_gain, class, 119
sca_lsf::sca_de::sca_mux, class, 121
sca_lsf::sca_de::sca_sink, class, 120
sca_lsf::sca_de::sca_source, class, 119
sca_lsf::sca_delay, class, 111
sca_lsf::sca_dot, class, 110
sca_lsf::sca_gain, class, 110
sca_lsf::sca_integ, class, 111
sca_lsf::sca_ltf_nd, class, 113
sca_lsf::sca_ltf_zp, class, 114
sca_lsf::sca_source, class, 112
sca_lsf::sca_ss, class, 115
sca_lsf::sca_sub, class, 109
sca_lsf::sca_tdf_demux, typedef
class sca_lsf::sca_tdf::sca_demux, 118
sca_lsf::sca_tdf_gain, typedef
class sca_lsf::sca_tdf::sca_gain, 116
sca_lsf::sca_tdf_mux, typedef
class sca_lsf::sca_tdf::sca_mux, 117
sca_lsf::sca_tdf_sink, typedef
class sca_lsf::sca_tdf::sca_sink, 117
sca_lsf::sca_tdf_source, typedef
class sca_lsf::sca_tdf::sca_source, 116
sca_lsf::sca_tdf::sca_demux, class, 118
sca_lsf::sca_tdf::sca_gain, class, 116
sca_lsf::sca_tdf::sca_mux, class, 117
sca_lsf::sca_tdf::sca_sink, class, 117
sca_lsf::sca_tdf::sca_source, class, 116
SCA_TDF_MODULE, macro
class sca_tdf::sca_module, 14
sca_tdf::sca_ltf_nd, class, 106
sca_tdf::sca_ltf_zp, class, 107
sca_tdf::sca_module, class, 105
sca_tdf::sca_ss, class, 107
sca_util::sca_close_tabular_trace_file, function, 79
sca_util::sca_close_vcd_trace_file, function, 79
sca_util::SCA_COMPLEX_J, constant, 73
sca_util::sca_create_tabular_trace_file, function, 79
sca_util::sca_create_vcd_trace_file, function, 79
sca_util::sca_decimation, function, 79
sca_util::sca_sampling, function, 79
sca_util::sca_write_comment, function, 80
S-domain
function sca_ac_analysis::sca_ac_s, 72
set_attributes, member function
class sca_tdf::sca_module, 105
set_delay, member function
class sca_tdf::sca_de::sca_in, 17, 105
class sca_tdf::sca_de::sca_out, 17, 105
class sca_tdf::sca_in, 17, 105
class sca_tdf::sca_out, 17, 105
set_mode, member function
class sca_util::sca_trace_file, 79
set_rate, member function
class sca_tdf::sca_de::sca_in, 17, 105
class sca_tdf::sca_de::sca_out, 17, 105
class sca_tdf::sca_in, 17, 105
class sca_tdf::sca_out, 17, 105
set_timeoffset, member function
class sca_tdf::sca_de::sca_in, 17, 105
class sca_tdf::sca_de::sca_out, 17, 105
set_timestep, member function
class sca_eln::sca_module, 57
class sca_lsf::sca_module, 43
class sca_tdf::sca_de::sca_in, 17, 105
class sca_tdf::sca_de::sca_out, 17, 105
class sca_tdf::sca_in, 17, 105
class sca_tdf::sca_module, 15, 105
class sca_tdf::sca_out, 17, 105
signals
class sca_eln::sca_node, 58
class sca_eln::sca_node_ref, 58
class sca_lsf::sca_signal, 44
class sca_tdf::sca_signal, 20

electrical linear networks, nodes, 58
linear signal flow, 44
timed data flow, 20
simulation
arguments, 77
frequency-domain, 78
time-domain, transient, 77
simulation control, 77
small-signal frequency-domain
analyses methods, 69
language constructs, 70
modeling fundamentals, 69
setup equation system , 69
solver, glossary, 146
state-space equations
class sca_tdf::sca_ss, 23
state-space function
class sca_lsf::sca_ss, 115
class sca_tdf::sca_ss, 107
structural composition
electrical linear networks, 59
linear signal flow modules, 44
timed data flow modules, 26

T

TDF (see timed data flow)
templates
class templates, 101
terminals
class sca_eln::sca_terminal, 58
electrical linear networks, 58
glossary, 146
testbench, 82
timed data flow
attributes, 7
baseband modeling, 89
behavioral modeling, 89
continuous-time modeling, 21
discrete-time modeling, 21
glossary, 146
language constructs, 14
laplace transfer functions, 22
modeling fundamentals, 7
module attributes, 15
multirate behavior, 28
port binding, 26
signal processing behavior, 13
state-space equations, 23
structural composition of modules, 26
time step assignment, 11
time step propagation, 11
time-domain processing, glossary, 146
time-domain simulation, 77
time step
electrical linear networks modules, 57
linear signal flow modules, 43
timed data flow modules, 15
timed data flow ports, 17
time step assignment and propagation
electrical linear networks, 56
linear signal flow, 42
timed data flow, 11, 89
trace variables
class sca_tdf::sca_trace_variable, 80
tracing
reopen trace file, 79
sca_util::sca_trace, function, 80
signal types, 80
time-domain, 79
to an output stream, 79
to tabular file, 79
to VCD file, 79
trace file control, 79
trace file mode, 79
writing comments, 80
typedef
sca_eln::sca_de_c, 138
sca_eln::sca_de_isink, 142
sca_eln::sca_de_isource, 140
sca_eln::sca_de_l, 138
sca_eln::sca_de_r, 137
sca_eln::sca_de_rswitch, 139
sca_eln::sca_de_vsink, 141
sca_eln::sca_de_vsource, 140
sca_eln::sca_tdf_c, 132
sca_eln::sca_tdf_isink, 136
sca_eln::sca_tdf_isource, 135
sca_eln::sca_tdf_l, 133
sca_eln::sca_tdf_r, 132
sca_eln::sca_tdf_rswitch, 134
sca_eln::sca_tdf_vsink, 136
sca_eln::sca_tdf_vsource, 134
sca_lsf::sca_de_demux, 121
sca_lsf::sca_de_gain, 119
sca_lsf::sca_de_mux, 121
sca_lsf::sca_de_sink, 120
sca_lsf::sca_de_source, 119
sca_lsf::sca_tdf_demux, 118
sca_lsf::sca_tdf_gain, 116
sca_lsf::sca_tdf_mux, 117
sca_lsf::sca_tdf_sink, 117
sca_lsf::sca_tdf_source, 116

U

use cases
architecture exploration, 2
executable specification, 2
integration validation, 3
virtual prototyping, 2
using directive, 97

W

write, member function
class sca_tdf::sca_de::sca_out, 18, 105
class sca_tdf::sca_out, 18, 105

Z

z-domain
function sca_ac_analysis::sca_ac_z, 73

This page is intentionally left blank.