March 8 2010

Standard SystemC AMS extensions Language Reference Manual

Abstract

This is the SystemC Analog Mixed Signal (AMS) extensions Language Reference Manual, version 1.0

Keywords

Open SystemC Initiative, SystemC, Analog Mixed Signal, Hardware Description Language, 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.

Introduction

This document defines the standard for the SystemC AMS extensions.

Contributors

The development of the SystemC AMS extensions Language Reference Manual was sponsored by the Open SystemC Initiative (OSCI) and was created under the leadership of the following people:

Martin Barnasconi, NXP Semiconductors (AMS Working Group Chair)
Karsten Einwich, Fraunhofer IIS/EAS
Christoph Grimm, TU Vienna (AMS Working Group Vice-Chair)
Alain Vachoux, EPFL

AMS Working Group

At the time this 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
Wolfgang Granig, Infineon
Mark Glasser, Mentor Graphics
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.

Contents

This page is intentionally left blank.

1. Overview

1.1. Scope

This standard defines the SystemC AMS extensions as an ANSI standard C++ class library based on SystemC for system design containing analog/mixed-signal (AMS) functionality.

1.2. Purpose

The general purpose of the SystemC AMS extensions is to provide a C++ standard for designers and architects who need to address complex heterogeneous systems that are a hybrid between hardware and software. This standard is built on the IEEE Std 1666-2005 (SystemC Language Reference Manual) and extends it to create analog/mixed-signal, multi-disciplinary models to simulate continuous-time, discrete-time, and discrete-event behavior simultaneously.

The specific purpose of this standard is to provide a precise and complete definition of the AMS class library so that a SystemC AMS implementation can be developed with reference to this standard alone. This standard is not intended to serve as a user's guide nor to provide an introduction to AMS extensions in SystemC, but does contain useful information for end users.

1.3. Subsets

It is anticipated that tool vendors will create implementations that support only a subset of this standard or that impose further constraints on the use of this standard. Such implementations are not fully compliant with this standard but may nevertheless claim partial compliance with this standard and may use the name SystemC AMS extensions.

1.4. Relationship with C++

This standard is closely related to the C++ programming language and adheres to the terminology used in ISO/IEC 14882:2003. This standard does not seek to restrict the usage of the C++ programming language; an application using the SystemC AMS extensions may use any of the facilities provided by C++, which in turn may use any of the facilities provided by C. However, where the facilities provided by this standard are used, they shall be used in accordance with the rules and constraints set out in this standard.

This standard defines the public interface to the SystemC AMS class library and the constraints on how those classes may be used. The SystemC AMS class library may be implemented in any manner whatsoever, provided only that the obligations imposed by this standard are honored.

A C++ class library may be extended using the mechanisms provided by the C++ language. Implementors and users are free to extend SystemC AMS extensions in this way, provided that they do not violate this standard.

1.5. Relationship with SystemC

This standard is built on the IEEE Std 1666-2005 (SystemC Language Reference Manual) and extends it using the mechanisms provided by the C++ language, to provide an additional layer of analog/mixed-signal constructs. Any SystemC compliant application shall behave the same in the presence of the SystemC AMS extensions.

1.6. Guidance for readers

Readers who are not entirely familiar with the SystemC AMS extensions should start with Annex A, "*Introduction to the SystemC AMS extensions*", which provides a brief informal summary of the subject

intended to aid in the understanding of the normative definitions. Such readers may also find it helpful to scan the examples embedded in the normative definitions and to see Annex B, "*Glossary*"

Readers should pay close attention to Clause 2, "*Terminology and conventions used in this standard*". An understanding of the terminology defined in that clause is necessary for a precise interpretation of this standard.

The semantic definitions given in the subsequent clauses detailing the individual classes are built upon the foundations laid in Clause 3, "*Core language definitions*".

The clauses from Clause 4 onward define the public interface to the SystemC AMS class library defining the predefined models of computation. The following information is listed for each class:

a. A brief class description.

b. A C++ source code listing of the class definition.

c. A statement of any constraints on the use of the class and its members.

d. A statement of the semantics of the class and its members.

e. For certain classes, a description of functions, typedefs, macros, and template parameters associated with the class.

For each predefined model of computation, the execution semantics for elaboration and simulation are defined.

Readers should bear in mind that the primary obligation of a tool vendor is to implement the abstract semantics defined in Clause 4, using the framework and constraints provided by the class definitions starting in Clause 3.

Annex A, "*Introduction to the SystemC AMS extensions*" is intended to aid the reader in the understanding of the structure and intent of the SystemC AMS class library.

Annex B, "*Glossary*" is giving informal descriptions of the terms used in this standard, followed by an Index.

1.7. Reference documents

The following documents are indispensable for the applications of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the document (including any amendments or corrigenda) applies.

This standard shall be used in conjunction with the following publications:

- ISO/IEC 14882:2003, Programming Languages—C++
- IEEE Std 1666-2005: IEEE Standard SystemC Language Reference Manual

2. Terminology and conventions used in this standard

2.1. Terminology

2.1.1. Shall, should, may, can

The word *shall* is used to indicate a mandatory requirement.

The word *should* is used to recommend a particular course of action, but does not impose any obligation.

The word *may* is used to mean shall be permitted (in the sense of being legally allowed).

The word *can* is used to mean shall be able to (in the sense of being technically possible).

In some cases, word usage is qualified to indicate on whom the obligation falls, such as *an application may* or *an implementation shall*.

2.1.2. Implementation, application

The word *implementation* is used to mean any specific implementation of the full SystemC AMS class library as defined in this standard, only the public interface of which need be exposed to the application.

The word *application* is used to mean a C++ program, written by an end user, that uses the SystemC AMS class library, that is, uses classes, functions, or macros defined in this standard.

2.1.3. Call, called from, derived from

The term *call* is taken to mean call directly or indirectly. Call indirectly means call an intermediate function which in turn calls the function in question, where the chain of function calls may be extended indefinitely.

Similarly, *called from* means called from directly or indirectly.

Except where explicitly qualified, the term *derived from* is taken to mean derived directly or indirectly from. Derived indirectly from means derived from one or more intermediate base classes.

2.1.4. Specific technical terms

The specific technical terms as defined in IEEE Std 1666-2005 (SystemC Language Reference Manual) also apply for the AMS extensions. In addition, the following technical terms are defined:

A *model of computation* (MoC) is a set of rules defining the behavior and interaction between AMS primitive modules. The defined models of computation in this standard are: timed data flow (TDF), linear signal flow (LSF) and electrical linear networks (ELN).

A *cluster* is a set of AMS primitive modules connected by channels, which are all associated with the same model of computation.

An *AMS primitive module* is a class derived from the class **sca_core::sca_module** and associated with a model of computation. A primitive module cannot be hierarchically decomposed and contains no child modules or channels. A TDF module is a primitive module derived from class **sca_tdf::sca_module**. An LSF module is a primitive module derived from class **sca_lsf::sca_module**. An ELN module is a primitive module derived from class **sca_eln::sca_module**.

An *AMS port* is a class derived from the class **sca_core::sca_port** and associated with a model of computation. A *primitive port* is a port of a primitive module.

An *AMS terminal* is a class derived from the class **sca_core::sca_port** and associated with the electrical linear networks model of computation.

An *AMS interface* is a class derived from the class **sca_core::sca_interface** and associated with a model of computation.

An *AMS interface proper* is an abstract class derived from the class **sca_core::sca_interface** and associated with a model of computation.

An *AMS channel* is a non-abstract class derived from one or more interfaces and associated with a model of computation.

An *AMS node* is an object of the class **sca_eln::sca_node** or **sca_eln::sca_node_ref** and is an AMS channel associated with the electrical linear networks model of computation.

An *AMS signal* is an object of the class **sca_tdf::sca_signal** or **sca_lsf::sca_signal** and is an AMS channel associated with the timed data flow or linear signal flow model of computation, respectively.

2.2. Syntactical conventions

2.2.1. Implementation-defined

The italicized term *implementation-defined* is used where part of a C++ definition is omitted from this standard. In such cases, an implementation shall provide an appropriate definition that honors the semantics defined in this standard.

2.2.2. Disabled

The italicized term *disabled* is used within a C++ class definition to indicate a group of member functions that shall be disabled by the implementation so that they cannot be called by an application. The disabled member functions are typically the default constructor, the copy constructor, or the assignment operator.

2.2.3. Ellipsis

An ellipsis, which consists of three consecutive dots (…), is used to indicate that irrelevant or repetitive parts of a C++ code listing or example have been omitted for clarity.

2.2.4. Class names

Class names italicized and annotated with a superscript dagger (†) should not be used explicitly within an application. Moreover, an application shall not create an object of such a class. An implementation is strongly recommended to use the given class name. However, an implementation may substitute an alternative class name in place of every occurrence of a particular daggered class name.

Only the class name is being considered here. Whether any part of the definition of the class is implementation-defined is a separate issue.

The class names in question are the following:

sca_core::*sca_assign_from_proxy*†
sca_core::*sca_assign_to_proxy*†
sca_tdf::*sca_ct_proxy*†
sca_tdf::*sca_ct_vector_proxy*†
sca_util::*sca_information_mask*†
sca_util::*sca_traceable_object*†

2.2.5. Prefixes for AMS extensions

The AMS extensions are denoted with the prefix **sca_** for namespaces, classes, functions, global definitions and variables and with the prefix **SCA_** for macros and enumeration values.

An application shall not make use of these prefixes for namespaces, functions, global definitions, variables, macros, and classes derived from the classes as defined in this standard.

2.3. Typographical conventions

The following typographical conventions are used in this standard:

1. The italic font is used for:

- Cross references to terms defined in Subclause 2.1, “Terminology”, Subclause 2.2, “Syntactical conventions”, and Annex B, “*Glossary*”.
- Arguments of member functions in class definitions and in the text that are generally substituted with real values by the implementation or application.

2. The bold font is used for all reserved keywords of SystemC and the AMS extensions as defined in namespaces, macros, constants, enum literals, classes, member functions, data members and types.
3. The constant-width (Courier) font is used:
   - for the SystemC AMS class definition including its member functions, data members and data types.
   - to illustrate SystemC AMS language examples when the exact usage is depicted.
   - for references to the SystemC AMS language syntax and headers.

The conventions listed previously are for ease of reading only. Editorial inconsistencies in the use of typography are unintentional and have no normative meaning in this standard.

2.4. Semantic conventions

2.4.1. Class definitions and the inheritance hierarchy

An implementation may differ from this standard in that an implementation may introduce additional base classes, class members, and friends to the classes defined in this standard. An implementation may modify the inheritance hierarchy by moving class members defined by this standard into base classes not defined by this standard. Such additions and modifications may be made as necessary in order to implement the semantics defined by this standard or in order to introduce additional functionality not defined by this standard.

2.4.2. Function definitions and side-effects

This standard explicitly defines the semantics of the C++ functions for the AMS class library for SystemC. Such functions shall not have any side-effects that would contradict the behavior explicitly mandated by this standard. In general, the reader should assume the common-sense rule that if it is explicitly stated that a function shall perform action A, that function shall not perform any action other than A, either directly or by calling another function defined in this standard. However, a function may, and indeed in certain circumstances shall, perform any tasks necessary for resource management, performance optimization, or to support any ancillary features of an implementation. As an example of resource management, it is assumed that a destructor will perform any tasks necessary to release the resources allocated by the corresponding constructor. As an example of an ancillary feature, an implementation could have the constructor for class **sca_core::sca_module** increment a count of the number of module instances in the module hierarchy.

2.4.3. Functions whose return type is a reference or a pointer

Many functions in this standard return a reference to an object or a pointer to an object, that is, the return type of the function is a reference or a pointer. This subclause gives some general rules defining the lifetime and the validity of such objects.

An object returned from a function by pointer or by reference is said to be valid during any period in which the object is not deleted and the value or behavior of the object remains accessible to the application. If an application refers to the returned object after it ceases to be valid, the behavior of the implementation shall be undefined.

2.4.3.1. Functions that return *this or an actual argument

In certain cases, the object returned is either an object (*this) returned by reference from its own member function (for example, the assignment operators), or is an object that was passed by reference as an actual argument to the function being called (for example, std::ostream& operator<<(std::ostream&, const T&)). In either case, the function call itself places no additional obligations on the implementation concerning the lifetime and validity of the object following return from the function call.

2.4.3.2. Functions that return char*

Certain functions have the return type char*, that is, they return a pointer to a null-terminated character string. Such strings shall remain valid until the end of the program.

2.4.4. Namespaces and internal naming

An implementation shall place every declaration and definition specified by this standard within one of the following namespaces: **sca_core**, **sca_tdf**, **sca_lsf**, **sca_eln**, **sca_ac_analysis** or **sca_util**.

The core language base classes shall be placed in the namespace **sca_core**.

For the predefined models of computation, the following namespaces shall be used:

- The predefined classes for timed data flow shall be placed in the namespace **sca_tdf**.
- The predefined classes for linear signal flow shall be placed in the namespace **sca_lsf**.
- The predefined classes for electrical linear networks shall be placed in the namespace **sca_eln**.

The predefined classes for small-signal frequency-domain analyses shall be placed in the namespace **sca_ac_analysis**. The utilities shall be placed in the namespace **sca_util**.

It is recommended that an implementation uses nested namespaces in order to reduce to a minimum the number of implementation-defined names in these namespaces.

For predefined primitive modules, which use ports to connect to a different model of computation, the namespace associated with the connected model of computation shall be used as nested namespace. The nested namespace **sca_de** shall be used for modules or ports, which are used to connect to SystemC discrete-event channels or ports.

In general, the choice of internal, implementation-specific names within an implementation can cause naming conflicts within an application. It is up to the implementor to choose names that are unlikely to cause naming conflicts within an application.

2.4.5. Non-compliant applications and errors

In the case where an application fails to meet an obligation imposed by this standard, the behavior of the AMS implementation shall be undefined in general. When this results in the violation of a diagnosable rule of the C++ standard, the C++ implementation will issue a diagnostic message in conformance with the C++ standard.

When this standard explicitly states that the failure of an application to meet a specific obligation is an *error* or a *warning*, the AMS implementation shall generate a diagnostic message by calling the function **sc_core::sc_report_handler::report**. In the case of an *error*, the implementation shall call function report with a severity of **sc_core::SC_ERROR**. In the case of a *warning*, the implementation shall call function report with a severity of **sc_core::SC_WARNING**.

An implementation or an application may choose to suppress run-time error checking and diagnostic messages because of considerations of efficiency or practicality. For example, an application may call member function **set_actions** of class **sc_core::sc_report_handler** to take no action for certain categories of reports. An application that fails to meet the obligations imposed by this standard remains in error. There are cases where this standard states explicitly that a certain behavior or result is *undefined*. This standard places no obligations on the implementation in such a circumstance. In particular, such a circumstance may or may not result in an *error* or a *warning*.

2.5. Notes and examples

Notes appear at the end of certain subclauses, designated by the upper-case word “NOTE”. Notes often describe consequences of rules defined elsewhere in this standard. Certain subclauses include examples

consisting of fragments of C++ source code. Such notes and examples are informative to help the reader but are not an official part of this standard.

This page is intentionally left blank.

3. Core language definitions

3.1. Class header files

To use the AMS class library features, an application shall include either of the C++ header files specified in this subclause at appropriate positions in the source code as required by the scope and linkage rules of C++.

3.1.1. #include “systemc-ams”

The header file named **systemc-ams** shall add the names **sca_core**, **sca_tdf**, **sca_lsf**, **sca_eln**, **sca_ac_analysis** and **sca_util**, as well as the names defined in IEEE Std 1666-2005 for the header file named **systemc**, to the declarative region in which it is included. The header file **systemc-ams** shall not introduce into the declarative region in which it is included any other names from this standard or any names from the standard C or C++ libraries.

It is recommended that applications include the header file **systemc-ams** rather than the header file **systemc-ams.h**.

3.1.2. #include “systemc-ams.h”

The header file named **systemc-ams.h** shall add names from the namespace **sca_core, sca_ac_analysis** and **sca_util** as defined in this subclause to the declarative region in which it is included. It is recommended that an implementation keeps the number of additional implementation-specific names introduced by this header file to a minimum.

The header file **systemc-ams.h** shall include at least the following:

```
#include "systemc.h"
#include "systemc-ams"

// Using declarations for the following names in the sca_ac_analysis namespace
using sca_ac_analysis::sca_ac_start;
using sca_ac_analysis::sca_ac_noise_start;
using sca_ac_analysis::sca_ac;
using sca_ac_analysis::sca_ac_is_running;
using sca_ac_analysis::sca_ac_noise;
using sca_ac_analysis::sca_ac_noise_is_running;
using sca_ac_analysis::sca_ac_f;
using sca_ac_analysis::sca_ac_w;
using sca_ac_analysis::sca_ac_s;
using sca_ac_analysis::sca_ac_z;
using sca_ac_analysis::sca_ac_delay;
using sca_ac_analysis::sca_ac_ltf_nd;
using sca_ac_analysis::sca_ac_ltf_zp;
using sca_ac_analysis::sca_ac_ss;
using sca_ac_analysis::SCA_LOG;
using sca_ac_analysis::SCA_LIN;

// Using declarations for the following names in the sca_util namespace
using sca_util::sca_trace_file;
using sca_util::sca_trace;
using sca_util::sca_create_tabular_trace_file;
using sca_util::sca_close_tabular_trace_file;
using sca_util::sca_create_vcd_trace_file;
using sca_util::sca_close_vcd_trace_file;
using sca_util::sca_write_comment;
using sca_util::sca_complex;
using sca_util::sca_matrix;
using sca_util::sca_vector;
using sca_util::sca_create_vector;
using sca_util::sca_information_on;
using sca_util::sca_information_off;
using sca_util::SCA_AC_REAL_IMAG;
using sca_util::SCA_AC_MAG_RAD;
using sca_util::SCA_AC_DB_DEG;
using sca_util::SCA_NOISE_SUM;
using sca_util::SCA_NOISE_ALL;
using sca_util::SCA_INTERPOLATE;
using sca_util::SCA_DONT_INTERPOLATE;
using sca_util::SCA_HOLD_SAMPLE;
using sca_util::sca_ac_format;
```

```
using sca_util::sca_noise_format;
using sca_util::sca_decimation;
using sca_util::sca_sampling;
using sca_util::sca_multirate;
using sca_util::SCA_COMPLEX_J;
using sca_util::SCA_INFINITY;
namespace sca_info = sca_util::sca_info;

// Using declarations for the following names in the sca_core namespace
using sca_core::sca_parameter;
using sca_core::sca_time;
using sca_core::sca_copyright;
using sca_core::sca_version;
using sca_core::sca_release;
```

3.2. Base class definitions

All names used in the base class definitions shall be placed in the namespace **sca_core**.

3.2.1. sca_core::sca_module

3.2.1.1. Description

The class **sca_core::sca_module** shall define the base class to derive primitive modules for the predefined models of computation.

3.2.1.2. Class definition

```
namespace sca_core {

  class sca_module : public sc_core::sc_module
  {
   public:
    virtual const char* kind() const;

    virtual void set_timestep( const sca_core::sca_time& );
    virtual void set_timestep( double, sc_core::sc_time_unit );

    virtual sca_core::sca_time get_timestep() const;

   protected:
    sca_module();
    virtual ~sca_module();
  };

  #define SCA_CTOR(name)       implementation-defined name( sc_core::sc_module_name )

} // namespace sca_core
```

3.2.1.3. Constraints on usage

Any primitive module as defined in Clause 4 shall be publicly derived from class **sca_core::sca_module**.

Objects of class **sca_core::sca_module** can only be constructed during elaboration. It shall be an error to instantiate a primitive module during simulation.

Although class **sca_core::sca_module** is derived from class **sc_core::sc_module**, the use of classes, member functions, and functions, which have direct access to the SystemC kernel shall not be allowed in the context of a module derived from class **sca_core::sca_module**.

The following member functions of class **sc_core::sc_module** shall not be called in the context of a module derived from class **sca_core::sca_module**:

- All forms of member function **sc_core::sc_module::wait**.
- All forms of member function **sc_core::sc_module::next_trigger**.
- All forms of member function **sc_core::sc_module::reset_signal_is**.
- Member function **sc_core::sc_module::dont_initialize**.
- Member function **sc_core::sc_module::set_stack_size**.

The following macros shall not be used in the context of a module derived from class **sca_core::sca_module**:

- Macro **SC_CTOR**.
- Macro **SC_HAS_PROCESS**.
- Macro **SC_METHOD**.
- Macro **SC_THREAD**.
- Macro **SC_CTHREAD**.
- Macro **SC_FORK**.
- Macro **SC_JOIN**.

The following functions shall not be used in the context of a module derived from class **sca_core::sca_module**:

- All forms of function **sc_core::wait**.
- All forms of function **sc_core::next_trigger**.
- Function **sc_core::sc_time_stamp**.
- Function **sc_core::sc_delta_count**.
- Function **sc_core::sc_get_current_process_handle**.
- Function **sc_core::sc_spawn**.

Objects of the following classes shall not be created in the context of modules derived from class **sca_core::sca_module**:

- Objects of class **sc_core::sc_event**.
- Objects of class **sc_core::sc_process_handle**.
- All objects which are derived from class **sc_core::sc_export_base**.
- All objects which are derived from class **sc_core::sc_interface**.
- All objects which are derived from class **sc_core::sc_module**.
- All objects which are derived from class **sc_core::sc_port_base** and not from **sca_core::sca_port**.

An application shall not derive from class **sca_core::sca_module** directly, but shall use the primitive modules defined in Clause 4.

3.2.1.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_core::sca_module**".

3.2.1.5. set_timestep

```
virtual void set_timestep( const sca_core::sca_time& );
virtual void set_timestep( double, sc_core::sc_time_unit );
```

The member function **set_timestep** shall define the timestep of the module according to the execution semantics of the associated predefined model of computation (see 4.1.3, 4.2.3, 4.3.3). If the member function is not called, the current timestep of the module is computed as defined in the execution semantics of the associated predefined model of computation. It shall be an error to call this function after the first **sc_core::sc_module::end_of_elaboration** callback has been executed.

3.2.1.6. get_timestep

```
virtual sca_core::sca_time get_timestep() const;
```

The member function **get_timestep** shall return the current timestep of the module according to the execution semantics of the associated predefined model of computation (see 4.1.3, 4.2.3, 4.3.3). It shall be an error to call this function before the elaboration phase has finished.

3.2.1.7. SCA_CTOR

The macro **SCA_CTOR** is provided for convenience when declaring or defining a constructor for a module derived from class **sca_core::sca_module**. The macro shall only be used at a place where the rules of C++ permit a constructor to be declared and can be used as the declarator of a constructor declaration or a constructor definition. The name of the module class being constructed shall be passed as the argument to the macro.

3.2.2. sca_core::sca_interface

3.2.2.1. Description

The class **sca_core::sca_interface** shall define the base class to derive interfaces for the predefined models of computation.

3.2.2.2. Class definition

```
namespace sca_core {

  class sca_interface : public sc_core::sc_interface
  {
   protected:
    sca_interface();

   private:
    // Disabled
    sca_interface( const sca_core::sca_interface& );
    sca_core::sca_interface& operator= ( const sca_core::sca_interface& );
  };

} // namespace sca_core
```

3.2.2.3. Constraints on usage

An application shall not use class **sca_core::sca_interface** as the direct base class for any class other than an interface proper.

3.2.3. sca_core::sca_prim_channel

3.2.3.1. Description

The class **sca_core::sca_prim_channel** shall be used as base class to derive primitive channels for the predefined models of computation.

3.2.3.2. Class definition

```
namespace sca_core {

  class sca_prim_channel : public sc_core::sc_object,
                           public sca_util::sca_traceable_object†
  {
   public:
    virtual const char* kind() const;

   protected:
    sca_prim_channel();
    explicit sca_prim_channel( const char* );
    virtual ~sca_prim_channel();

   private:
    // Disabled
    sca_prim_channel( const sca_core::sca_prim_channel& );
    sca_core::sca_prim_channel& operator= ( const sca_core::sca_prim_channel& );
  };

} // namespace sca_core
```


3.2.3.3. Constraints on usage

Any primitive channel as defined in Clause 4 shall be publicly derived from class **sca_core::sca_prim_channel**.

Objects of class **sca_core::sca_prim_channel** can only be constructed during elaboration. It shall be an error to instantiate a primitive channel during simulation.

NOTE—Because the constructors are protected, class **sca_core::sca_prim_channel** cannot be instantiated directly. An application shall use only the channels defined in Clause 4 and shall not derive directly any channel from this class.

3.2.3.4. Constructors

```
sca_prim_channel();

explicit sca_prim_channel( const char* );
```

The constructor for class **sca_core::sca_prim_channel** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sc_core::sc_object** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_prim_channel**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sc_core::sc_object**.

3.2.3.5. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string “**sca_core::sca_prim_channel**”.

3.2.4. sca_core::sca_port

3.2.4.1. Description

The class **sca_core::sca_port** shall define the base class to derive ports for the predefined models of computation. The class **sca_core::sca_port** shall implement the interface of class **sca_util**::*sca_traceable_object*[†], in such a way that the channel, to which the port is bound, can be traced.

3.2.4.2. Class definition

```
namespace sca_core {

  template <class IF>
  class sca_port : public sc_core::sc_port<IF, 1, sc_core::SC_ONE_OR_MORE_BOUND >,
                   public sca_util::sca_traceable_object†
  {
   public:
    virtual const char* kind() const;

   protected:
    sca_port();
    explicit sca_port( const char* );
    virtual ~sca_port();
  };

} // namespace sca_core
```

3.2.4.3. Template parameter IF

The argument of template **sca_core::sca_port** shall be the name of an interface proper. The interface shall be derived from class **sc_core::sc_interface**.

3.2.4.4. Constraints on usage

An application shall not use the class **sca_core::sca_port** to instantiate ports, but shall use the ports defined in Clause 4.

3.2.4.5. Constructors

```
sca_port();

explicit sca_port( const char* );
```

The constructor for class **sca_core::sca_port** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sc_core::sc_port** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_port**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sc_core::sc_port**.

3.2.4.6. kind

```
virtual const char* kind() const;
```

The member function kind shall return the string “**sca_core::sca_port**”.

3.2.5. sca_core::sca_time

The class **sca_core::sca_time** shall be used to represent simulation time for the AMS extensions.

```
namespace sca_core { typedef sc_core::sc_time sca_time; }
```

NOTE—The type **sca_core::sca_time** has been introduced to facilitate future extensions to decouple the time resolution used in the AMS extensions from the time resolution as defined in IEEE Std 1666-2005 (SystemC Language Reference Manual).

3.2.6. sca_core::sca_parameter_base

3.2.6.1. Description

The class **sca_core::sca_parameter_base** shall define a type independent base class for module parameters. After construction, parameters shall be unlocked.

NOTE—All instances of class **sca_core::sca_parameter_base** become part of the object hierarchy to facilitate the access to the primitive module parameter values.

3.2.6.2. Class definition

```
namespace sca_core {

  class sca_parameter_base : public sc_core::sc_object
  {
   public:
    virtual const char* kind() const;

    virtual std::string to_string() const = 0 ;
    virtual void print( std::ostream& = std::cout ) const = 0;

    void lock();
    void unlock();
    bool is_locked() const;

   protected:
    sca_parameter_base();
    explicit sca_parameter_base( const char* );
    virtual ~sca_parameter_base();

   private:
    // Disabled
    sca_parameter_base( const sca_core::sca_parameter_base& );
    sca_core::sca_parameter_base& operator= ( const sca_core::sca_parameter_base& );
  };

  std::ostream& operator<< ( std::ostream&, const sca_core::sca_parameter_base& );

} // namespace sca_core
```


3.2.6.3. Constructors

```
sca_parameter_base();

explicit sca_parameter_base( const char* );
```

The constructor for class **sca_core::sca_parameter_base** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sc_core::sc_object** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**("**sca_parameter_base**") to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sc_core::sc_object**.

3.2.6.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_core::sca_parameter_base**".

3.2.6.5. to_string

```
virtual std::string to_string() = 0;
```

The member function **to_string** shall perform the conversion of the parameter value to an object of class std::string.

3.2.6.6. print

```
virtual void print( std::ostream& = std::cout ) = 0;
```

The member function **print** shall print the parameter value to the stream passed as an argument.

3.2.6.7. lock

```
void lock();
```

The member function **lock** shall prevent any further assignment to the parameter. It shall be an error if an assignment is executed on a locked parameter.

3.2.6.8. unlock

```
void unlock();
```

The member function **unlock** shall allow further assignments to the parameter.

3.2.6.9. is_locked

```
bool is_locked() const;
```

The member function **is_locked** shall return true if the parameter is locked, and false otherwise.

3.2.6.10. operator<<

```
std::ostream& operator<< ( std::ostream&, const sca_core::sca_parameter_base& );
```

The **operator**<< shall write the value of the parameter passed as the second argument to the stream passed as the first argument by calling calling the member function **print**(std::ostream).

3.2.7. sca_core::sca_parameter

3.2.7.1. Description

The class **sca_core::sca_parameter** shall assign a parameter to a module.

3.2.7.2. Class definition

```
namespace sca_core {

  template<class T>
  class sca_parameter : public sca_core::sca_parameter_base
  {
   public:
    sca_parameter();
    explicit sca_parameter( const char* name_ );
    sca_parameter( const char* name_, const T& default_value );
    ~sca_parameter();

    virtual const char* kind() const;

    virtual std::string to_string() const;
    virtual void print( std::ostream& = std::cout ) const;

    const T& get() const;
    operator const T&() const;

    void set( const T& );
    sca_core::sca_parameter<T>& operator= ( const T& value );
    sca_core::sca_parameter<T>& operator= ( const sca_core::sca_parameter<T>& value );
  };

} // namespace sca_core
```

3.2.7.3. Template parameter T

The argument passed as template parameter **T** shall be either a C++ type for which the predefined semantics for assignment are adequate (for example, a fundamental type or a pointer), or a type **T** that obeys each of the following rules:

a. The following stream operator shall be defined and should copy the state of the object given as the second argument to the stream given as the first argument. The way in which the state information is formatted is undefined by this standard.

```
std::ostream& operator<< ( std::ostream&, const T& );
```

b. If the default assignment semantics are inadequate (in the sense given in this subclause), the following assignment operator should be defined for the type **T**. In either case (default assignment or explicit operator), the semantics of assignment should be sufficient to assign the state of an object of type **T** such that the value of the left operand is indistinguishable from the value of the right operand.

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

c. If any constructor for type **T** exists, a default constructor for type **T** shall be defined.

3.2.7.4. Constructors

The constructors shall only be called within the context of a **sc_core::sc_module** during module construction.

```
sca_parameter();
```

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_parameter**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sca_core::sca_parameter_base**. The actual parameter value shall be created by the default constructor of the corresponding type.

```
explicit sca_parameter( const char* name_ );
```

The constructor shall pass the character string *name_* through to the constructor belonging to the base class **sca_core::sca_parameter_base** to set the string name of the instance in the module hierarchy. The actual parameter value shall be created by the default constructor of the corresponding type.

```
sca_parameter( const char* name_, const T& default_value );
```

The constructor shall pass the character string *name_* through to the constructor belonging to the base class **sca_core::sca_parameter_base** to set the string name of the instance in the module hierarchy. The

actual parameter value shall be created by the default constructor and initialized with the default value *default_value*.

3.2.7.5. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_core::sca_parameter**".

3.2.7.6. to_string

```
virtual std::string to_string() const;
```

The member function **to_string** shall perform the conversion of the parameter value to an object of class std::string. Conversion shall be done by calling **operator**<< (std::ostream&, const T&). (See 3.2.7.3.)

3.2.7.7. print

```
virtual void print( std::ostream& = std::cout ) const;
```

The member function **print** shall print the current parameter value to the stream passed as an argument by calling **operator**<< (std::ostream&, const T&). (See 3.2.7.3.)

3.2.7.8. get

```
const T& get() const;

operator const T&() const;
```

The member function **get** and **operator const T&** shall return a const reference to the actual parameter value. If the member functions **lock** or **unlock** were not yet executed, the member function **get** and **operator const T&** shall execute the member function **lock** of the base class **sca_core::sca_parameter_base**.

3.2.7.9. set

```
void set( const T& value );

sca_core::sca_parameter<T>& operator= ( const T& value );
```

The member function **set** and **operator**= shall assign the *value* to the parameter.

```
sca_core::sca_parameter<T>& operator= ( const sca_core::sca_parameter<T>& value );
```

The **operator=** shall copy the value of the parameter passed as argument.

3.2.8. sca_core::*sca_assign_from_proxy*†

3.2.8.1. Description

The class **sca_core::***sca_assign_from_proxy*† shall be a helper class to facilitate the implementation of the assignment operator from one class to another.

NOTE—This class is the base class of **sca_tdf::***sca_ct_proxy*† and **sca_tdf::***sca_ct_vector_proxy*†, to implement the assignment of the values returned by these classes to a port or vector, which type is passed to this base class as template parameter of type **T**.

3.2.8.2. Class definition

```
namespace sca_core {

  template<class T>
  class sca_assign_from_proxy†
  {
    implementation-defined
  };

} // namespace sca_core
```


3.2.8.3. Constraint on usage

An application shall not explicitly create an instance of class **sca_core::***sca_assign_from_proxy*†.

3.2.9. sca_core::*sca_assign_to_proxy*†

3.2.9.1. Description

The class **sca_core::***sca_assign_to_proxy*† shall be a helper class to facilitate the implementation of operators for an assignment of values to an object.

NOTE—This class is used to support the implementation of the **operator**[] for the classes **sca_tdf::sca_out**, **sca_tdf::sca_de::sca_out** and **sca_tdf::sca_trace_variable**.

3.2.9.2. Class definition

```
namespace sca_core {

  template<class T, class TV>
  class sca_assign_to_proxy†
  {
    sca_core::sca_assign_to_proxy†<T, TV>& operator= ( const TV& value );
  };

} // namespace sca_core
```

3.2.9.3. operator=

```
sca_assign_to_proxy†<T, TV>& operator= ( const TV& value );
```

The **operator**= performs the assignment of value *value* to an object of class **T**.

3.2.9.4. Constraint on usage

An application shall not explicitly create an instance of class **sca_core::***sca_assign_to_proxy*†.

4. Predefined models of computation

4.1. Timed data flow model of computation

The TDF model of computation shall define the procedural behavior that processes samples, which are tagged in time. A TDF module shall define time domain processing, which is activated when a predefined number of samples is available at its input port(s) and generates a predefined number of output samples at its output port(s). Since the number of read and written samples is known and fixed, the activation schedule of a set of connected TDF modules can be statically determined. For the communication with the SystemC kernel, predefined specialized ports shall be used to maintain synchronization. For synchronization, the tagged time of the samples shall be used. A TDF module is a primitive module that cannot be further hierarchically decomposed.

4.1.1. TDF class definitions

All names used in the TDF class definitions shall be placed in the namespace **sca_tdf**.

4.1.1.1. sca_tdf::sca_module

4.1.1.1.1. Description

The class **sca_tdf::sca_module** shall define the base class for all TDF primitive modules.

4.1.1.1.2. Class definition

```
namespace sca_tdf {

  class sca_module : public sca_core::sca_module
  {
   public:
    virtual const char* kind() const;

   protected:
    typedef void ( sca_tdf::sca_module::*sca_module_method )();

    virtual void set_attributes();
    virtual void initialize();
    virtual void processing();
    virtual void ac_processing();

    void register_processing( sca_tdf::sca_module::sca_module_method );
    void register_ac_processing( sca_tdf::sca_module::sca_module_method );

    sca_core::sca_time get_time() const;

    explicit sca_module( sc_core::sc_module_name );
    sca_module();

    virtual ~sca_module();
  };

#define SCA_TDF_MODULE(name) struct name : sca_tdf::sca_module

} // namespace sca_tdf
```

4.1.1.1.3. Constraints on usage

Modules, channels, signals and ports outside of the namespace **sca_tdf** shall not be instantiated in the context of class **sca_tdf::sca_module**.

Objects of class **sca_tdf::sca_module** can only be constructed during elaboration. It shall be an error to instantiate such a module during simulation. Every class derived (directly or indirectly) from class **sca_tdf::sca_module** shall have at least one constructor. Every such constructor shall have one and only one argument of class **sc_core::sc_module_name**. A string-valued argument shall be passed to the constructor of every module instance.

Inter-module communication for TDF modules shall be accomplished using interface method calls, that is, a module should communicate with its environment through its ports.

4.1.1.1.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_tdf::sca_module**".

4.1.1.1.5. set_attributes

```
virtual void set_attributes();
```

The member function **set_attributes** shall provide a context to set attributes, which are required for TDF MoC elaboration (see 4.1.3). The attributes can be defined using the member function **set_timestep** of a TDF module and member functions **set_timestep**, **set_delay** and **set_rate** for ports of classes **sca_tdf::sca_in** and **sca_tdf::sca_out** and in addition **set_timeoffset** for ports of classes **sca_tdf::sca_de::sca_in** and **sca_tdf::sca_de::sca_out**. The member function **set_attributes** shall be called during the elaboration phase (see 4.1.3.1.1). The application shall not call this member function.

4.1.1.1.6. initialize

```
virtual void initialize();
```

The member function **initialize** shall provide a context to set initial values to member variables and ports. The member function shall be called once-only after the callback to the member function **start_of_simulation** and before the first call to the member function **processing** of a TDF module. The member function **initialize** shall initialize the delay samples of all ports if their delay attribute has been set to a value greater than zero. The application shall not call this member function.

4.1.1.1.7. processing

```
virtual void processing();
```

The member function **processing** shall provide a context to define the time-domain behavior of the TDF module. It may be replaced by a registered application-defined member function (see 4.1.1.1.9). It shall be a warning if a TDF module does not implement a single member function **processing** or a registered application-defined member function when time-domain simulation starts. If no application-defined member function is registered, this member function shall be called during time-domain simulation (see 5.1). The application shall not call this member function.

4.1.1.1.8. ac_processing

```
virtual void ac_processing();
```

The member function **ac_processing** shall provide a context to define the small-signal frequency-domain behavior of the TDF module. It may be replaced by a registered application-defined member function (see 4.1.1.1.10). If no application-defined member function is registered, this function shall be called during small-signal frequency-domain simulation (see 5.2). The application shall not call this member function.

4.1.1.1.9. register_processing

```
void register_processing( sca_tdf::sca_module::sca_module_method );
```

The member function **register_processing** shall register a time-domain processing member function as a replacement to the default time-domain processing member function **processing**. The argument shall be a pointer to a member function of the TDF module. The registered application-defined member function shall behave in the same way as defined in member function **processing** (see 4.1.1.1.7). The member function **register_processing** shall only be called during module construction, otherwise it shall be an error. It shall be an error if more than one member function is registered.

4.1.1.1.10. register_ac_processing

```
void register_processing( sca_tdf::sca_module::sca_module_method );
```

The member function **register_ac_processing** shall register a small-signal frequency-domain processing member function as a replacement to the default small-signal frequency-domain processing member

function **ac_processing**. The argument shall be a pointer to a member function of the TDF module. The registered application-defined member function shall behave in the same way as defined in member function **ac_processing** (see 4.1.1.1.8). The member function **register_ac_processing** shall only be called during module construction, otherwise it shall be an error. It shall be an error if more than one member function is registered.

4.1.1.1.11. get_time

```
sca_core::sca_time get_time() const;
```

The member function **get_time** shall return the current module time of type **sca_core::sca_time**. It represents the time of the first input sample of the current module activation. During elaboration and initialization the member function shall return the value **sc_core::SC_ZERO_TIME**.

NOTE—The function **sc_core::sc_time_stamp** should not be used in a TDF module as there may be time offsets between the current module time of the TDF module and the SystemC kernel time.

4.1.1.1.12. Constructor

```
explicit sca_module( sc_core::sc_module_name );

sca_module();
```

Module names are managed by class **sc_core::sc_module_name**, not by class **sc_core::sc_module**. The string name of the module instance is initialized using the value of the string name passed as an argument to the constructor.

4.1.1.1.13. SCA_TDF_MODULE

The macro **SCA_TDF_MODULE** may be used to prefix the definition of a **sca_tdf::sca_module**, but the use of the macro is not obligatory.

Example:

```
SCA_TDF_MODULE(M1)
{
  // ports, data members, member functions
  ...

  SCA_CTOR(M1);
};

M1::M1( sc_core::sc_module_name )
{
  // constructor body
}
```

4.1.1.2. sca_tdf::sca_signal_if

4.1.1.2.1. Description

The class **sca_tdf::sca_signal_if** shall define an interface proper for a primitive channel of class **sca_tdf::sca_signal**. The interface class member functions are implementation-defined.

4.1.1.2.2. Class definition

```
namespace sca_tdf {

  template<class T>
  class sca_signal_if : public sca_core::sca_interface
  {
   protected:
    sca_signal_if();

   private:
    // Other members
    implementation-defined

    // Disabled
    sca_signal_if( const sca_tdf::sca_signal_if<T>& );
```



```
    sca_tdf::sca_signal_if<T>& operator= ( const sca_tdf::sca_signal_if<T>& );
  };

} // namespace sca_tdf
```

4.1.1.3. sca_tdf::sca_signal

4.1.1.3.1. Description

The class **sca_tdf::sca_signal** shall define a primitive channel for the TDF MoC. It shall be used for connecting modules derived from class **sca_tdf::sca_module** using port classes **sca_tdf::sca_in** and **sca_tdf::sca_out**. An application shall not access the associated interface directly.

4.1.1.3.2. Class definition

```
namespace sca_tdf {

  template<class T>
  class sca_signal : public sca_tdf::sca_signal_if<T>,
                     public sca_core::sca_prim_channel
  {
   public:
    sca_signal();
    explicit sca_signal( const char* name_ );

    virtual const char* kind() const;

   private:
    // Disabled
    sca_signal( const sca_tdf::sca_signal<T>& );
  };

} // namespace sca_tdf
```

4.1.1.3.3. Template parameter T

The argument passed as template parameter **T** shall be either a C++ type for which the predefined semantics for assignment are adequate (for example, a fundamental type or a pointer), or a type **T** that obeys each of the following rules:

a. The following stream operator shall be defined and should copy the state of the object given as the second argument to the stream given as the first argument. The way in which the state information is formatted is undefined by this standard. The implementation shall use this operator for writing trace values in time-domain simulation (see 6.1).

   ```
   std::ostream& operator<< ( std::ostream&, const T& );
   ```

b. If the default assignment semantics are inadequate (in the sense given in this subclause), the following assignment operator should be defined for the type **T**. In either case (default assignment or explicit operator), the semantics of assignment should be sufficient to assign the state of an object of type **T** such that the value of the left operand is indistinguishable from the value of the right operand. The implementation shall use this assignment operator within the implementation for writing to ports.

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

c. If any constructor for type **T** exists, a default constructor for type **T** shall be defined.

4.1.1.3.4. Constructors

```
sca_signal();

explicit sca_signal( const char* name_ );
```

The constructor for class **sca_tdf::sca_signal** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sca_core::sca_prim_channel** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_tdf_signal**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sca_core::sca_prim_channel**.

4.1.1.3.5. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_tdf::sca_signal**".

4.1.1.4. sca_tdf::sca_in

4.1.1.4.1. Description

The class **sca_tdf::sca_in** shall define a port class for the TDF MoC. It provides functions for defining or getting attribute values (e.g. sampling rate or timestep), for initialization and for reading input samples.

4.1.1.4.2. Class definition

```
namespace sca_tdf {

  template<class T>
  class sca_in : public sca_core::sca_port< sca_tdf::sca_signal_if<T> >
  {
   public:
    sca_in();
    explicit sca_in( const char* name_ );

    void set_delay( unsigned long nsamples );
    void set_rate( unsigned long rate );
    void set_timestep( const sca_core::sca_time& tstep );
    void set_timestep( double tstep, sc_core::sc_time_unit unit );

    unsigned long get_delay() const;
    unsigned long get_rate() const;
    sca_core::sca_time get_time( unsigned long sample_id = 0 ) const;
    sca_core::sca_time get_timestep() const;

    virtual const char* kind() const;

    void initialize( const T& value, unsigned long sample_id = 0 );
    const T& read( unsigned long sample_id = 0 ) const;
    operator const T&() const;
    const T& operator[] ( unsigned long sample_id ) const;

   private:
    // Disabled
    sca_in( const sca_tdf::sca_in<T>& );
  };

} // namespace sca_tdf
```

4.1.1.4.3. Template parameter T

The argument passed as template parameter **T** shall be either a C++ type for which the predefined semantics for assignment are adequate (for example, a fundamental type or a pointer), or a type **T** that obeys each of the following rules:

a. The following stream operator shall be defined and should copy the state of the object given as the second argument to the stream given as the first argument. The way in which the state information is formatted is undefined by this standard. The implementation shall use this operator for writing trace values in time-domain simulation (see 6.1).

   ```
   std::ostream& operator<< ( std::ostream&, const T& );
   ```

b. If the default assignment semantics are inadequate (in the sense given in this subclause), the following assignment operator should be defined for the type **T**. In either case (default assignment or explicit operator), the semantics of assignment should be sufficient to assign the state of an object of type **T** such that the value of the left operand is indistinguishable from the value of the right operand. The implementation shall use this assignment operator within the implementation for writing to ports.

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

c. If any constructor for type **T** exists, a default constructor for type **T** shall be defined.

4.1.1.4.4. Constructors

```
sca_in();
```



```
explicit sca_in( const char* name_ );
```

The constructor for class **sca_tdf::sca_in** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sca_core::sca_port** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_tdf_in**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sca_core::sca_port**.

4.1.1.4.5. set_delay

```
void set_delay( unsigned long nsamples );
```

The member function **set_delay** shall define the number of samples to be inserted before the first input sample. If the member function is not called, the port shall have a delay of zero. It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.1.4.6. set_rate

```
void set_rate( unsigned long rate );
```

The member function **set_rate** shall define the number of samples that can be read per execution of the member function **read**. The argument *rate* shall have a positive, nonzero value. If the member function is not called, the port rate shall be equal to 1. It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.1.4.7. set_timestep

```
void set_timestep( const sca_core::sca_time& tstep );
void set_timestep( double tstep, sc_core::sc_time_unit unit );
```

The member function **set_timestep** shall define the timestep between two consecutive samples. If the member function is not called, the current timestep of the port is computed as defined in the execution semantics (see 4.1.3). It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.1.4.8. get_delay

```
unsigned long get_delay() const;
```

The member function **get_delay** shall return the delay set at the port. It shall be an error to call this member function before the elaboration phase has finished.

4.1.1.4.9. get_rate

```
unsigned long get_rate() const;
```

The member function **get_rate** shall return the rate set at the port. It shall be an error to call this member function before the elaboration phase has finished.

4.1.1.4.10. get_time

```
sca_core::sca_time get_time( unsigned long sample_id = 0 ) const;
```

The member function **get_time** shall return the time of the sample of index *sample_id*. It shall be an error to call this member function before the elaboration phase has finished.

The following relation shall hold:

$$P.\mathbf{get\_time}(sample\_id) = M.\mathbf{get\_time}() + \frac{M.\mathbf{get\_timestep}() \cdot sample\_id}{P.\mathbf{get\_rate}()}$$

where *P* is an instance of a port of class **sca_tdf::sca_in** and *M* is the parent module derived from class **sca_tdf::sca_module** (see 4.1.3).

NOTE—The relation is valid within the time resolution bound, which is returned by the function **sc_core::sc_get_time_resolution** (see 4.1.3.1.2).

4.1.1.4.11. get_timestep

```
sca_core::sca_time get_timestep() const;
```

The member function **get_timestep** shall return the current timestep of the port. It shall be an error to call this member function before the elaboration phase has finished.

4.1.1.4.12. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_tdf::sca_in**".

4.1.1.4.13. initialize

```
void initialize( const T& value, unsigned long sample_id = 0 );
```

The member function **initialize** shall initialize one sample at the port. The argument *sample_id* denotes the index of the sample being written. The samples shall be indexed from zero to *P*.**get_delay**()–1, where *P* denotes the port. It shall be an error if *sample_id* is greater than or equal to the port delay. This member function shall only be called in the member function **sca_tdf::sca_module::initialize** of the current module, otherwise it shall be an error. Consecutive initializations with the same *sample_id* shall overwrite the value.

NOTE—The writing of an initial value to a port of class **sca_tdf::sca_in** requires that the port has been assigned a delay using the member function **set_delay**, which shall be called in the member function **set_attributes** of the TDF module.

4.1.1.4.14. read

```
const T& read( unsigned long sample_id = 0 ) const;

operator const T&() const;

const T& operator[] ( unsigned long sample_id ) const;
```

The member function **read**, **operator const T&** and **operator[]** shall return a reference to the value of a particular sample that is available at a port of class **sca_tdf::sca_in**. The argument *sample_id* denotes the index of the sample being read. The samples shall be indexed from zero to *P*.**get_rate**()–1, where *P* denotes the port. A *sample_id* of zero shall refer to the first input sample in time. It shall be an error if *sample_id* is greater than or equal to the port rate.

The member function **read**, **operator const T&** and **operator[]** shall only be called in the time-domain or small-signal frequency-domain processing member function of the current module, otherwise it shall be an error. Consecutive reads with the same *sample_id* during the same module activation shall return the same value.

4.1.1.5. sca_tdf::sca_out

4.1.1.5.1. Description

The class **sca_tdf::sca_out** shall define a port class for the TDF MoC. It provides functions for defining or getting attribute values (e.g. sampling rate or timestep), for initialization and for writing output samples.

4.1.1.5.2. Class definition

```
namespace sca_tdf {

  template<class T>
  class sca_out : public sca_core::sca_port< sca_tdf::sca_signal_if<T> >
  {
   public:
    sca_out();
    explicit sca_out( const char* name_ );

    void set_delay( unsigned long nsamples );
    void set_rate( unsigned long rate );
```

```
    void set_timestep( const sca_core::sca_time& tstep );
    void set_timestep( double tstep, sc_core::sc_time_unit unit );

    unsigned long get_delay() const;
    unsigned long get_rate() const;
    sca_core::sca_time get_time( unsigned long sample_id = 0 ) const;
    sca_core::sca_time get_timestep() const;

    virtual const char* kind() const;

    void initialize( const T& value, unsigned long sample_id = 0 );
    void write( const T& value, unsigned long sample_id = 0 );
    void write( sca_core::sca_assign_from_proxy†<sca_tdf::sca_out<T> >& );
    sca_tdf::sca_out<T>& operator= ( const T& );
    sca_tdf::sca_out<T>& operator= ( const sca_tdf::sca_in<T>& );
    sca_tdf::sca_out<T>& operator= ( const sca_tdf::sca_de::sca_in<T>& );
    sca_tdf::sca_out<T>& operator= ( sca_core::sca_assign_from_proxy†<
                                       sca_tdf::sca_out<T> >& );
    sca_core::sca_assign_to_proxy†<sca_tdf::sca_out<T>,T>& operator[] ( unsigned long sample_id );

  private:
    // Disabled
    sca_out( const sca_tdf::sca_out<T>& );
  };

} // namespace sca_tdf
```

4.1.1.5.3. Template parameter T

The argument passed as template parameter **T** shall be either a C++ type for which the predefined semantics for assignment are adequate (for example, a fundamental type or a pointer), or a type **T** that obeys each of the following rules:

a. The following stream operator shall be defined and should copy the state of the object given as the second argument to the stream given as the first argument. The way in which the state information is formatted is undefined by this standard. The implementation shall use this operator for writing trace values in time-domain simulation (see 6.1).

```
std::ostream& operator<< ( std::ostream&, const T& );
```

b. If the default assignment semantics are inadequate (in the sense given in this subclause), the following assignment operator should be defined for the type **T**. In either case (default assignment or explicit operator), the semantics of assignment should be sufficient to assign the state of an object of type **T** such that the value of the left operand is indistinguishable from the value of the right operand. The implementation shall use this assignment operator within the implementation for writing to ports.

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

c. If any constructor for type **T** exists, a default constructor for type **T** shall be defined.

4.1.1.5.4. Constructors

```
sca_out();

explicit sca_out( const char* name_ );
```

The constructor for class **sca_tdf::sca_out** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sca_core::sca_port** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_tdf_out**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sca_core::sca_port**.

4.1.1.5.5. set_delay

```
void set_delay( unsigned long nsamples );
```

The member function **set_delay** shall define the number of samples to be inserted before the first output sample. If the member function is not called, the port shall have a delay of zero. It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.1.5.6. set_rate

```
void set_rate( unsigned long rate );
```

The member function **set_rate** shall define the number of samples that can be written per execution of the member function **write**. The argument *rate* shall have a positive, nonzero value. If the member function is not called, the port rate shall be equal to 1. It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.1.5.7. set_timestep

```
void set_timestep( const sca_core::sca_time& tstep );
void set_timestep( double tstep, sc_core::sc_time_unit unit );
```

The member function **set_timestep** shall define the timestep between two consecutive samples. If the member function is not called, the current timestep of the port is computed as defined in the execution semantic (see 4.1.3). It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.1.5.8. get_delay

```
unsigned long get_delay() const;
```

The member function **get_delay** shall return the delay set at the port. It shall be an error to call this member function before the elaboration phase has finished.

4.1.1.5.9. get_rate

```
unsigned long get_rate() const;
```

The member function **get_rate** shall return the rate set at the port. It shall be an error to call this member function before the elaboration phase has finished.

4.1.1.5.10. get_time

```
sca_core::sca_time get_time( unsigned long sample_id = 0 ) const;
```

The member function **get_time** shall return the time of the sample of index *sample_id*. It shall be an error to call this member function before the elaboration phase has finished.

The following relation shall hold:

$$P.\mathbf{get\_time}(sample\_id) = M.\mathbf{get\_time}() + \frac{M.\mathbf{get\_timestep}() \cdot sample\_id}{P.\mathbf{get\_rate}()}$$

where *P* is an instance of a port of class **sca_tdf::sca_out** and *M* is the parent module derived from class **sca_tdf::sca_module** (see 4.1.3).

NOTE—The relation is valid within the time resolution bound, which is returned by the function **sc_core::sc_get_time_resolution** (see 4.1.3.1.2).

4.1.1.5.11. get_timestep

```
sca_core::sca_time get_timestep() const;
```

The member function **get_timestep** shall return the current timestep of the port. It shall be an error to call this member function before the elaboration phase has finished.

4.1.1.5.12. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_tdf::sca_out**".

4.1.1.5.13. initialize

```
void initialize( const T& value, unsigned long sample_id = 0 );
```


The member function **initialize** shall initialize one sample at the port. The argument *sample_id* denotes the index of the sample being written. The samples shall be indexed from zero to *P*.**get_delay**()–1, where *P* denotes the port. It shall be an error if *sample_id* is greater than or equal to the port delay.

This member function shall only be called in the member function **sca_tdf::sca_module::initialize** of the current module, otherwise it shall be an error. Consecutive initializations with the same *sample_id* shall overwrite the value.

NOTE—The writing of an initial value to a port of class **sca_tdf::sca_out** requires that the port has been assigned a delay using the member function **set_delay**, which shall be called in the member function **set_attributes** of the TDF module.

4.1.1.5.14. write

```
void write( const T& value, unsigned long sample_id = 0 );

sca_tdf::sca_out<T>& operator= ( const T& );

sca_core::sca_assign_to_proxy†<sca_tdf::sca_out<T>,T>& operator[] ( unsigned long sample_id );
```

The member function **write**, **operator=** and **operator[]** shall write one sample to the port. The argument *sample_id* denotes the index of the sample being written. The samples shall be indexed from zero to *P*.**get_rate**()–1, where *P* denotes the port. It shall be an error if *sample_id* is greater than or equal to the port rate.

```
sca_tdf::sca_out<T>& operator= ( const sca_tdf::sca_in<T>& );

sca_tdf::sca_out<T>& operator= ( const sca_tdf::sca_de::sca_in<T>& );
```

The **operator=** shall copy the values available at the input port of class **sca_tdf::sca_in** or **sca_tdf::sca_de::sca_in** to the output port.

```
void write( sca_core::sca_assign_from_proxy†<sca_tdf::sca_out<T> >& );

sca_tdf::sca_out<T>& operator= ( sca_core::sca_assign_from_proxy†<sca_tdf::sca_out<T> >& );
```

The member function **write** and **operator=** shall copy the values made available through the object of class **sca_core::***sca_assign_from_proxy*† to the output port.

The member function **write**, **operator=** and **operator[]** shall only be called in the time-domain processing member function of the current module, otherwise it shall be an error. Consecutive writes with the same *sample_id* during the same module activation shall overwrite the value.

4.1.1.6. sca_tdf::sca_de::sca_in, sca_tdf::sc_in

4.1.1.6.1. Description

The class **sca_tdf::sca_de::sca_in** shall define a specialized port class for the TDF MoC. It provides functions for defining or getting attribute values (e.g. sampling rate or timestep), for initialization and for reading input values. The port shall perform the synchronization between the TDF MoC and the SystemC kernel (see 4.1.3.2.3). A port of class **sca_tdf::sca_de::sca_in** can be only a member of a module derived from class **sca_tdf::sca_module**, otherwise it shall be an error.

4.1.1.6.2. Class definition

```
namespace sca_tdf {

  namespace sca_de {

    template<class T>
    class sca_in : public sca_core::sca_port< sc_core::sc_signal_in_if<T> >
    {
     public:
      sca_in();
      explicit sca_in( const char* name_ );

      void set_delay( unsigned long nsamples );
      void set_rate( unsigned long rate );
      void set_timestep( const sca_core::sca_time& tstep );
      void set_timestep( double tstep, sc_core::sc_time_unit unit );
      void set_timeoffset( const sca_core::sca_time& toffset );
      void set_timeoffset( double toffset, sc_core::sc_time_unit unit );
```



```
      unsigned long get_delay() const;
      unsigned long get_rate() const;
      sca_core::sca_time get_time( unsigned long sample_id = 0 ) const;
      sca_core::sca_time get_timestep() const;
      sca_core::sca_time get_timeoffset() const;

      virtual const char* kind() const;

      void initialize( const T& value, unsigned long sample_id = 0 );
      const T& read( unsigned long sample_id = 0 );
      operator const T&();
      const T& operator[] ( unsigned long sample_id );

      void bind( sc_core::sc_signal_in_if<T>& );
      void operator()( sc_core::sc_signal_in_if<T>& );

      void bind( sc_core::sc_port<sc_core::sc_signal_in_if<T> >& );
      void operator()( sc_core::sc_port<sc_core::sc_signal_in_if<T> >& );

      void bind( sc_core::sc_port<sc_core::sc_signal_inout_if<T> >& );
      void operator()( sc_core::sc_port<sc_core::sc_signal_inout_if<T> >& );

     private:
      // Disabled
      sca_in( const sca_tdf::sca_de::sca_in<T>& );
    };

  } // namespace sca_de

  template<class T>
  class sc_in: public sca_tdf::sca_de::sca_in<T>
  {
   public:
    sc_in() : sca_tdf::sca_de::sca_in<T>() {}
    explicit sc_in( const char* name_ ) : sca_tdf::sca_de::sca_in<T>( name_ ) {}
  };

} // namespace sca_tdf
```

4.1.1.6.3. Template parameter T

The argument passed as template parameter **T** shall be either a C++ type for which the predefined semantics for assignment are adequate (for example, a fundamental type or a pointer), or a type **T** that obeys each of the following rules:

a. The following stream operator shall be defined and should copy the state of the object given as the second argument to the stream given as the first argument. The way in which the state information is formatted is undefined by this standard. The implementation shall use this operator for writing trace values in time-domain simulation (see 6.1).

```
std::ostream& operator<< ( std::ostream&, const T& );
```

b. If the default assignment semantics are inadequate (in the sense given in this subclause), the following assignment operator should be defined for the type **T**. In either case (default assignment or explicit operator), the semantics of assignment should be sufficient to assign the state of an object of type **T** such that the value of the left operand is indistinguishable from the value of the right operand. The implementation shall use this assignment operator within the implementation for writing to ports.

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

c. If any constructor for type **T** exists, a default constructor for type **T** shall be defined.

4.1.1.6.4. Constructors

```
sca_in();

explicit sca_in( const char* name_ );
```

The constructor for class **sca_tdf::sca_de::sca_in** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sca_core::sca_port** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_tdf_sc_in**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sca_core::sca_port**.

4.1.1.6.5. set_delay

```
void set_delay( unsigned long nsamples );
```

The member function **set_delay** shall define the number of samples to be inserted before the first input sample. If the member function is not called, the port shall have a delay of zero. It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.1.6.6. set_rate

```
void set_rate( unsigned long rate );
```

The member function **set_rate** shall define the number of samples that can be read per execution of the member function **read**. The argument *rate* shall have a positive, nonzero value. If the member function is not called, the port rate shall be equal to 1. It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.1.6.7. set_timestep

```
void set_timestep( const sca_core::sca_time& tstep );
void set_timestep( double tstep, sc_core::sc_time_unit unit );
```

The member function **set_timestep** shall define the timestep between two consecutive samples. If the member function is not called, the current timestep of the port is computed as defined in the execution semantics (see 4.1.3). It shall be an error if the function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.1.6.8. set_timeoffset

```
void set_timeoffset( const sca_core::sca_time& toffset );
void set_timeoffset( double toffset, sc_core::sc_time_unit unit );
```

The member function **set_timeoffset** shall define the absolute time of the first sample. If the member function is not called, the time offset is zero. It shall be an error if the time offset *toffset* is larger than or equal to the current timestep of the port (see 4.1.1.6.12). It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.1.6.9. get_delay

```
unsigned long get_delay() const;
```

The member function **get_delay** shall return the delay set at the port. It shall be an error to call this member function before the elaboration phase has finished.

4.1.1.6.10. get_rate

```
unsigned long get_rate() const;
```

The member function **get_rate** shall return the rate set at the port. It shall be an error to call this member function before the elaboration phase has finished.

4.1.1.6.11. get_time

```
sca_core::sca_time get_time( unsigned long sample_id = 0 ) const;
```

The member function **get_time** shall return the time of the sample of index *sample_id*. It shall be an error to call this function before the elaboration phase has finished.

The following relation shall hold:

$$P.\mathbf{get\_time}(sample\_id) = M.\mathbf{get\_time}() + P.\mathbf{get\_timeoffset}() + \frac{M.\mathbf{get\_timestep}() \cdot sample\_id}{P.\mathbf{get\_rate}()}$$

where *P* is an instance of a port of class **sca_tdf::sca_de::sca_in** and *M* is the parent module derived from class **sca_tdf::sca_module** (see 4.1.3).

NOTE—The relation is valid within the time resolution bound, which is returned by the function **sc_core::sc_get_time_resolution** (see 4.1.3.1.2).

4.1.1.6.12. get_timestep

```
sca_core::sca_time get_timestep() const;
```

The member function **get_timestep** shall return the current timestep of the port. It shall be an error to call this member function before the elaboration phase has finished.

4.1.1.6.13. get_timeoffset

```
sca_core::sca_time get_timeoffset() const;
```

The member function **get_timeoffset** shall return the time offset of the port. It shall be an error to call this member function before the elaboration phase has finished.

4.1.1.6.14. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_tdf::sca_de::sca_in**".

4.1.1.6.15. initialize

```
void initialize( const T& value, unsigned long sample_id = 0 );
```

The member function **initialize** shall initialize one sample at the port. The argument *sample_id* denotes the index of the sample being written. The samples shall be indexed from zero to *P*.**get_delay**()–1, where *P* denotes the port. It shall be an error if *sample_id* is greater than or equal to the port delay.

This member function shall only be called in the member function **sca_tdf::sca_module::initialize** of the current module, otherwise it shall be an error. Consecutive initializations with the same *sample_id* shall overwrite the value.

NOTE—The writing of an initial value to a port of class **sca_tdf::sca_de::sca_in** requires that the port has been assigned a delay using the member function **set_delay**, which shall be called in the member function **set_attributes** of the TDF module.

4.1.1.6.16. read

```
const T& read( unsigned long sample_id = 0 );

operator const T&();

const T& operator[] ( unsigned long sample_id );
```

The member function **read**, **operator const T&** and **operator[]** shall return a reference to the value of a particular sample that is available at a port of class **sca_tdf::sca_de::sca_in**. The argument *sample_id* denotes the index of the sample being read. The samples shall be indexed from zero to *P*.**get_rate**()–1, where *P* denotes the port. A *sample_id* of zero shall refer to the first input sample. It shall be an error if *sample_id* is greater than or equal to the port rate.

The member function **read**, **operator const T&** and **operator[]** shall only be called in the time-domain or small-signal frequency-domain processing member function of the current module, otherwise it shall be an error. Consecutive reads with the same *sample_id* during the same module activation shall return the same value.

The value of a sample shall be read by the member function **read** of the interface proper of class **sc_core::sc_signal_in_if**. The member function **read** of the interface proper of class **sc_core::sc_signal_in_if** shall be called in the evaluation phase at the first delta cycle of the associated time of the sample. (see 4.1.3).

NOTE—The execution of the member function **read** of an interface proper of class **sc_core::sc_signal_in_if** is delayed by *P*.**get_delay**() · *P*.**get_timestep**() + *P*.**get_timeoffset**(), where *P* is an instance of a port of class **sca_tdf::sca_de::sca_in**.

4.1.1.6.17. bind, operator()

```
void bind( sc_core::sc_signal_in_if<T>& );
void operator()( sc_core::sc_signal_in_if<T>& );

void bind( sc_core::sc_port<sc_core::sc_signal_in_if<T> >& );
void operator()( sc_core::sc_port<sc_core::sc_signal_in_if<T> >& );

void bind( sc_core::sc_port<sc_core::sc_signal_inout_if<T> >& );
void operator()( sc_core::sc_port<sc_core::sc_signal_inout_if<T> >& );
```

The member function **bind** and **operator()** shall each call member function **bind** of the base class, passing through their parameters as arguments to the function **bind**, in order to bind the object of class **sca_tdf::sca_de::sca_in** to the channel or port instance passed as an argument.

4.1.1.7. sca_tdf::sca_de::sca_out, sca_tdf::sc_out

4.1.1.7.1. Description

The class **sca_tdf::sca_de::sca_out** shall define a specialized port class for the TDF MoC. It provides functions for defining or getting attribute values (e.g. sampling rate or timestep), for initialization and for writing output values. The port shall perform the synchronization between the TDF MoC and the SystemC kernel (see 4.1.3.2.3). A port of class **sca_tdf::sca_de::sca_out** can be only a member of a module derived from class **sca_tdf::sca_module**, otherwise it shall be an error.

4.1.1.7.2. Class definition

```
namespace sca_tdf {

  namespace sca_de {

    template<class T>
    class sca_out : public sca_core::sca_port< sc_core::sc_signal_inout_if<T> >
    {
     public:
      sca_out();
      explicit sca_out( const char* name_ );

      void set_delay( unsigned long nsamples );
      void set_rate( unsigned long rate );
      void set_timestep( const sca_core::sca_time& tstep );
      void set_timestep( double tstep, sc_core::sc_time_unit unit );
      void set_timeoffset( const sca_core::sca_time& toffset );
      void set_timeoffset( double toffset, sc_core::sc_time_unit unit );

      unsigned long get_delay() const;
      unsigned long get_rate() const;
      sca_core::sca_time get_time( unsigned long sample_id = 0 ) const;
      sca_core::sca_time get_timestep() const;
      sca_core::sca_time get_timeoffset() const;

      virtual const char* kind() const;

      void initialize( const T& value, unsigned long sample_id = 0 );
      void write( const T& value, unsigned long sample_id = 0 );
      void write( sca_core::sca_assign_from_proxy†<sca_tdf::sca_de::sca_out<T> >& );
      sca_tdf::sca_de::sca_out<T>& operator= ( const T& );
      sca_tdf::sca_de::sca_out<T>& operator= ( const sca_tdf::sca_in<T>& );
      sca_tdf::sca_de::sca_out<T>& operator= ( const sca_tdf::sca_de::sca_in<T>& );
      sca_tdf::sca_de::sca_out<T>& operator= ( sca_core::sca_assign_from_proxy†<
                                               sca_tdf::sca_de::sca_out<T> >& );
      sca_core::sca_assign_to_proxy†<sca_tdf::sca_de::sca_out<T>,T>& operator[] (
                                                                  unsigned long sample_id );

    private:
      // Disabled
      sca_out( const sca_tdf::sca_de::sca_out<T>& );
    };

  } // namespace sca_de

  template<class T>
  class sc_out: public sca_tdf::sca_de::sca_out<T>
  {
   public:
    sc_out() : sca_tdf::sca_de::sca_out<T>() {}
    explicit sc_out( const char* name_ ) : sca_tdf::sca_de::sca_out<T>( name_ ) {}
```



```
  };
} // namespace sca_tdf
```

4.1.1.7.3. Template parameter T

The argument passed as template parameter **T** shall be either a C++ type for which the predefined semantics for assignment are adequate (for example, a fundamental type or a pointer), or a type **T** that obeys each of the following rules:

a. The following stream operator shall be defined and should copy the state of the object given as the second argument to the stream given as the first argument. The way in which the state information is formatted is undefined by this standard. The implementation shall use this operator for writing trace values in time-domain simulation (see 6.1).

```
std::ostream& operator<< ( std::ostream&, const T& );
```

b. If the default assignment semantics are inadequate (in the sense given in this subclause), the following assignment operator should be defined for the type **T**. In either case (default assignment or explicit operator), the semantics of assignment should be sufficient to assign the state of an object of type **T** such that the value of the left operand is indistinguishable from the value of the right operand. The implementation shall use this assignment operator within the implementation for writing to ports.

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

c. If any constructor for type **T** exists, a default constructor for type **T** shall be defined.

4.1.1.7.4. Constructors

```
sca_out();

explicit sca_out( const char* name_ );
```

The constructor for class **sca_tdf::sca_de::sca_out** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sca_core::sca_port** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_tdf_sc_out**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sca_core::sca_port**.

4.1.1.7.5. set_delay

```
void set_delay( unsigned long nsamples );
```

The member function **set_delay** shall define the number of samples to be inserted before the first input sample. If the member function is not called, the port shall have a delay of zero. It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.1.7.6. set_rate

```
void set_rate( unsigned long rate );
```

The member function **set_rate** shall define the number of samples that can be written per execution of the member function **write**. The argument *rate* shall have a positive, nonzero value. If the member function is not called, the port rate shall be equal to 1. It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.1.7.7. set_timestep

```
void set_timestep( const sca_core::sca_time& tstep );

void set_timestep( double tstep, sc_core::sc_time_unit unit );
```

The member function **set_timestep** shall define the timestep between two consecutive samples. If the member function is not called, the current timestep of the port is computed as defined in the execution semantic (see 4.1.3). It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.1.7.8. set_timeoffset

```
void set_timeoffset( const sca_core::sca_time& toffset );
void set_timeoffset( double toffset, sc_core::sc_time_unit unit );
```

The member function **set_timeoffset** shall define the absolute time of the first sample. If the member function is not called, the time offset is zero. It shall be an error if the time offset *toffset* is larger than or equal to the current timestep of the port (see 4.1.1.7.12). It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.1.7.9. get_delay

```
unsigned long get_delay() const;
```

The member function **get_delay** shall return the delay set at the port. It shall be an error to call this member function before the elaboration phase has finished.

4.1.1.7.10. get_rate

```
unsigned long get_rate() const;
```

The member function **get_rate** shall return the rate set at the port. It shall be an error to call this member function before the elaboration phase has finished.

4.1.1.7.11. get_time

```
sca_core::sca_time get_time( unsigned long sample_id = 0 ) const;
```

The member function **get_time** shall return the time of the sample of index *sample_id*. It shall be an error to call this function before the elaboration phase has finished.

The following relation shall hold:

$$P.\mathbf{get\_time}(sample\_id) = M.\mathbf{get\_time}() + P.\mathbf{get\_timeoffset}() + \frac{M.\mathbf{get\_timestep}() \cdot sample\_id}{P.\mathbf{get\_rate}()}$$

where *P* is an instance of a port of class **sca_tdf::sca_de::sca_out** and *M* is the parent module derived from class **sca_tdf::sca_module** (see 4.1.3).

NOTE—The relation is valid within the time resolution bound, which is returned by the function **sc_core::sc_get_time_resolution** (see 4.1.3.1.2).

4.1.1.7.12. get_timestep

```
sca_core::sca_time get_timestep() const;
```

The member function **get_timestep** shall return the current timestep of the port. It shall be an error to call this member function before the elaboration phase has finished.

4.1.1.7.13. get_timeoffset

```
sca_core::sca_time get_timeoffset() const;
```

The member function **get_timeoffset** shall return the time offset of the port. It shall be an error to call this member function before the elaboration phase has finished.

4.1.1.7.14. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_tdf::sca_de::sca_out**".

4.1.1.7.15. initialize

```
void initialize( const T& value, unsigned long sample_id = 0 );
```

The member function **initialize** shall initialize one sample at the port. The argument *sample_id* denotes the index of the sample being written. The samples shall be indexed from zero to *P*.**get_delay**()–1, where *P* denotes the port. It shall be an error if *sample_id* is greater than or equal to the port delay.

This member function shall only be called in the member function **sca_tdf::sca_module::initialize** of the current module, otherwise it shall be an error. Consecutive initializations with the same *sample_id* shall overwrite the value.

NOTE—The writing of an initial value to a port of class **sca_tdf::sca_de::sca_out** requires that the port has been assigned a delay using the member function **set_delay**, which shall be called in the member function **set_attributes** of the TDF module.

4.1.1.7.16. write

```
void write( const T& value, unsigned long sample_id = 0 );

sca_tdf::sca_de::sca_out<T>& operator= ( const T& );

sca_core::sca_assign_to_proxy†<sca_tdf::sca_de::sca_out<T>,T>& operator[] (
                                                              unsigned long sample_id );
```

The member function **write**, **operator=** and **operator[]** shall write one sample to the port. The argument *sample_id* denotes the index of the sample being written. The samples shall be indexed from zero to *P*.**get_rate**()–1, where *P* denotes the port. It shall be an error if *sample_id* is greater than or equal to the port rate.

```
sca_tdf::sca_de::sca_out<T>& operator= ( const sca_tdf::sca_in<T>& );

sca_tdf::sca_de::sca_out<T>& operator= ( const sca_tdf::sca_de::sca_in<T>& );
```

The **operator=** shall copy the values available at the input port of class **sca_tdf::sca_in** or **sca_tdf::sca_de::sca_in** to the output port.

```
void write( sca_core::sca_assign_from_proxy†<sca_tdf::sca_de::sca_out<T> >& );

sca_tdf::sca_de::sca_out<T>& operator= ( sca_core::sca_assign_from_proxy†<sca_tdf::sca_de::sca_out<T> >& );
```

The member function **write** and **operator=** shall copy the values made available through the object of class **sca_core::***sca_assign_from_proxy*† to the output port.

The member function **write**, **operator=** and **operator[]** shall only be called in the time-domain processing member function of the current module, otherwise it shall be an error. Consecutive writes with the same *sample_id* during the same module activation shall overwrite the value.

The value of a sample shall be written by the member function **write** of the interface proper of class **sc_core::sc_signal_inout_if**. The member function **write** shall be called in the evaluation phase at the first delta cycle of the associated time of the sample. (see 4.1.3).

NOTE—The execution of the member function **write** of an interface proper of class **sc_core::sc_signal_inout_if** is delayed by *P*.**get_delay**() · *P*.**get_timestep**() + *P*.**get_timeoffset**(), where *P* is an instance of a port of class **sca_tdf::sca_de::sca_out**.

4.1.1.8. sca_tdf::sca_trace_variable

4.1.1.8.1. Description

The class **sca_tdf::sca_trace_variable** shall implement a variable, which can be traced in a trace file of class **sca_util::sca_trace_file**.

4.1.1.8.2. Class definition

```
namespace sca_tdf {

  template<class T>
  class sca_trace_variable : public sc_core::sc_object,
                             public sca_util::sca_traceable_object†
  {
   public:
    sca_trace_variable();
    explicit sca_trace_variable( const char* name_ );

    virtual const char* kind() const;

    void set_rate( unsigned long rate );
```



```
    void set_timeoffset( const sca_core::sca_time& toffset );
    void set_timeoffset( double toffset, sc_core::sc_time_unit unit );

    void write( const T& value, unsigned long sample_id = 0 );
    sca_tdf::sca_trace_variable<T>& operator= ( const T& value );
    sca_tdf::sca_trace_variable<T>& operator= ( const sca_tdf::sca_in<T>& port );
    sca_tdf::sca_trace_variable<T>& operator= ( const sca_tdf::sca_de::sca_in<T>& port );

    sca_core::sca_assign_to_proxy†<sca_tdf::sca_trace_variable<T>,T>& operator[] (
                                                                    unsigned long sample_id );
};

} // namespace sca_tdf
```

4.1.1.8.3. Constraint on usage

An application shall instantiate an object of class **sca_tdf::sca_trace_variable** only in the context of a class derived from **sca_tdf::sca_module**. An application shall write to an object of this class only within the member function **processing** of the parent module derived from class **sca_tdf::sca_module**. An application shall call the member functions **set_rate** and **set_timeoffset** only in the context of the member function **set_attributes** of the parent module derived from class **sca_tdf::sca_module**.

4.1.1.8.4. Constructors

```
sca_trace_variable();

explicit sca_trace_variable( const char* name_ );
```

The constructor for class **sca_tdf::sca_trace_variable** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sc_core::sc_object** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_trace_variable**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sc_core::sc_object**.

4.1.1.8.5. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string “**sca_tdf::sca_trace_variable**”.

4.1.1.8.6. set_rate

```
void set_rate( unsigned long rate );
```

The member function **set_rate** shall set the number of samples which an application shall write to the trace variable while executing time-domain processing. If the member function is not called, the rate of the trace variable shall be equal to 1.

4.1.1.8.7. set_timeoffset

```
void set_timeoffset( const sca_core::sca_time& toffset );

void set_timeoffset( double toffset, sc_core::sc_time_unit unit );
```

The member function **set_timeoffset** shall define the absolute time of the first sample. If the member function is not called, the time offset is zero. It shall be an error if the time offset *toffset* is larger than or equal to the current timestep of the parent module as returned by the member function **sca_tdf::sca_module::get_timestep** divided by the rate set by the member function **set_rate** of the class **sca_tdf::sca_trace_variable**.

4.1.1.8.8. write

```
void write( const T& value, unsigned long sample_id = 0 );

sca_tdf::sca_trace_variable<T>& operator= ( const T& );

sca_tdf::sca_trace_variable<T>& operator= ( const sca_tdf::sca_in<T>& );
```



```
sca_tdf::sca_trace_variable<T>& operator= ( const sca_tdf::sca_de::sca_in<T>& );

sca_core::sca_assign_to_proxy†<sca_tdf::sca_trace_variable<T>,T>& operator[] (
                                                        unsigned long sample_id );
```

The member function **write**, **operator=** and **operator[]** shall write one sample to the trace variable. The argument *sample_id* denotes the index of the sample being written. The samples shall be indexed from zero to *tv*.**get_rate**()–1, where *tv* denotes the trace variable. It shall be an error if *sample_id* is greater than or equal to the trace variable rate.

The member function **write**, **operator=** and **operator[]** shall only be called in the time-domain processing member function of the current module, otherwise it shall be an error. Consecutive writes with the same *sample_id* during the same module activation shall overwrite the value.

4.1.2. Hierarchical TDF composition and port binding

The hierarchical composition of TDF modules shall use modules derived from class **sc_core::sc_module** and the constructor or its equivalent macro definitions. A hierarchical module can include modules and ports of different models of computation. Port binding rules shall follow IEEE Std 1666-2005 as well as the following specific rules as defined in this subclause. Otherwise, it shall be an error.

- A port of class **sca_tdf::sca_in** shall only be bound to a primitive channel of class **sca_tdf::sca_signal** or to a port of class **sca_tdf::sca_in** or **sca_tdf::sca_out** of the parent module.
- A port of class **sca_tdf::sca_out** shall only be bound to a primitive channel of class **sca_tdf::sca_signal** or to a port of class **sca_tdf::sca_out** of the parent module.
- A port of class **sca_tdf::sca_in** or **sca_tdf::sca_out** shall be bound to exactly one primitive channel of class **sca_tdf::sca_signal** throughout the whole hierarchy.
- A primitive channel of class **sca_tdf::sca_signal** shall have exactly one primitive port of class **sca_tdf::sca_out** bound to it and may have one or more primitive ports of class **sca_tdf::sca_in** bound to it throughout the whole hierarchy.
- A port of class **sca_tdf::sca_de::sca_in** shall only be bound to a channel derived from an interface proper of class **sc_core::sc_signal_in_if** or to a port of class **sc_core::sc_in** or **sc_core::sc_out** of the parent module.
- A port of class **sca_tdf::sca_de::sca_out** shall only be bound to a channel derived from an interface proper of class **sc_core::sc_signal_inout_if** or to a port of class **sc_core::sc_out** of the parent module.

4.1.3. TDF MoC elaboration and simulation

An implementation of the TDF MoC in a SystemC AMS class library shall include a public shell consisting of the predefined classes, functions, macros, and so forth that can be used directly by an application. An implementation also includes a TDF solver that implements the functionality of the TDF class library. The underlying semantics of the TDF solver are defined in this subclause.

The execution of a SystemC AMS application that includes TDF modules consists of elaboration followed by simulation. Elaboration results in the consistent composition of the TDF modules through the computation of TDF attributes. Simulation involves the activation of the member functions **initialize** and **processing** of the TDF modules. In addition to providing support for elaboration and simulation, the TDF solver may also provide implementation-specific functionality beyond the scope of this standard. As an example of such functionality, the TDF solver may compute a static schedule for time-domain processing and may report information about the TDF module composition.

4.1.3.1. TDF elaboration

The primary purpose of TDF elaboration is to create internal data structures for the TDF solver to support the semantics of TDF simulation. The TDF elaboration as described in this clause and in the following subclauses shall execute in one **sc_core::sc_module::end_of_elaboration** callback. The actions stated in the following subclauses shall occur, in the given order, during TDF elaboration and only during TDF elaboration. The description of such actions uses the concept of a TDF cluster, which is a set of TDF modules connected by channels of class **sca_tdf::sca_signal**.

NOTE—It is not defined in which order the TDF elaboration and an application-defined **sca_tdf::sca_module::end_of_elaboration** callback are executed.

4.1.3.1.1. TDF attribute setting

The TDF elaboration phase shall execute, in no particular order, all the member functions **set_attributes** of the modules derived from class **sca_tdf::sca_module**.

4.1.3.1.2. TDF timestep calculation and propagation

The composition of TDF modules involves the computation and the propagation of consistent values for the timesteps at each port of classes **sca_tdf::sca_in**, **sca_tdf::sca_out**, **sca_tdf::sca_de::sca_in** and **sca_tdf::sca_de::sca_out**, and for each TDF module processing function. The port and module timesteps are said to be consistent if they differ by less than the time resolution as returned by the function **sc_core::sc_get_time_resolution**. It shall be an error if consistency is not met.

The timestep of a module derived from class **sca_tdf::sca_module** shall be consistent with the rate and timestep attribute of any port within that module, according to the following relation:

$$M.\textbf{get\_timestep}() = P.\textbf{get\_timestep}() \cdot P.\textbf{get\_rate}()$$

where *M* is an instance of a module derived from class **sca_tdf::sca_module** and *P* is an instance of a port of class **sca_tdf::sca_in**, **sca_tdf::sca_out**, **sca_tdf::sca_de::sca_in** or **sca_tdf::sca_de::sca_out**.

The timestep values for ports bound to the same channel of class **sca_tdf::sca_signal** shall be consistent. The assigned and propagated timestep values shall be consistent throughout the TDF cluster. It shall be an error if a timestep value is not assigned to at least one TDF module or one port of class **sca_tdf::sca_in**, **sca_tdf::sca_out**, **sca_tdf::sca_de::sca_in** or **sca_tdf::sca_de::sca_out** in a TDF cluster. After successful TDF elaboration, all assigned timestep values shall be overridden by the propagated timestep values, rounded to the next smallest multiple of the time resolution, as returned by the function **sc_core::sc_get_time_resolution**.

Each sample read from or written to a port of class **sca_tdf::sca_in**, **sca_tdf::sca_out**, **sca_tdf::sca_de::sca_in** or **sca_tdf::sca_de::sca_out** shall be associated with an absolute time of type **sca_core::sca_time**. The first sample shall be associated with a time set to **sc_core::SC_ZERO_TIME** and the following samples to multiples of the propagated timestep attribute of the port.

4.1.3.1.3. TDF computability check

It shall be an error if TDF clusters are not computable. For each TDF cluster, let *R* be a vector of positive integer values $r_{M_1}, r_{M_2}, \ldots, r_{M_N}$, whose size *N* is the number of modules in the cluster. A TDF cluster is said to be computable if all three following conditions are met:

1. For every pair of ports P_i and P_j belonging, respectively, to modules M_i and M_j of the same cluster and which are bound to the same channel of class **sca_tdf::sca_signal**, the following equation shall hold:

   $$r_{M_i} \cdot P_i.\textbf{get\_rate}() = r_{M_j} \cdot P_j.\textbf{get\_rate}()$$

   It should be noted that modules M_i and M_j may denote the same module.
2. For each cluster, there exists an order of activation of the TDF modules that fulfills the activation conditions as defined in Subclause 4.1.3.2.2, such that each TDF module M_i shall be activated exactly r_{M_i} times.
3. For each cluster, there exists an activation order of modules that guarantees that the time stamps of samples read from ports of class **sca_tdf::sca_de::sca_in** at a particular module activation are always smaller or equal to the time stamps of samples written to ports of class **sca_tdf::sca_de::sca_out** at later scheduled module activations.

4.1.3.2. TDF simulation

This subclause defines the process of time-domain simulation of a TDF cluster. The simulation of TDF modules involves the execution of a TDF initialization phase followed by activations of the time-domain processing member functions.

4.1.3.2.1. TDF initialization

The TDF initialization phase shall include the execution, in no particular order, of all the member functions **initialize** of the TDF modules. The TDF initialization phase shall start after the callbacks to the member functions **start_of_simulation** and before the first call to the first scheduled member function **processing**.

The initial sample values at ports with an associated delay greater than zero are defined by the execution of the port member function **initialize**, if this member function is called in the module member function **initialize**. Otherwise, the initial sample values are defined by the default constructor of the corresponding data type.

Samples written by the member function **initialize** of a port of class **sca_tdf::sca_out** shall be available to all connected ports of class **sca_tdf::sca_in** before the first sample is written by executing the member function **processing** (see 4.1.1.5.13).

4.1.3.2.2. TDF processing

The member function **processing** of class **sca_tdf::sca_module** shall be called, if the required number of samples is available at all the module input ports. The number of required samples is defined by the rates of the ports of class **sca_tdf::sca_in**. After execution of the member function **processing**, the required samples shall be considered as consumed and thus not available anymore.

The number of produced samples is defined by the rates of the ports of class **sca_tdf::sca_out**. After execution of the member function **processing**, the produced samples shall be available to all connected ports of class **sca_tdf::sca_in**.

The samples written by the member function **initialize** of a port of class **sca_tdf::sca_in** or class **sca_tdf::sca_de::sca_in** shall be available first at this port in the order of their sample indexes. (see 4.1.1.4.13 and 4.1.1.6.15).

NOTE 1—Samples which are not written remain undefined.

NOTE 2—Samples available at a port of class **sca_tdf::sca_in** become ordered as follows: 1. samples as defined by the port delay, 2. samples as defined by the port delay of the connected port of class **sca_tdf::sca_out**, 3. samples as written by the member function **processing** of the module that instantiates the connected port of class **sca_tdf::sca_out**.

NOTE 3—The member function **sca_tdf::sca_module::end_of_simulation** may be used to perform post processing actions.

4.1.3.2.3. Synchronization with SystemC kernel

Synchronization with the SystemC kernel shall be done exclusively by using ports of class **sca_tdf::sca_de::sca_in** and class **sca_tdf::sca_de::sca_out**.

It shall be ensured, that while executing the member function **processing** of a module and reading from a port of class **sca_tdf::sca_de::sca_in**, the requested samples are available (see 4.1.1.6.16).

It shall be ensured, that while executing the member function **processing** of a module and writing to a port of class **sca_tdf::sca_de::sca_out**, the sample can be written at the corresponding time (see 4.1.1.7.16).

4.1.3.3. Running elaboration and simulation

The implementation shall use the same elaboration and simulation semantics as defined in IEEE Std 1666-2005 (SystemC Language Reference Manual).

NOTE—TDF modules can be instantiated in the **sc_main** context and the elaboration and simulation can be controlled by the function **sc_core::sc_start**.

4.1.4. Embedded linear dynamic equations

A module derived from class **sca_tdf::sca_module** can embed linear dynamic equations in its member function **processing** given in the form of linear transfer functions in the Laplace domain or state-space equations. The equations shall be solved by considering samples as continuous-time signals. The solution

shall be a continuous-time signal represented by a reference to an object of class **sca_tdf::***sca_ct_proxy*† or **sca_tdf::***sca_ct_vector_proxy*†. Only solutions at discrete time points shall be made available to the TDF context. The required sampling shall be realized by the objects of class **sca_tdf::***sca_ct_proxy*† or **sca_tdf::***sca_ct_vector_proxy*† depending on the output argument.

The discrete time points at which the input values are sampled shall be derived from the timestep value of the module derived from class **sca_tdf::sca_module** (see 4.1.3.2.1), defined from a specific timestep value associated with the equations or derived from ports of class **sca_tdf::sca_in** or **sca_tdf::sca_de::sca_in**. If a timestep value is defined for the equations, the timestep value shall be smaller or equal to the time distance between the last computed solution of the equations and the time of the current activation of the module derived from class **sca_tdf::sca_module**, in which the equations are embedded.

The coefficients of the equation system to be solved can be changed between computations of solutions. The computation of a solution shall be executed at least once in the member function **processing** of the module derived from class **sca_tdf::sca_module**. The time of the last solution shall not be greater than the time of the current activation of the member function **processing** of the module derived from class **sca_tdf::sca_module**, in which the equations are embedded. When the time of the last computed solution is smaller than the current module time, the time step shall be extended by the difference between these two times.

The embedded linear dynamic equation classes shall be instantiated as a member of a module derived from class **sca_tdf::sca_module**. The classes shall be instantiated before the callback **before_end_of_elaboration**. After the computation of the first solution of the equations, the sizes of the coefficient vectors or matrices, representing the number of equations, shall not be changed.

4.1.4.1. sca_tdf::*sca_ct_proxy*†

4.1.4.1.1. Description

The class **sca_tdf::***sca_ct_proxy*† shall be a helper class, which shall map the computed continuous-time solution to sampled output values. An instance of this class shall exist only as reference returned by the member functions **calculate** or **operator()** of class **sca_tdf::sca_ltf_nd** and **sca_tdf::sca_ltf_zp** (see 4.1.4.3.7, 4.1.4.4.7).

4.1.4.1.2. Class definition

```
namespace sca_tdf {

  class sca_ct_proxy† : public sca_core::sca_assign_from_proxy†<sca_util::sca_vector<double> >,
                        public sca_core::sca_assign_from_proxy†<sca_tdf::sca_out<double> >,
                        public sca_core::sca_assign_from_proxy†<sca_tdf::sca_de::sca_out<double> >
  {
   public:
    double to_double() const;
    void   to_vector( sca_util::sca_vector<double>&, unsigned long nsamples = 0 ) const;
    const  sca_util::sca_vector<double>& to_vector( unsigned long nsamples = 0 ) const;
    void   to_port( sca_tdf::sca_out<double>& ) const;
    void   to_port( sca_tdf::sca_de::sca_out<double>& ) const;

    operator double() const;

   private:
    // Disabled
    sca_ct_proxy†();

    void assign_to( sca_util::sca_vector<double>& );
    void assign_to( sca_tdf::sca_out<double>& );
    void assign_to( sca_tdf::sca_de::sca_out<double>& );
  };

} // namespace sca_tdf
```

4.1.4.1.3. Constraint on usage

An application shall not explicitly create an instance of class **sca_tdf::***sca_ct_proxy*†.

4.1.4.1.4. to_double

```
double to_double() const;
```



```
operator double() const;
```

The member function **to_double** and **operator double()** shall sample the continuous-time solution at the end of the current time interval and shall return the value.

4.1.4.1.5. to_vector

```
void to_vector( sca_util::sca_vector<double>&, unsigned long nsamples = 0 ) const;

const sca_util::sca_vector<double>& to_vector( unsigned long nsamples = 0 ) const;
```

The member function **to_vector** shall sample the continuous-time solution with constant timesteps, starting at the beginning of the current time interval plus the time interval divided by the number of samples *nsamples* and finishing with the end time of the current calculation with *nsamples* samples. If *nsamples* is zero, the number of input samples of the current calculation shall be used. The member function shall resize the vector and copy the result or return a reference to a vector of the appropriate size.

4.1.4.1.6. to_port

```
void to_port( sca_tdf::sca_out<double>& port ) const;

void to_port( sca_tdf::sca_de::sca_out<double>& port ) const;
```

The member function **to_port** shall sample the continuous-time solution with the constant timestep associated with the port *port* using the member function *port*.**get_timestep**(), starting at the absolute time associated with the first sample of the port *port* using the member function *port*.**get_time**(0) and finishing at the end of the current time interval as returned by the member function *port*.**get_time**(*port*.**get_rate**()–1) or as provided by the defined timestep. The result will be written to the corresponding samples of the port.

4.1.4.1.7. assign_to

```
void assign_to( sca_util::sca_vector<double>& );

void assign_to( sca_tdf::sca_out<double>& );

void assign_to( sca_tdf::sca_de::sca_out<double>& );
```

The member function **assign_to** shall use the class **sca_core::***sca_assign_from_proxy*†, to map **operator=** of class **sca_util::sca_vector** to the member function **to_vector**. Equally, the **operator=** of a port of class **sca_tdf::sca_out** or **sca_tdf::sca_de::sca_out** shall be mapped to the member function **to_port**.

4.1.4.2. sca_tdf::*sca_ct_vector_proxy*†

4.1.4.2.1. Description

The class **sca_tdf::***sca_ct_vector_proxy*† shall be a helper class, which shall map the computed continuous-time solution to sampled output values. An instance of this class shall exist only as reference returned by the member functions **calculate** or **operator()** of class **sca_tdf::sca_ss** (see 4.1.4.5.6).

4.1.4.2.2. Class definition

```
namespace sca_tdf {

  class sca_ct_vector_proxy† :
    public sca_core::sca_assign_from_proxy†<sca_util::sca_matrix<double> >,
    public sca_core::sca_assign_from_proxy†<sca_tdf::sca_out<sca_util::sca_vector<double> > >,
    public sca_core::sca_assign_from_proxy†<sca_tdf::sca_de::sca_out<sca_util::sca_vector<double> > >
  {
   public:
    const sca_util::sca_vector<double>& to_vector() const;
    void  to_matrix( sca_util::sca_matrix<double>&, unsigned long nsamples = 0 ) const;
    const sca_util::sca_matrix<double>& to_matrix( unsigned long nsamples = 0 ) const;
    void  to_port( sca_tdf::sca_out< sca_util::sca_vector<double> >& ) const;
    void  to_port( sca_tdf::sca_de::sca_out< sca_util::sca_vector<double> >& ) const;

    operator const sca_util::sca_vector<double>&() const;

   private:
    // Disabled
    sca_ct_vector_proxy†();
```

```
    void assign_to( sca_util::sca_matrix<double>& );
    void assign_to( sca_tdf::sca_out<sca_util::sca_vector<double> >& );
    void assign_to( sca_tdf::sca_de::sca_out<sca_util::sca_vector<double> >& );
  };

} // namespace sca_tdf
```

4.1.4.2.3. Constraint on usage

An application shall not explicitly create an instance of class **sca_tdf::***sca_ct_vector_proxy*†.

4.1.4.2.4. to_vector

```
const sca_util::sca_vector<double>& to_vector() const;

operator const sca_util::sca_vector<double>&() const;
```

The member function **to_vector** shall sample the continuous-time solution at the end of the current time interval and shall return the values.

4.1.4.2.5. to_matrix

```
void to_matrix( sca_util::sca_matrix<double>&, unsigned long nsamples = 0 ) const;

const sca_util::sca_matrix<double>& to_matrix( unsigned long nsamples = 0 ) const;
```

The member function **to_matrix** shall sample the continuous-time solution with constant timesteps, starting at the beginning of the current time interval plus the time interval divided by the number of samples *nsamples* and finishing with the end time of the current calculation with *nsamples* samples. If *nsamples* is zero, the number of input samples of the current calculation shall be used. The member function shall resize the matrix and copy the result or return a reference to a matrix of the appropriate size. The column size of the matrix shall be equal to the number of samples.

4.1.4.2.6. to_port

```
void to_port( sca_tdf::sca_out< sca_util::sca_vector<double> >& port ) const;

void to_port( sca_tdf::sca_de::sca_out< sca_util::sca_vector<double> >& port ) const;
```

The member function **to_port** shall sample the continuous-time solution with the constant timestep associated with the port *port* using the member function *port*.**get_timestep**(), starting at the absolute time associated with the first sample of the port *port* using the member function *port*.**get_time**(0) and finishing at the end of the current time interval as returned by the member function *port*.**get_time**(*port*.**get_rate**()–1) or as provided by the defined timestep. The result will be written to the corresponding samples of the port.

4.1.4.2.7. assign_to

```
void assign_to( sca_util::sca_matrix<double>& );

void assign_to( sca_tdf::sca_out<sca_util::sca_vector<double> >& );

void assign_to( sca_tdf::sca_de::sca_out<sca_util::sca_vector<double> >& );
```

The member function **assign_to** shall use the class **sca_core::***sca_assign_from_proxy*†, to map the **operator=** of class **sca_util::sca_matrix** to the member function **to_matrix**. Equally, the **operator=** of a port of class **sca_tdf::sca_out** or **sca_tdf::sca_de::sca_out** shall be mapped to the member function **to_port**.

4.1.4.3. sca_tdf::sca_ltf_nd

4.1.4.3.1. Description

The class **sca_tdf::sca_ltf_nd** shall implement 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 argument *delay* is the time continuous delay applied to the values available at the input.

4.1.4.3.2. Class definition

```
namespace sca_tdf {

  class sca_ltf_nd : public sc_core::sc_object
  {
   public:
    sca_ltf_nd();
    explicit sca_ltf_nd( const char* );

    virtual const char* kind() const;

    void set_max_delay( const sca_core::sca_time& );
    void set_max_delay( double tstep, sc_core::sc_time_unit unit );

    sca_tdf::sca_ct_proxy†& calculate( const sca_util::sca_vector<double>& num,
                                       const sca_util::sca_vector<double>& den,
                                             sca_util::sca_vector<double>& state,
                                             double                        input,
                                             double                        k = 1.0,
                                             sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_proxy†& calculate( const sca_util::sca_vector<double>& num,
                                       const sca_util::sca_vector<double>& den,
                                             sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                             sca_util::sca_vector<double>& state,
                                             double                        input,
                                             double                        k = 1.0,
                                             sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_proxy†& calculate( const sca_util::sca_vector<double>& num,
                                       const sca_util::sca_vector<double>& den,
                                             sca_util::sca_vector<double>& state,
                                       const sca_util::sca_vector<double>& input,
                                             double                        k = 1.0,
                                             sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_proxy†& calculate( const sca_util::sca_vector<double>& num,
                                       const sca_util::sca_vector<double>& den,
                                             sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                             sca_util::sca_vector<double>& state,
                                       const sca_util::sca_vector<double>& input,
                                             double                        k = 1.0,
                                             sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_proxy†& calculate( const sca_util::sca_vector<double>& num,
                                       const sca_util::sca_vector<double>& den,
                                             sca_util::sca_vector<double>& state,
                                       const sca_tdf::sca_in<double>&      input,
                                             double                        k = 1.0 );

    sca_tdf::sca_ct_proxy†& calculate( const sca_util::sca_vector<double>& num,
                                       const sca_util::sca_vector<double>& den,
                                             sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                             sca_util::sca_vector<double>& state,
                                       const sca_tdf::sca_in<double>&      input,
                                             double                        k = 1.0 );

    sca_tdf::sca_ct_proxy†& calculate( const sca_util::sca_vector<double>& num,
                                       const sca_util::sca_vector<double>& den,
                                             sca_util::sca_vector<double>& state,
                                       const sca_tdf::sca_de::sca_in<double>& input,
                                             double                           k = 1.0 );

    sca_tdf::sca_ct_proxy†& calculate( const sca_util::sca_vector<double>& num,
                                       const sca_util::sca_vector<double>& den,
                                             sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                             sca_util::sca_vector<double>& state,
                                       const sca_tdf::sca_de::sca_in<double>& input,
                                             double                           k = 1.0 );

    sca_tdf::sca_ct_proxy†& calculate( const sca_util::sca_vector<double>& num,
                                       const sca_util::sca_vector<double>& den,
                                             double                        input,
                                             double                        k = 1.0,
                                             sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );
```

```
sca_tdf::sca_ct_proxy† & calculate( const sca_util::sca_vector<double>& num,
                                    const sca_util::sca_vector<double>& den,
                                          sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                          double                       input,
                                          double                       k = 1.0,
                                          sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy† & calculate( const sca_util::sca_vector<double>& num,
                                    const sca_util::sca_vector<double>& den,
                                    const sca_util::sca_vector<double>& input,
                                          double                       k = 1.0,
                                          sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy† & calculate( const sca_util::sca_vector<double>& num,
                                    const sca_util::sca_vector<double>& den,
                                          sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                    const sca_util::sca_vector<double>& input,
                                          double                       k = 1.0,
                                          sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy† & calculate( const sca_util::sca_vector<double>& num,
                                    const sca_util::sca_vector<double>& den,
                                    const sca_tdf::sca_in<double>&      input,
                                          double                       k = 1.0 );

sca_tdf::sca_ct_proxy† & calculate( const sca_util::sca_vector<double>& num,
                                    const sca_util::sca_vector<double>& den,
                                          sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                    const sca_tdf::sca_in<double>&      input,
                                          double                       k = 1.0 );

sca_tdf::sca_ct_proxy† & calculate( const sca_util::sca_vector<double>& num,
                                    const sca_util::sca_vector<double>& den,
                                    const sca_tdf::sca_de::sca_in<double>& input,
                                          double                         k = 1.0 );

sca_tdf::sca_ct_proxy† & calculate( const sca_util::sca_vector<double>& num,
                                    const sca_util::sca_vector<double>& den,
                                          sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                    const sca_tdf::sca_de::sca_in<double>& input,
                                          double                         k = 1.0 );

sca_tdf::sca_ct_proxy† & operator() ( const sca_util::sca_vector<double>& num,
                                      const sca_util::sca_vector<double>& den,
                                            sca_util::sca_vector<double>& state,
                                            double                       input,
                                            double                       k = 1.0,
                                            sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy† & operator() ( const sca_util::sca_vector<double>& num,
                                      const sca_util::sca_vector<double>& den,
                                            sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                            sca_util::sca_vector<double>& state,
                                            double                       input,
                                            double                       k = 1.0,
                                            sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy† & operator() ( const sca_util::sca_vector<double>& num,
                                      const sca_util::sca_vector<double>& den,
                                            sca_util::sca_vector<double>& state,
                                      const sca_util::sca_vector<double>& input,
                                            double                       k = 1.0,
                                            sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy† & operator() ( const sca_util::sca_vector<double>& num,
                                      const sca_util::sca_vector<double>& den,
                                            sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                            sca_util::sca_vector<double>& state,
                                      const sca_util::sca_vector<double>& input,
                                            double                       k = 1.0,
                                            sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy† & operator() ( const sca_util::sca_vector<double>& num,
                                      const sca_util::sca_vector<double>& den,
                                            sca_util::sca_vector<double>& state,
                                      const sca_tdf::sca_in<double>&      input,
                                            double                       k = 1.0 );

sca_tdf::sca_ct_proxy† & operator() ( const sca_util::sca_vector<double>& num,
                                      const sca_util::sca_vector<double>& den,
                                            sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                            sca_util::sca_vector<double>& state,
                                      const sca_tdf::sca_in<double>&      input,
```



```
                                        double                      k = 1.0 );

    sca_tdf::sca_ct_proxy†& operator() ( const sca_util::sca_vector<double>& num,
                                         const sca_util::sca_vector<double>& den,
                                               sca_util::sca_vector<double>& state,
                                         const sca_tdf::sca_de::sca_in<double>& input,
                                               double                         k = 1.0 );

    sca_tdf::sca_ct_proxy†& operator() ( const sca_util::sca_vector<double>& num,
                                         const sca_util::sca_vector<double>& den,
                                               sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                               sca_util::sca_vector<double>& state,
                                         const sca_tdf::sca_de::sca_in<double>& input,
                                               double                         k = 1.0 );

    sca_tdf::sca_ct_proxy†& operator() ( const sca_util::sca_vector<double>& num,
                                         const sca_util::sca_vector<double>& den,
                                               double                      input,
                                               double                      k = 1.0,
                                               sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_proxy†& operator() ( const sca_util::sca_vector<double>& num,
                                         const sca_util::sca_vector<double>& den,
                                               double                      input,
                                               sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                               double                      k = 1.0,
                                               sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_proxy†& operator() ( const sca_util::sca_vector<double>& num,
                                         const sca_util::sca_vector<double>& den,
                                         const sca_util::sca_vector<double>& input,
                                               double                      k = 1.0,
                                               sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_proxy†& operator() ( const sca_util::sca_vector<double>& num,
                                         const sca_util::sca_vector<double>& den,
                                               sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                         const sca_util::sca_vector<double>& input,
                                               double                      k = 1.0,
                                               sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_proxy†& operator() ( const sca_util::sca_vector<double>& num,
                                         const sca_util::sca_vector<double>& den,
                                         const sca_tdf::sca_in<double>&      input,
                                               double                      k = 1.0 );

    sca_tdf::sca_ct_proxy†& operator() ( const sca_util::sca_vector<double>& num,
                                         const sca_util::sca_vector<double>& den,
                                               sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                         const sca_tdf::sca_in<double>&      input,
                                               double                      k = 1.0 );

    sca_tdf::sca_ct_proxy†& operator() ( const sca_util::sca_vector<double>& num,
                                         const sca_util::sca_vector<double>& den,
                                         const sca_tdf::sca_de::sca_in<double>& input,
                                               double                      k = 1.0 );

    sca_tdf::sca_ct_proxy†& operator() ( const sca_util::sca_vector<double>& num,
                                         const sca_util::sca_vector<double>& den,
                                               sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                         const sca_tdf::sca_de::sca_in<double>& input,
                                               double                      k = 1.0 );
  };

} // namespace sca_tdf
```

4.1.4.3.3. Constructors

```
sca_ltf_nd();

explicit sca_ltf_nd( const char* );
```

The constructor for class **sca_tdf::sca_ltf_nd** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sc_core::sc_object** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_ltf_nd**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sc_core::sc_object**.

4.1.4.3.4. Constraint on usage

The vectors *num* and *den* shall have at least one element, respectively.

4.1.4.3.5. kind

```
virtual const char* kind() const;
```

The member function kind shall return the string "**sca_tdf::sca_ltf_nd**".

4.1.4.3.6. set_max_delay

```
void set_max_delay( const sca_core::sca_time& );

void set_max_delay( double tstep, sc_core::sc_time_unit unit );
```

The member function **set_max_delay** shall define the maximum allowable time continuous delay of the input values. If the member function is not called, the maximum allowable delay shall be set to the current timestep used, at which the input values are available. It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.4.3.7. calculate, operator()

```
sca_tdf::sca_ct_proxy†& calculate(...);

sca_tdf::sca_ct_proxy†& operator() (...);
```

The member function **calculate** and **operator()** shall return the continuous-time signal of the Laplace-domain variable *s* in the numerator-denominator form, using a reference to the class **sca_tdf::***sca_ct_proxy*†.

The arguments include the gain of the transfer function *k*, the vectors of the numerator coefficients *num* and denominator coefficients *den*, the time continuous delay *delay*, the state vector *state*, the value of the input at the current time *input*, and the timestep *tstep*.

The first element of the vectors *num* and *den* shall be the coefficient of order zero of the respective polynomial.

The argument *delay* specifies the time continuous delay which shall be applied to the input values before calculating the linear transfer function. The delay shall be smaller than or equal to the current timestep used, at which the input values are available, or if the member function **set_max_delay** has been called, smaller than or equal to the delay set by the member function **set_max_delay**. If the argument *delay* is not specified, the time continuous delay shall be set to the value **sc_core::SC_ZERO_TIME**.

If the state vector *state* is not explicitly used as argument, the states shall be stored internally. If the size of the state vector is zero, its size shall be defined by the member function **calculate** or **operator=** and the vector elements shall be initialized to zero. Otherwise, the size of the state vector shall be consistent with the numerator and denumerator sizes. The relation between the numerator and denumerator sizes and the size of the state vector is implementation-defined.

If the timestep value *tstep* is not specified as argument, or if it is set to the value **sc_core::SC_ZERO_TIME**, the member function **calculate** and **operator()** shall define a timestep value equal to the time distance between the time reached by the last execution of the member function **calculate** and the time of the current activation of the module derived from class **sca_tdf::sca_module**, in which the transfer function is embedded. A specified timestep shall be smaller or equal to the time distance between the time reached by the last execution of the member function **calculate** and the time of the current activation of the module derived from class **sca_tdf::sca_module**, in which the transfer function is embedded.

If a value of type double is used as input argument, the value shall be interpreted as forming a continuous-time signal from the end of the last calculation time interval to the end of the current time interval. If a vector of class **sca_util::sca_vector**<double> is used as input argument, the values shall be interpreted as forming a continuous-time signal of equidistant distributed samples from the end of the last calculation time interval to the end of the current time interval. If a port of class **sca_tdf::sca_in**<double> or **sca_tdf::sca_de::sca_in**<double> is used as input argument, the samples available at the port shall be interpreted as forming a continuous-time signal using the associated time points.

The output settling behavior resulting from a change of coefficients during simulation is implementation-defined. If the state vector is stored internally, the state vector shall reset to zero when such a change occurs.

In case the state vector elements are set to zero, the output value shall be zero as long as the input value is zero.

4.1.4.4. sca_tdf::sca_ltf_zp

4.1.4.4.1. Description

The class **sca_tdf::sca_ltf_zp** shall implement 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 argument *delay* is the time continuous delay applied to the values available at the input.

4.1.4.4.2. Class definition

```
namespace sca_tdf {

  class sca_ltf_zp : public sc_core::sc_object
  {
   public:
    sca_ltf_zp();
    explicit sca_ltf_zp( const char* );

    virtual const char* kind() const;

    void set_max_delay( const sca_core::sca_time& );
    void set_max_delay( double tstep, sc_core::sc_time_unit unit );

    sca_tdf::sca_ct_proxy†& calculate(
                            const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                            const sca_util::sca_vector<sca_util::sca_complex>& poles,
                                  sca_util::sca_vector<double>& state,
                                  double                        input,
                                  double                        k = 1.0,
                                  sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_proxy†& calculate(
                            const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                            const sca_util::sca_vector<sca_util::sca_complex>& poles,
                                  sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                  sca_util::sca_vector<double>& state,
                                  double                        input,
                                  double                        k = 1.0,
                                  sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_proxy†& calculate(
                            const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                            const sca_util::sca_vector<sca_util::sca_complex>& poles,
                                  sca_util::sca_vector<double>& state,
                            const sca_util::sca_vector<double>& input,
                                  double                        k = 1.0,
                                  sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_proxy†& calculate(
                            const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                            const sca_util::sca_vector<sca_util::sca_complex>& poles,
                                  sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                  sca_util::sca_vector<double>& state,
                            const sca_util::sca_vector<double>& input,
                                  double                        k = 1.0,
                                  sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_proxy†& calculate(
                            const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                            const sca_util::sca_vector<sca_util::sca_complex>& poles,
                                  sca_util::sca_vector<double>& state,
```

```
                        const sca_tdf::sca_in<double>&      input,
                              double                        k = 1.0 );

sca_tdf::sca_ct_proxy†& calculate(
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                              sca_util::sca_vector<double>& state,
                        const sca_tdf::sca_in<double>&      input,
                              double                        k = 1.0 );

sca_tdf::sca_ct_proxy†& calculate(
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_util::sca_vector<double>&    state,
                        const sca_tdf::sca_de::sca_in<double>& input,
                              double                           k = 1.0 );

sca_tdf::sca_ct_proxy†& calculate(
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                              sca_util::sca_vector<double>&    state,
                        const sca_tdf::sca_de::sca_in<double>& input,
                              double                           k = 1.0 );

sca_tdf::sca_ct_proxy†& calculate(
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              double input,
                              double k = 1.0,
                              sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy†& calculate(
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                              double input,
                              double k = 1.0,
                              sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy†& calculate(
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                        const sca_util::sca_vector<double>& input,
                              double                        k = 1.0,
                              sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy†& calculate(
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                        const sca_util::sca_vector<double>& input,
                              double                        k = 1.0,
                              sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy†& calculate(
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                        const sca_tdf::sca_in<double>& input,
                              double                   k = 1.0 );

sca_tdf::sca_ct_proxy†& calculate(
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                        const sca_tdf::sca_in<double>& input,
                              double                   k = 1.0 );

sca_tdf::sca_ct_proxy†& calculate(
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                        const sca_tdf::sca_de::sca_in<double>& input,
                              double                           k = 1.0 );

sca_tdf::sca_ct_proxy†& calculate(
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                        const sca_tdf::sca_de::sca_in<double>& input,
                              double                           k = 1.0 );

sca_tdf::sca_ct_proxy†& operator() (
```



```
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_util::sca_vector<double>& state,
                              double                        input,
                              double                        k = 1.0,
                              sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy†& operator() (
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                              sca_util::sca_vector<double>& state,
                              double                        input,
                              double                        k = 1.0,
                              sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy†& operator() (
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_util::sca_vector<double>& state,
                        const sca_util::sca_vector<double>& input,
                              double                        k = 1.0,
                              sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy†& operator() (
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                              sca_util::sca_vector<double>& state,
                        const sca_util::sca_vector<double>& input,
                              double                        k = 1.0,
                              sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy†& operator() (
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_util::sca_vector<double>& state,
                        const sca_tdf::sca_in<double>&      input,
                              double                        k = 1.0 );

sca_tdf::sca_ct_proxy†& operator() (
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                              sca_util::sca_vector<double>& state,
                        const sca_tdf::sca_in<double>&      input,
                              double                        k = 1.0 );

sca_tdf::sca_ct_proxy†& operator() (
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_util::sca_vector<double>&    state,
                        const sca_tdf::sca_de::sca_in<double>& input,
                              double                           k = 1.0 );

sca_tdf::sca_ct_proxy†& operator() (
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                              sca_util::sca_vector<double>&    state,
                        const sca_tdf::sca_de::sca_in<double>& input,
                              double                           k = 1.0 );

sca_tdf::sca_ct_proxy†& operator() (
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              double input,
                              double k = 1.0,
                              sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy†& operator() (
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                              sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                              double input,
                              double k = 1.0,
                              sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_proxy†& operator() (
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                        const sca_util::sca_vector<double>& input,
                              double                        k = 1.0,
```

```
                                sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_proxy†& operator() (
                            const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                            const sca_util::sca_vector<sca_util::sca_complex>& poles,
                                  sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                            const sca_util::sca_vector<double>& input,
                                  double                        k = 1.0,
                                  sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_proxy†& operator() (
                            const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                            const sca_util::sca_vector<sca_util::sca_complex>& poles,
                            const sca_tdf::sca_in<double>& input,
                                  double                   k = 1.0 );

    sca_tdf::sca_ct_proxy†& operator() (
                            const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                            const sca_util::sca_vector<sca_util::sca_complex>& poles,
                                  sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                            const sca_tdf::sca_in<double>& input,
                                  double                   k = 1.0 );

    sca_tdf::sca_ct_proxy†& operator() (
                            const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                            const sca_util::sca_vector<sca_util::sca_complex>& poles,
                            const sca_tdf::sca_de::sca_in<double>& input,
                                  double                           k = 1.0 );

    sca_tdf::sca_ct_proxy†& operator() (
                            const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                            const sca_util::sca_vector<sca_util::sca_complex>& poles,
                                  sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                            const sca_tdf::sca_de::sca_in<double>& input,
                                  double                           k = 1.0 );
  };

} // namespace sca_tdf
```

4.1.1.4.4.3. Constructors

```
sca_ltf_zp();

explicit sca_ltf_zp( const char* );
```

The constructor for class **sca_tdf::sca_ltf_zp** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sc_core::sc_object** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_ltf_zp**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sc_core::sc_object**.

4.1.1.4.4.4. Constraint on usage

The expansion of the numerator and the denominator shall result in a real value, respectively.

4.1.1.4.4.5. kind

```
virtual const char* kind() const;
```

The member function kind shall return the string “**sca_tdf::sca_ltf_zp**”.

4.1.1.4.4.6. set_max_delay

```
void set_max_delay( const sca_core::sca_time& );

void set_max_delay( double tstep, sc_core::sc_time_unit unit );
```

The member function **set_max_delay** shall define the maximum allowable time continuous delay of the input values. If the member function is not called, the maximum allowable delay shall be set to the current timestep used, at which the input values are available. It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.4.4.7. calculate, operator()

```
sca_tdf::sca_ct_proxy†& calculate(...);

sca_tdf::sca_ct_proxy†& operator() (...);
```

The member function **calculate** and **operator()** shall return the continuous-time signal of the Laplace-domain variable *s* in the zero-pole form, using a reference to the class **sca_tdf::***sca_ct_proxy*†.

The arguments include the gain of the transfer function *k*, the vectors of the zero coefficients *zeros* and pole coefficients *poles*, the time continuous delay *delay*, the state vector *state*, the value of the input at the current time *input*, and the timestep *tstep*.

Each element of the vectors *zeros* and *poles* shall define a root of the transfer function. The root shall be a value of type **sca_util::sca_complex**. It shall be an error if the expansion of the zeros and poles has a nonzero imaginary part. If the size of the vector *zeros*, respectively *poles*, is zero, then the numerator, respectively the denominator, of the transfer function shall be equal to the value 1.0 + 0.0*j*.

The argument *delay* specifies the time continuous delay which shall be applied to the input values before calculating the linear transfer function. The delay shall be smaller than or equal to the current timestep used, at which the input values are available, or if the member function **set_max_delay** has been called, smaller than or equal to the delay set by the member function **set_max_delay**. If the argument *delay* is not specified, the time continuous delay shall be set to the value **sc_core::SC_ZERO_TIME**.

If the state vector *state* is not explicitly used as argument, the states shall be stored internally. If the size of the state vector is zero, its size shall be defined by the member function **calculate** or **operator()** and the vector elements shall be initialized to zero. Otherwise, the size of the state vector shall be consistent with the numerator and denumerator sizes. The relation between the numerator and denumerator sizes and the size of the state vector is implementation-defined.

If the timestep value *tstep* is not specified as argument, or if it is set to the value **sc_core::SC_ZERO_TIME**, the member function **calculate** and **operator()** shall define a timestep value equal to the time distance between the time reached by the last execution of the member function **calculate** and the time of the current activation of the module derived from class **sca_tdf::sca_module**, in which the transfer function is embedded. A specified timestep shall be smaller or equal to the time distance between the time reached by the last execution of the member function **calculate** and the time of the current activation of the module derived from class **sca_tdf::sca_module**, in which the transfer function is embedded.

If a value of type double is used as input argument, the value shall be interpreted as forming a continuous-time signal from the end of the last calculation time interval to the end of the current time interval. If a vector of class **sca_util::sca_vector**<double> is used as input argument, the values shall be interpreted as forming a continuous-time signal of equidistant distributed samples from the end of the last calculation time interval to the end of the current time interval. If a port of class **sca_tdf::sca_in**<double> or **sca_tdf::sca_de::sca_in**<double> is used as input argument, the samples available at the port shall be interpreted as forming a continuous-time signal using the associated time points.

The output settling behavior resulting from a change of coefficients during simulation is implementation-defined. If the state vector is stored internally the state vector shall reset to zero when such a change occurs.

In case the state vector elements are set to zero, the output value shall be zero as long as the input value is zero.

4.1.4.5. sca_tdf::sca_ss

4.1.4.5.1. Description

The class **sca_tdf::sca_ss** shall implement a system whose 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 argument *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.

4.1.4.5.2. Class definition

```
namespace sca_tdf {

  class sca_ss : public sc_core::sc_object
  {
   public:
    sca_ss();
    explicit sca_ss( const char* );

    virtual const char* kind() const;

    void set_max_delay( const sca_core::sca_time& );
    void set_max_delay( double tstep, sc_core::sc_time_unit unit );


    sca_tdf::sca_ct_vector_proxy†& calculate(
                                  const sca_util::sca_matrix<double>& a, // matrix A
                                  const sca_util::sca_matrix<double>& b, // matrix B
                                  const sca_util::sca_matrix<double>& c, // matrix C
                                  const sca_util::sca_matrix<double>& d, // matrix D
                                        sca_util::sca_vector<double>& s, // state vector s(t)
                                  const sca_util::sca_vector<double>& x, // input vector x(t)
                                        sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_vector_proxy†& calculate(
                                  const sca_util::sca_matrix<double>& a, // matrix A
                                  const sca_util::sca_matrix<double>& b, // matrix B
                                  const sca_util::sca_matrix<double>& c, // matrix C
                                  const sca_util::sca_matrix<double>& d, // matrix D
                                        sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                        sca_util::sca_vector<double>& s, // state vector s(t)
                                  const sca_util::sca_vector<double>& x, // input vector x(t)
                                        sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_vector_proxy†& calculate(
                                  const sca_util::sca_matrix<double>& a,
                                  const sca_util::sca_matrix<double>& b,
                                  const sca_util::sca_matrix<double>& c,
                                  const sca_util::sca_matrix<double>& d,
                                        sca_util::sca_vector<double>& s,
                                  const sca_util::sca_matrix<double>& x,
                                        sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_vector_proxy†& calculate(
                                  const sca_util::sca_matrix<double>& a,
                                  const sca_util::sca_matrix<double>& b,
                                  const sca_util::sca_matrix<double>& c,
                                  const sca_util::sca_matrix<double>& d,
                                        sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                        sca_util::sca_vector<double>& s,
                                  const sca_util::sca_matrix<double>& x,
                                        sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_vector_proxy†& calculate(
                                  const sca_util::sca_matrix<double>& a,
                                  const sca_util::sca_matrix<double>& b,
                                  const sca_util::sca_matrix<double>& c,
                                  const sca_util::sca_matrix<double>& d,
                                        sca_util::sca_vector<double>& s,
                                  const sca_tdf::sca_in< sca_util::sca_vector<double> >& x );

    sca_tdf::sca_ct_vector_proxy†& calculate(
                                  const sca_util::sca_matrix<double>& a,
                                  const sca_util::sca_matrix<double>& b,
                                  const sca_util::sca_matrix<double>& c,
                                  const sca_util::sca_matrix<double>& d,
                                        sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                        sca_util::sca_vector<double>& s,
                                  const sca_tdf::sca_in< sca_util::sca_vector<double> >& x );
```



```
sca_tdf::sca_ct_vector_proxy†& calculate(
                              const sca_util::sca_matrix<double>& a,
                              const sca_util::sca_matrix<double>& b,
                              const sca_util::sca_matrix<double>& c,
                              const sca_util::sca_matrix<double>& d,
                                    sca_util::sca_vector<double>& s,
                              const sca_tdf::sca_de::sca_in< sca_util::sca_vector<double> >& x );

sca_tdf::sca_ct_vector_proxy†& calculate(
                              const sca_util::sca_matrix<double>& a,
                              const sca_util::sca_matrix<double>& b,
                              const sca_util::sca_matrix<double>& c,
                              const sca_util::sca_matrix<double>& d,
                                    sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                    sca_util::sca_vector<double>& s,
                              const sca_tdf::sca_de::sca_in< sca_util::sca_vector<double> >& x );

sca_tdf::sca_ct_vector_proxy†& calculate(
                              const sca_util::sca_matrix<double>& a,
                              const sca_util::sca_matrix<double>& b,
                              const sca_util::sca_matrix<double>& c,
                              const sca_util::sca_matrix<double>& d,
                              const sca_util::sca_vector<double>& x,
                                    sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_vector_proxy†& calculate(
                              const sca_util::sca_matrix<double>& a,
                              const sca_util::sca_matrix<double>& b,
                              const sca_util::sca_matrix<double>& c,
                              const sca_util::sca_matrix<double>& d,
                                    sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                              const sca_util::sca_vector<double>& x,
                                    sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_vector_proxy†& calculate(
                              const sca_util::sca_matrix<double>& a,
                              const sca_util::sca_matrix<double>& b,
                              const sca_util::sca_matrix<double>& c,
                              const sca_util::sca_matrix<double>& d,
                              const sca_util::sca_matrix<double>& x,
                                    sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_vector_proxy†& calculate(
                              const sca_util::sca_matrix<double>& a,
                              const sca_util::sca_matrix<double>& b,
                              const sca_util::sca_matrix<double>& c,
                              const sca_util::sca_matrix<double>& d,
                                    sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                              const sca_util::sca_matrix<double>& x,
                                    sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_vector_proxy†& calculate(
                              const sca_util::sca_matrix<double>& a,
                              const sca_util::sca_matrix<double>& b,
                              const sca_util::sca_matrix<double>& c,
                              const sca_util::sca_matrix<double>& d,
                              const sca_tdf::sca_in< sca_util::sca_vector<double> >& x );

sca_tdf::sca_ct_vector_proxy†& calculate(
                              const sca_util::sca_matrix<double>& a,
                              const sca_util::sca_matrix<double>& b,
                              const sca_util::sca_matrix<double>& c,
                              const sca_util::sca_matrix<double>& d,
                                    sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                              const sca_tdf::sca_in< sca_util::sca_vector<double> >& x );

sca_tdf::sca_ct_vector_proxy†& calculate(
                              const sca_util::sca_matrix<double>& a,
                              const sca_util::sca_matrix<double>& b,
                              const sca_util::sca_matrix<double>& c,
                              const sca_util::sca_matrix<double>& d,
                              const sca_tdf::sca_de::sca_in< sca_util::sca_vector<double> >& x );

sca_tdf::sca_ct_vector_proxy†& calculate(
                              const sca_util::sca_matrix<double>& a,
                              const sca_util::sca_matrix<double>& b,
                              const sca_util::sca_matrix<double>& c,
                              const sca_util::sca_matrix<double>& d,
                                    sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                              const sca_tdf::sca_de::sca_in< sca_util::sca_vector<double> >& x );

sca_tdf::sca_ct_vector_proxy†& operator() (
```

```
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
                                const sca_util::sca_matrix<double>& d,
                                      sca_util::sca_vector<double>& s,
                                const sca_util::sca_vector<double>& x,
                                      sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_vector_proxy†& operator() (
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
                                const sca_util::sca_matrix<double>& d,
                                      sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                      sca_util::sca_vector<double>& s,
                                const sca_util::sca_vector<double>& x,
                                      sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_vector_proxy†& operator() (
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
                                const sca_util::sca_matrix<double>& d,
                                      sca_util::sca_vector<double>& s,
                                const sca_util::sca_matrix<double>& x,
                                      sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_vector_proxy†& operator() (
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
                                const sca_util::sca_matrix<double>& d,
                                      sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                      sca_util::sca_vector<double>& s,
                                const sca_util::sca_matrix<double>& x,
                                      sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_vector_proxy†& operator() (
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
                                const sca_util::sca_matrix<double>& d,
                                      sca_util::sca_vector<double>& s,
                                const sca_tdf::sca_in< sca_util::sca_vector<double> >& x );

sca_tdf::sca_ct_vector_proxy†& operator() (
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
                                const sca_util::sca_matrix<double>& d,
                                      sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                      sca_util::sca_vector<double>& s,
                                const sca_tdf::sca_in< sca_util::sca_vector<double> >& x );

sca_tdf::sca_ct_vector_proxy†& operator() (
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
                                const sca_util::sca_matrix<double>& d,
                                      sca_util::sca_vector<double>& s,
                                const sca_tdf::sca_de::sca_in< sca_util::sca_vector<double> >& x );

sca_tdf::sca_ct_vector_proxy†& operator() (
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
                                const sca_util::sca_matrix<double>& d,
                                      sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                      sca_util::sca_vector<double>& s,
                                const sca_tdf::sca_de::sca_in< sca_util::sca_vector<double> >& x );

sca_tdf::sca_ct_vector_proxy†& operator() (
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
                                const sca_util::sca_matrix<double>& d,
                                const sca_util::sca_vector<double>& x,
                                      sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

sca_tdf::sca_ct_vector_proxy†& operator() (
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
```



```
                                const sca_util::sca_matrix<double>& d,
                                      sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                const sca_util::sca_vector<double>& x,
                                      sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_vector_proxy†& operator() (
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
                                const sca_util::sca_matrix<double>& d,
                                const sca_util::sca_matrix<double>& x,
                                      sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_vector_proxy†& operator() (
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
                                const sca_util::sca_matrix<double>& d,
                                      sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                const sca_util::sca_matrix<double>& x,
                                      sca_core::sca_time tstep = sc_core::SC_ZERO_TIME );

    sca_tdf::sca_ct_vector_proxy†& operator() (
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
                                const sca_util::sca_matrix<double>& d,
                                const sca_tdf::sca_in< sca_util::sca_vector<double> >& x );

    sca_tdf::sca_ct_vector_proxy†& operator() (
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
                                const sca_util::sca_matrix<double>& d,
                                      sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                const sca_tdf::sca_in< sca_util::sca_vector<double> >& x );

    sca_tdf::sca_ct_vector_proxy†& operator() (
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
                                const sca_util::sca_matrix<double>& d,
                                const sca_tdf::sca_de::sca_in< sca_util::sca_vector<double> >& x );

    sca_tdf::sca_ct_vector_proxy†& operator() (
                                const sca_util::sca_matrix<double>& a,
                                const sca_util::sca_matrix<double>& b,
                                const sca_util::sca_matrix<double>& c,
                                const sca_util::sca_matrix<double>& d,
                                      sca_core::sca_time delay = sc_core::SC_ZERO_TIME,
                                const sca_tdf::sca_de::sca_in< sca_util::sca_vector<double> >& x );
  };

} // namespace sca_tdf
```

4.1.4.5.3. Constructors

```
sca_ss();

explicit sca_ss( const char* );
```

The constructor for class **sca_tdf::sca_ss** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sc_core::sc_object** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_ss**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sc_core::sc_object**.

4.1.4.5.4. kind

```
virtual const char* kind() const;
```

The member function kind shall return the string “**sca_tdf::sca_ss**”.

4.1.4.5.5. set_max_delay

```
void set_max_delay( const sca_core::sca_time& );

void set_max_delay( double tstep, sc_core::sc_time_unit unit );
```


The member function **set_max_delay** shall define the maximum allowable time continuous delay of the input values. If the member function is not called, the maximum allowable delay shall be set to the current timestep used, at which the input values are available. It shall be an error if the member function is called outside the member function **sca_tdf::sca_module::set_attributes** of the current module (see 4.1.1.1.5).

4.1.4.5.6. calculate, operator()

```
sca_tdf::sca_ct_vector_proxy†& calculate(...);

sca_tdf::sca_ct_vector_proxy†& operator() (...);
```

The member function **calculate** and **operator()** shall return the continuous-time signal of the state-space equation system using a reference to the class **sca_tdf::***sca_ct_vector_proxy*†.

The arguments include the matrices *a*, *b*, *c*, and *d*, the time continuous delay *delay* , the state vector *s*, the input vector *x* and the timestep *tstep*. It shall be an error if one of the following conditions is not met:

1. Argument *a* shall be a square matrix of the size of state vector *s*.
2. The number of columns in matrix *b* and the number of columns in matrix *d* is equal to the size of the input vector *x*.
3. The number of rows in matrix *b* and the number of columns in matrix *c* is equal to the size of the state vector *s*.
4. The number of rows in matrices *c* and *d* is equal to the size of the output vector *y*.

The value of the state vector shall be kept after a change of values of matrix coefficients.

The argument *delay* specifies the time continuous delay which shall be applied to the input values before calculating the linear transfer function. The delay shall be smaller than or equal to the current timestep used, at which the input values are available, or if the member function **set_max_delay** has been called, smaller than or equal to the delay set by the member function **set_max_delay**. If the argument *delay* is not specified, the time continuous delay shall be set to the value **sc_core::SC_ZERO_TIME**.

If the state vector *state* is not explicitly used as argument, the states shall be stored internally. If the size of the state vector is zero, its size shall be defined by the member function **calculate** or **operator=** and the vector elements shall be initialized to zero. Otherwise, the size of the state vector shall be consistent with the coefficient matrix sizes.

If the timestep value *tstep* is not specified as argument, or if it is set to the value **sc_core::SC_ZERO_TIME**, the member function **calculate** and **operator()** shall define a timestep value equal to the time distance between the time reached by the last execution of the member function **calculate** and the time of the current activation of the module derived from class **sca_tdf::sca_module**, in which the state space equation is embedded. A specified timestep shall be smaller or equal to the time distance between the time reached by the last execution of the member function **calculate** and the time of the current activation of the module derived from class **sca_tdf::sca_module**, in which the state space equation is embedded.

If a vector of class **sca_util::sca_vector**<double> is used as input argument, the values shall be interpreted as forming a continuous-time signal of equidistant distributed samples from the end of the last calculation time interval to the end of the current time interval. If a matrix of class **sca_util::sca_matrix**<double> is used as input argument, the matrix columns shall be interpreted as forming continuous-time signal of equidistant distributed samples from the end of the last calculation time interval to the end of the current time interval. If a port of class **sca_tdf::sca_in**< **sca_util::sca_vector**<double> > or **sca_tdf::sca_de::sca_in**< **sca_util::sca_vector**<double> > is used as input argument, the samples available at the port shall be interpreted as forming a continuous-time signal using the associated time points.

4.2. Linear signal flow model of computation

The LSF model of computation shall define the behavior of non-conservative continuous-time systems as mathematical relations between quantities represented by real-value functions of the independent variable time. The resulting differential and algebraic equation system which is defined by the set of connected predefined LSF primitive modules shall be solved during simulation. The mathematical relation defined by each LSF primitive module shall contribute to this overall equation system. The predefined set of

LSF primitive modules shall support the basic operators required to define LSF behavior as defined in this subclause.

4.2.1. LSF class definitions

All names used in the LSF class definitions shall be placed in the namespace **sca_lsf**.

4.2.1.1. sca_lsf::sca_module

4.2.1.1.1. Description

The class **sca_lsf::sca_module** shall define the base class for all LSF primitive modules. An application shall not derive from this class directly, but shall use the predefined primitive modules as defined in the following clauses.

4.2.1.1.2. Class definition

```
namespace sca_lsf {

  class sca_module : public sca_core::sca_module
  {
   public:
    virtual const char* kind() const;

   protected:
    sca_module();
    virtual ~sca_module();
  };
} // namespace sca_lsf
```

4.2.1.2. sca_lsf::sca_signal_if

4.2.1.2.1. Description

The class **sca_lsf::sca_signal_if** shall define an interface proper for a primitive channel of class **sca_lsf::sca_signal**. The interface class member functions are implementation-defined.

4.2.1.2.2. Class definition

```
namespace sca_lsf {

  class sca_signal_if : public sca_core::sca_interface
  {
   protected:
    sca_signal_if();

   private:
    // Other members
    implementation-defined

    // Disabled
    sca_signal_if( const sca_lsf::sca_signal_if& );
    sca_lsf::sca_signal_if& operator= ( const sca_lsf::sca_signal_if& );
  };

} // namespace sca_lsf
```

4.2.1.3. sca_lsf::sca_signal

4.2.1.3.1. Description

The class **sca_lsf::sca_signal** shall define a primitive channel for the LSF MoC. It shall be used for connecting modules derived from class **sca_lsf::sca_module** using ports of class **sca_lsf::sca_in** and **sca_lsf::sca_out**. An application shall not access the associated interface directly.

4.2.1.3.2. Class definition

```
namespace sca_lsf {

  class sca_signal : public sca_lsf::sca_signal_if,
                     public sca_core::sca_prim_channel
  {
```

```
   public:
    sca_signal();
    explicit sca_signal( const char* );

    virtual const char* kind() const;

   private:
    // Disabled
    sca_signal( const sca_lsf::sca_signal& );
  };

} // namespace sca_lsf
```

4.2.1.3.3. Constructors

```
sca_signal();

explicit sca_signal( const char* );
```

The constructor for class **sca_lsf::sca_signal** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sca_core::sca_prim_channel** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_lsf_signal**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sca_core::sca_prim_channel**.

4.2.1.3.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string “**sca_lsf::sca_signal**”.

4.2.1.4. sca_lsf::sca_in

4.2.1.4.1. Description

The class **sca_lsf::sca_in** shall define a port class for the LSF MoC.

4.2.1.4.2. Class definition

```
namespace sca_lsf {

  class sca_in : public sca_core::sca_port< sca_lsf::sca_signal_if >
  {
   public:
    sca_in();
    explicit sca_in( const char* );

    virtual const char* kind() const;

   private:
    // Other members
    implementation-defined

    // Disabled
    sca_in( const sca_lsf::sca_in& );
  };

} // namespace sca_lsf
```

4.2.1.4.3. Constructors

```
sca_in();

explicit sca_in( const char* );
```

The constructor for class **sca_lsf::sca_in** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sca_core::sca_port** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_lsf_in**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sca_core::sca_port**.

4.2.1.4.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_in**".

4.2.1.5. sca_lsf::sca_out

4.2.1.5.1. Description

The class **sca_lsf::sca_out** shall define a port class for the LSF MoC.

4.2.1.5.2. Class definition

```
namespace sca_lsf {

  class sca_out : public sca_core::sca_port< sca_lsf::sca_signal_if >
  {
   public:
    sca_out();
    explicit sca_out( const char* );

    virtual const char* kind() const;

   private:
    // Other members
    implementation-defined

    // Disabled
    sca_out( const sca_lsf::sca_out& );
  };

} // namespace sca_lsf
```

4.2.1.5.3. Constructors

```
sca_out();

explicit sca_out( const char* );
```

The constructor for class **sca_lsf::sca_out** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sca_core::sca_port** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**("**sca_lsf_out**") to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sca_core::sca_port**.

4.2.1.5.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_out**".

4.2.1.6. sca_lsf::sca_add

4.2.1.6.1. Description

The class **sca_lsf::sca_add** shall implement a primitive module for the LSF MoC that realizes the weighted addition of two LSF signals. The primitive shall contribute the following equation to the equation system:

$$y(t) = k_1 \cdot x_1(t) + k_2 \cdot x_2(t)$$

where $x_1(t)$ and $x_2(t)$ are the two LSF input signals, k_1 and k_2 are constant weighting coefficients, and $y(t)$ is the LSF output signal.

4.2.1.6.2. Class definition

```
namespace sca_lsf {

  class sca_add : public sca_lsf::sca_module
```



```
  {
   public:
    sca_lsf::sca_in  x1; // LSF inputs
    sca_lsf::sca_in  x2;

    sca_lsf::sca_out y;  // LSF output

    sca_core::sca_parameter<double> k1; // weighting coefficients
    sca_core::sca_parameter<double> k2;

    virtual const char* kind() const;

    explicit sca_add( sc_core::sc_module_name, double k1_ = 1.0, double k2_ = 1.0 )
      : x1( "x1" ), x2( "x2" ), y( "y" ), k1( "k1", k1_ ), k2( "k2", k2_ )
    { implementation-defined }
  };

} // namespace sca_lsf
```

4.2.1.6.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_add**".

4.2.1.7. sca_lsf::sca_sub

4.2.1.7.1. Description

The class **sca_lsf::sca_sub** shall implement a primitive module for the LSF MoC that realizes the weighted subtraction of two LSF signals. The primitive shall contribute the following equation to the equation system:

$$y(t) = k_1 \cdot x_1(t) - k_2 \cdot x_2(t)$$

where $x_1(t)$ and $x_2(t)$ are the two LSF input signals, k_1 and k_2 are constant weighting coefficients, and $y(t)$ is the LSF output signal.

4.2.1.7.2. Class definition

```
namespace sca_lsf {

  class sca_sub : public sca_lsf::sca_module
  {
   public:
    sca_lsf::sca_in  x1; // LSF inputs
    sca_lsf::sca_in  x2;

    sca_lsf::sca_out y;  // LSF output

    sca_core::sca_parameter<double> k1; // weighting coefficients
    sca_core::sca_parameter<double> k2;

    virtual const char* kind() const;

    explicit sca_sub( sc_core::sc_module_name, double k1_ = 1.0, double k2_ = 1.0 )
      : x1( "x1" ), x2( "x2" ), y( "y" ), k1( "k1", k1_ ), k2( "k2", k2_ )
    { implementation-defined }
  };

} // namespace sca_lsf
```

4.2.1.7.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_sub**".

4.2.1.8. sca_lsf::sca_gain

4.2.1.8.1. Description

The class **sca_lsf::sca_gain** shall implement a primitive module for the LSF MoC that realizes the multiplication of an LSF signal by a constant gain. The primitive shall contribute the following equation to the equation system:

$$y(t) = k \cdot x(t)$$

where k is the constant gain coefficient, $x(t)$ is the LSF input signal, and $y(t)$ is the LSF output signal.

4.2.1.8.2. Class definition

```
namespace sca_lsf {

  class sca_gain : public sca_lsf::sca_module
  {
   public:
    sca_lsf::sca_in  x; // LSF input

    sca_lsf::sca_out y; // LSF output

    sca_core::sca_parameter<double> k; // gain coefficient

    virtual const char* kind() const;

    explicit sca_gain( sc_core::sc_module_name, double k_ = 1.0 )
      : x( "x" ), y( "y" ), k( "k", k_ )
    { implementation-defined }
  };

} // namespace sca_lsf
```

4.2.1.8.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_gain**".

4.2.1.9. sca_lsf::sca_dot

4.2.1.9.1. Description

The class **sca_lsf::sca_dot** shall implement a primitive module for the LSF MoC that realizes the scaled first-order time derivative of an LSF signal. The primitive shall contribute the following equation to the equation system:

$$y(t) = k \cdot \frac{dx(t)}{dt}$$

where k is the constant scale coefficient, $x(t)$ is the LSF input signal, and $y(t)$ is the LSF output signal.

4.2.1.9.2. Class definition

```
namespace sca_lsf {

  class sca_dot : public sca_lsf::sca_module
  {
   public:
    sca_lsf::sca_in  x; // LSF input

    sca_lsf::sca_out y; // LSF output

    sca_core::sca_parameter<double> k; // scale coefficient

    virtual const char* kind() const;

    explicit sca_dot( sc_core::sc_module_name, double k_ = 1.0 )
      : x( "x" ), y( "y" ), k( "k", k_ )
    { implementation-defined }
  };

} // namespace sca_lsf
```

4.2.1.9.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_dot**".

4.2.1.10. sca_lsf::sca_integ

4.2.1.10.1. Description

The class **sca_lsf::sca_integ** shall implement a primitive module for the LSF MoC that realizes the scaled time-domain integration of an LSF signal. The primitive shall contribute the following equation to the equation system:

$$y(t) = k \cdot \int_o^t x(t)dt + y_0$$

where k is the constant scale coefficient, $x(t)$ is the LSF input signal, y_0 is the initial condition at $t = 0$, and $y(t)$ is the LSF output signal. The integration is done from time $t = 0$ to the current time t.

4.2.1.10.2. Class definition

```
namespace sca_lsf {

  class sca_integ : public sca_lsf::sca_module
  {
   public:
    sca_lsf::sca_in  x; // LSF input

    sca_lsf::sca_out y; // LSF output

    sca_core::sca_parameter<double> k;  // scale coefficient
    sca_core::sca_parameter<double> y0; // initial condition at t=0

    virtual const char* kind() const;

    explicit sca_integ( sc_core::sc_module_name, double k_ = 1.0, double y0_ = 0.0 )
      : x( "x" ), y( "y" ), k( "k", k_ ), y0( "y0", y0_ )
    { implementation-defined }
  };

} // namespace sca_lsf
```

4.2.1.10.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_integ**".

4.2.1.11. sca_lsf::sca_delay

4.2.1.11.1. Description

The class **sca_lsf::sca_delay** shall implement a primitive module for the LSF MoC that generates a scaled time-delayed version of an LSF signal. The primitive shall contribute the following equation to the equation system:

$$y(t) = \begin{cases} y_0 & t \le delay \\ k \cdot x(t - delay) & t > delay \end{cases}$$

where t is the time, $delay$ is the time delay in seconds, k is the constant scale coefficient, $x(t)$ is the LSF input signal, y_0 is the output value before the delay is in effect, and $y(t)$ is the LSF output signal.

4.2.1.11.2. Class definition

```
namespace sca_lsf {

  class sca_delay : public sca_lsf::sca_module
  {
   public:
    sca_lsf::sca_in  x; // LSF input

    sca_lsf::sca_out y; // LSF output

    sca_core::sca_parameter<sca_core::sca_time> delay; // time delay
    sca_core::sca_parameter<double>             k;     // scale coefficient
```



```
    sca_core::sca_parameter<double>             y0;    // output value before delay is in effect

    virtual const char* kind() const;

    explicit sca_delay( sc_core::sc_module_name, sca_core::sca_time delay_ = sc_core::SC_ZERO_TIME,
                                                 double k_  = 1.0,
                                                 double y0_ = 0.0 )
      : x( "x" ), y( "y" ), delay( "delay", delay_ ), k( "k", k_ ), y0( "y0", y0_ )
    { implementation-defined }
  };

} // namespace sca_lsf
```

4.2.1.11.3. Constraint of usage

The delay shall be greater or equal to zero.

4.2.1.11.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_delay**".

4.2.1.12. sca_lsf::sca_source

4.2.1.12.1. Description

The class **sca_lsf::sca_source** shall implement a primitive module for the LSF MoC that realizes a source for an LSF signal. In time-domain simulation, the primitive shall contribute the following equation to the equation system:

$$y(t) = \begin{cases} init\_value & t < delay \\ offset + amplitude \cdot \sin(2\pi \cdot frequency \cdot (t - delay) + phase) & t \geq delay \end{cases}$$

where *t* is the time, *delay* is the initial delay in seconds, *init_value* is the initial value, *offset* is the offset, *amplitude* is the source amplitude, *frequency* is the source frequency in hertz, *phase* is the source phase in radians, π is the pi constant, and $y(t)$ is the LSF output signal. Source parameters shall be set to zero by default.

In small-signal frequency-domain simulation, the primitive shall contribute the following equation to the equation system:

$$y(f) = ac\_amplitude \cdot \{\cos(ac\_phase) + j \cdot \sin(ac\_phase)\}$$

where *f* is the simulation frequency, *ac_amplitude* is the small-signal amplitude, and *ac_phase* is the small-signal phase in radians.

In small-signal frequency-domain noise simulation, the primitive shall contribute the following equation to the equation system:

$$y(f) = ac\_noise\_amplitude$$

where *f* is the simulation frequency, and *ac_noise_amplitude* is the small-signal noise amplitude.

4.2.1.12.2. Class definition

```
namespace sca_lsf {

  class sca_source : public sca_lsf::sca_module
  {
   public:
    sca_lsf::sca_out y; // LSF output

    sca_core::sca_parameter<double> init_value;
    sca_core::sca_parameter<double> offset;
    sca_core::sca_parameter<double> amplitude;
    sca_core::sca_parameter<double> frequency;
```

```
    sca_core::sca_parameter<double> phase;
    sca_core::sca_parameter<sca_core::sca_time> delay;
    sca_core::sca_parameter<double> ac_amplitude;
    sca_core::sca_parameter<double> ac_phase;
    sca_core::sca_parameter<double> ac_noise_amplitude;

    virtual const char* kind() const;

    explicit sca_source( sc_core::sc_module_name, double init_value_   = 0.0,
                                                  double offset_       = 0.0,
                                                  double amplitude_    = 0.0,
                                                  double frequency_    = 0.0,
                                                  double phase_        = 0.0,
                                                  sca_core::sca_time delay_ = sc_core::SC_ZERO_TIME,
                                                  double ac_amplitude_ = 0.0,
                                                  double ac_phase_     = 0.0,
                                                  double ac_noise_amplitude_ = 0.0 )
      : y( "y" ),
        init_value( "init_value", init_value_ ),
        offset( "offset", offset_ ),
        amplitude( "amplitude", amplitude_ ),
        frequency( "frequency", frequency_ ),
        phase( "phase", phase_ ),
        delay( "delay", delay_ ),
        ac_amplitude( "ac_amplitude", ac_amplitude_ ),
        ac_phase( "ac_phase", ac_phase_ ),
        ac_noise_amplitude( "ac_noise_amplitude", ac_noise_amplitude_ )
     { implementation-defined }
  };

} // namespace sca_lsf
```

4.2.1.12.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_source**".

4.2.1.13. sca_lsf::sca_ltf_nd

4.2.1.13.1. Description

The class **sca_lsf::sca_ltf_nd** shall implement a primitive module for the LSF MoC that realizes a scaled Laplace transfer function in the time-domain in the numerator-denominator form (see 4.1.4.3). The primitive shall contribute the following equation to the equation system:

$$den_{N-1}\frac{d^{N-1}y(t)}{dt} + den_{N-2}\frac{d^{N-2}y(t)}{dt} + \cdots + den_1\frac{dy(t)}{dt} + den_0 \cdot y(t)$$

$$= k \cdot \left(num_{M-1}\frac{d^{M-1}x(t-delay)}{dt} + num_{M-2}\frac{d^{M-2}x(t-delay)}{dt} + \cdots + num_1\frac{dx(t-delay)}{dt} + num_0 \cdot x(t-delay) \right)$$

where k is the constant gain coefficient, M and N are the number of numerator and denominator coefficients, respectively, indexed with i, $x(t)$ is the LSF input signal, num_i and den_i are real-valued coefficients of the numerator and denominator, respectively, *delay* is the time continuous delay in seconds, applied to the values available at the input, and $y(t)$ is the LSF output signal.

4.2.1.13.2. Class definition

```
namespace sca_lsf {

  class sca_ltf_nd : public sca_lsf::sca_module
  {
   public:
    sca_lsf::sca_in  x; // LSF input

    sca_lsf::sca_out y; // LSF output

    sca_core::sca_parameter< sca_util::sca_vector<double> > num;  // numerator coefficients
    sca_core::sca_parameter< sca_util::sca_vector<double> > den;  // denumerator coefficients
    sca_core::sca_parameter< sca_core::sca_time > delay; // time delay
    sca_core::sca_parameter< double >             k;     // gain coefficient
```



```
    virtual const char* kind() const;

    explicit sca_ltf_nd( sc_core::sc_module_name,
                const sca_util::sca_vector<double>& num_ = sca_util::sca_create_vector( 1.0 ),
                const sca_util::sca_vector<double>& den_ = sca_util::sca_create_vector( 1.0 ),
                      double k_ = 1.0 )
      : x( "x" ), y( "y" ), num( "num", num_ ), den( "den", den_ ),
        delay( "delay" , delay_ ), k( "k", k_ )
    { implementation-defined }

    sca_ltf_nd( sc_core::sc_module_name,
                const sca_util::sca_vector<double>& num_,
                const sca_util::sca_vector<double>& den_,
                      sca_core::sca_time delay_,
                      double k_ = 1.0 )
      : x( "x" ), y( "y" ), num( "num", num_ ), den( "den", den_ ),
        delay( "delay" , delay_ ), k( "k", k_ )
    { implementation-defined }
  };

} // namespace sca_lsf
```

4.2.1.13.3. Constraint on usage

The vectors *num* and *den* shall have at least one element, respectively.

4.2.1.13.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_ltf_nd**".

4.2.1.14. sca_lsf::sca_ltf_zp

4.2.1.14.1. Description

The class **sca_lsf::sca_ltf_zp** shall implement a primitive module for the LSF MoC that realizes a scaled Laplace transfer function in the time-domain in the zero-pole form (see 4.1.4.4). The primitive shall contribute the following equation to the equation system:

$$\left(\frac{d}{dt} - poles_{N-1}\right)\left(\frac{d}{dt} - poles_{N-2}\right)\cdots\left(\frac{d}{dt} - poles_{1}\right)\left(\frac{d}{dt} - poles_{0}\right)y(t)$$

$$= k \cdot \left\{\left(\frac{d}{dt} - zeros_{M-1}\right)\left(\frac{d}{dt} - zeros_{M-2}\right)\cdots\left(\frac{d}{dt} - zeros_{1}\right)\left(\frac{d}{dt} - zeros_{0}\right)x(t - delay)\right\}$$

where k is the constant gain coefficient, M and N are the number of zeros and poles, respectively, indexed with i, $x(t)$ is the LSF input signal, $zeros_i$ and $poles_i$ are complex-valued zeros and poles, respectively, *delay* is the time continuous delay in seconds applied to the values available at the input, and $y(t)$ is the LSF output signal.

4.2.1.14.2. Class definition

```
namespace sca_lsf {

  class sca_ltf_zp : public sca_lsf::sca_module
  {
   public:
    sca_lsf::sca_in  x; // LSF input

    sca_lsf::sca_out y; // LSF output

    sca_core::sca_parameter< sca_util::sca_vector<sca_util::sca_complex> > zeros;
    sca_core::sca_parameter< sca_util::sca_vector<sca_util::sca_complex> > poles;
    sca_core::sca_parameter< sca_core::sca_time > delay; // time delay
    sca_core::sca_parameter< double >             k;     // gain coefficient

    virtual const char* kind() const;

    explicit sca_ltf_zp( sc_core::sc_module_name,
                const sca_util::sca_vector<sca_util::sca_complex>& zeros_ =
                      sca_util::sca_vector<sca_util::sca_complex>(),
                const sca_util::sca_vector<sca_util::sca_complex>& poles_ =
```

```
                    sca_util::sca_vector<sca_util::sca_complex>(),
                    double k_ = 1.0 )
      : x( "x" ), y( "y" ), zeros( "zeros", zeros_ ), poles( "poles", poles_ ),
        delay( "delay", delay_ ), k( "k", k_ )
    { implementation-defined }

    sca_ltf_zp( sc_core::sc_module_name,
                const sca_util::sca_vector<sca_util::sca_complex>& zeros_,
                const sca_util::sca_vector<sca_util::sca_complex>& poles_,
                      sca_core::sca_time delay_,
                      double k_ = 1.0 )
      : x( "x" ), y( "y" ), zeros( "zeros", zeros_ ), poles( "poles", poles_ ),
        delay( "delay", delay_ ), k( "k", k_ )
    { implementation-defined }
  };

} // namespace sca_lsf
```

4.2.1.14.3. Constraint on usage

The expansion of the numerator and the denominator shall result in a real value, respectively. It shall be an error if after expansion the imaginary part is numerically not zero.

4.2.1.14.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_ltf_zp**".

4.2.1.15. sca_lsf::sca_ss

4.2.1.15.1. Description

The class **sca_lsf::sca_ss** shall implement a primitive module for the LSF MoC that realizes a system whose behavior is defined by single-input single-output state-space equations (see 4.1.4.5). The primitive shall contribute the following equation to the equation system:

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

where $s(t)$ is the state vector, $x(t)$ is the LSF input signal, *delay* is the time continuous delay in seconds applied to the values available at the input, and $y(t)$ is the LSF output signal. **A** is a n-by-n matrix, where n is the number of states, **B** and **C** are vectors of size n, and **D** is a real value.

4.2.1.15.2. Class definition

```
namespace sca_lsf {

  class sca_ss : public sca_lsf::sca_module
  {
   public:
    sca_lsf::sca_in  x; // LSF input

    sca_lsf::sca_out y; // LSF output

    sca_core::sca_parameter< sca_util::sca_matrix<double> > a; // matrix A of size n-by-n
    sca_core::sca_parameter< sca_util::sca_matrix<double> > b; // matrix B with one column of size n
    sca_core::sca_parameter< sca_util::sca_matrix<double> > c; // matrix C with one row of size n
    sca_core::sca_parameter< sca_util::sca_matrix<double> > d; // matrix D of size 1
    sca_core::sca_parameter< sca_core::sca_time > delay; // time delay

    virtual const char* kind() const;

    explicit sca_ss( sc_core::sc_module_name,
            const sca_util::sca_matrix<double>& a_ = sca_util::sca_matrix<double>(),
            const sca_util::sca_matrix<double>& b_ = sca_util::sca_matrix<double>(),
            const sca_util::sca_matrix<double>& c_ = sca_util::sca_matrix<double>(),
            const sca_util::sca_matrix<double>& d_ = sca_util::sca_matrix<double>(),
                  sca_core::sca_time delay_ = sc_core::SC_ZERO_TIME )
      : x( "x" ), y( "y" ), a( "a", a_ ), b( "b", b_ ), c( "c", c_ ), d( "d", d_ ), delay( "delay", delay_ )
    { implementation-defined }
  };
```



```
} // namespace sca_lsf
```

4.2.1.15.3. Constraint on usage

It shall be an error if one of the following conditions is not met:

1. Argument *a* shall be a square matrix of the size of state vector *s*.
2. Argument *b* shall be a matrix with one column and the size of state vector *s* rows.
3. Argument *c* shall be a matrix with one row and the size of state vector *s* columns.
4. Argument *d* shall be a matrix of one row and one column.

NOTE—The class **sca_lsf::sca_ss** uses matrices similar to class **sca_tdf::sca_ss**.

4.2.1.15.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_ss**".

4.2.1.16. sca_lsf::sca_tdf::sca_gain, sca_lsf::sca_tdf_gain

4.2.1.16.1. Description

The class **sca_lsf::sca_tdf::sca_gain** shall implement a primitive module for the LSF MoC that realizes the scaled multiplication of a TDF input signal by an LSF input signal. The primitive shall contribute the following equation to the equation system:

$$y(t) = scale \cdot inp \cdot x(t)$$

where *scale* is the constant scale coefficient, *inp* is the TDF input signal that shall be interpreted as a continuous-time signal, $x(t)$ is the LSF input signal, and $y(t)$ is the LSF output signal.

4.2.1.16.2. Class definition

```
namespace sca_lsf {

  namespace sca_tdf {

    class sca_gain : public sca_lsf::sca_module
    {
     public:
      ::sca_tdf::sca_in<double> inp; // TDF input

      sca_lsf::sca_in  x; // LSF input

      sca_lsf::sca_out y; // LSF output

      sca_core::sca_parameter<double> scale; // scale coefficient

      virtual const char* kind() const;

      explicit sca_gain( sc_core::sc_module_name, double scale_ = 1.0 )
        : inp( "inp" ), x( "x" ), y( "y" ), scale( "scale", scale_ )
      { implementation-defined }
    };

  } // namespace sca_tdf

  typedef sca_lsf::sca_tdf::sca_gain sca_tdf_gain;

} // namespace sca_lsf
```

4.2.1.16.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_tdf::sca_gain**".

4.2.1.17. sca_lsf::sca_tdf::sca_source, sca_lsf::sca_tdf_source

4.2.1.17.1. Description

The class **sca_lsf::sca_tdf::sca_source** shall implement a primitive module for the LSF MoC that realizes the scaled conversion of a TDF signal to an LSF signal. The primitive shall contribute the following equation to the equation system:

$$y(t) = scale \cdot inp$$

where *scale* is the constant scale coefficient, *inp* is the TDF input signal that shall be interpreted as a continuous-time signal, and $y(t)$ is the LSF output signal.

4.2.1.17.2. Class definition

```
namespace sca_lsf {

  namespace sca_tdf {

    class sca_source : public sca_lsf::sca_module
    {
     public:
      ::sca_tdf::sca_in<double> inp; // TDF input

      sca_lsf::sca_out y; // LSF output

      sca_core::sca_parameter<double> scale; // scale coefficient

      virtual const char* kind() const;

      explicit sca_source( sc_core::sc_module_name, double scale_ = 1.0 )
        : inp( "inp" ), y( "y" ), scale( "scale", scale_ )
      { implementation-defined }
    };

  } // namespace sca_tdf

  typedef sca_lsf::sca_tdf::sca_source sca_tdf_source;

} // namespace sca_lsf
```

4.2.1.17.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_tdf::sca_source**".

4.2.1.18. sca_lsf::sca_tdf::sca_sink, sca_lsf::sca_tdf_sink

4.2.1.18.1. Description

The class **sca_lsf::sca_tdf::sca_sink** shall implement a primitive module for the LSF MoC that realizes a scaled conversion from an LSF signal to a TDF signal. The value of the LSF input signal $x(t)$ shall be scaled with coefficient *scale* and written to the TDF output port *outp*.

4.2.1.18.2. Class definition

```
namespace sca_lsf {

  namespace sca_tdf {

    class sca_sink : public sca_lsf::sca_module
    {
     public:
      sca_lsf::sca_in x; // LSF input

      ::sca_tdf::sca_out<double> outp; // TDF output

      sca_core::sca_parameter<double> scale; // scale coefficient

      virtual const char* kind() const;

      explicit sca_sink( sc_core::sc_module_name, double scale_ = 1.0 )
```



```
      : x( "x" ), outp( "outp" ), scale( "scale", scale_ )
    { implementation-defined }
  };

} // namespace sca_tdf

typedef sca_lsf::sca_tdf::sca_sink sca_tdf_sink;

} // namespace sca_lsf
```

4.2.1.18.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_tdf::sca_sink**".

4.2.1.19. sca_lsf::sca_tdf::sca_mux, sca_lsf::sca_tdf_mux

4.2.1.19.1. Description

The class **sca_lsf::sca_tdf::sca_mux** shall implement a primitive module for the LSF MoC that realizes the selection of one of two LSF signals by a TDF control signal (multiplexer). The primitive shall contribute the following equation to the equation system:

$$y(t) = \begin{cases} x_1(t) & ctrl = false \\ x_2(t) & ctrl = true \end{cases}$$

where *ctrl* is the TDF control signal, $x_1(t)$ and $x_2(t)$ are the LSF input signals, and $y(t)$ is the LSF output signal.

4.2.1.19.2. Class definition

```
namespace sca_lsf {

  namespace sca_tdf {

    class sca_mux : public sca_lsf::sca_module
    {
     public:
      sca_lsf::sca_in  x1; // LSF inputs
      sca_lsf::sca_in  x2;

      sca_lsf::sca_out y;  // LSF output

      ::sca_tdf::sca_in<bool> ctrl; // TDF control input

      virtual const char* kind() const;

      explicit sca_mux( sc_core::sc_module_name )
        : x1( "x1" ), x2( "x2" ), y( "y" ), ctrl( "ctrl" )
      { implementation-defined }
    };

  } // namespace sca_tdf

  typedef sca_lsf::sca_tdf::sca_mux sca_tdf_mux;

} // namespace sca_lsf
```

4.2.1.19.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_tdf::sca_mux**".

4.2.1.20. sca_lsf::sca_tdf::sca_demux, sca_lsf::sca_tdf_demux

4.2.1.20.1. Description

The class **sca_lsf::sca_tdf::sca_demux** shall implement a primitive module for the LSF MoC that realizes the routing of an LSF input signal to either one of two LSF output signals controlled by a TDF signal (demultiplexer). The primitive shall contribute the following equations to the equation system:

$$y_1(t) = \begin{cases} x(t) & ctrl = false \\ 0 & ctrl = true \end{cases}$$

$$y_2(t) = \begin{cases} 0 & ctrl = false \\ x(t) & ctrl = true \end{cases}$$

where *ctrl* is the TDF control signal, $x(t)$ is the LSF input signal, and $y_1(t)$ and $y_2(t)$ are the LSF output signals.

4.2.1.20.2. Class definition

```
namespace sca_lsf {

  namespace sca_tdf {

    class sca_demux : public sca_lsf::sca_module
    {
     public:
      sca_lsf::sca_in  x;  // LSF input

      sca_lsf::sca_out y1; // LSF outputs
      sca_lsf::sca_out y2;

      ::sca_tdf::sca_in<bool> ctrl; // TDF control input

      virtual const char* kind() const;

      explicit sca_demux( sc_core::sc_module_name )
        : x( "x" ), y1( "y1" ), y2( "y2" ), ctrl( "ctrl" )
      { implementation-defined }
    };

  } // namespace sca_tdf

  typedef sca_lsf::sca_tdf::sca_demux sca_tdf_demux;

} // namespace sca_lsf
```

4.2.1.20.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_tdf::sca_demux**".

4.2.1.21. sca_lsf::sca_de::sca_gain, sca_lsf::sca_de_gain

4.2.1.21.1. Description

The class **sca_lsf::sca_de::sca_gain** shall implement a primitive module for the LSF MoC that realizes the scaled multiplication of a discrete-event input signal by an LSF input signal. The primitive shall contribute the following equation to the equation system:

$$y(t) = scale \cdot inp \cdot x(t)$$

where *scale* is the constant scale coefficient, *inp* is the discrete-event input signal that shall be interpreted as a discrete-time signal, $x(t)$ is the LSF input signal, and $y(t)$ is the LSF output signal.

4.2.1.21.2. Class definition

```
namespace sca_lsf {

  namespace sca_de {

    class sca_gain : public sca_lsf::sca_module
    {
     public:
      sc_core::sc_in<double> inp; // discrete-event input

      sca_lsf::sca_in  x; // LSF input

      sca_lsf::sca_out y; // LSF output

      sca_core::sca_parameter<double> scale; // scale coefficient
```



```
      virtual const char* kind() const;

      explicit sca_gain( sc_core::sc_module_name, double scale_ = 1.0 )
        : inp( "inp" ), x( "x" ), y( "y" ), scale( "scale", scale_ )
      { implementation-defined }
    };

  } // namespace sca_de

  typedef sca_lsf::sca_de::sca_gain sca_de_gain;

} // namespace sca_lsf
```

4.2.1.21.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_de::sca_gain**".

4.2.1.22. sca_lsf::sca_de::sca_source, sca_lsf::sca_de_source

4.2.1.22.1. Description

The class **sca_lsf::sca_de::sca_source** shall implement a primitive module for the LSF MoC that realizes the scaled conversion of a discrete-event input signal to an LSF signal. The primitive shall contribute the following equation to the equation system:

$$y(t) = scale \cdot inp$$

where *scale* is the constant scale coefficient, *inp* is the discrete-event input signal that shall be interpreted as a discrete-time signal, and $y(t)$ is the LSF output signal.

4.2.1.22.2. Class definition

```
namespace sca_lsf {

  namespace sca_de {

    class sca_source : public sca_lsf::sca_module
    {
     public:
      sc_core::sc_in<double> inp; // discrete-event input

      sca_lsf::sca_out y; // LSF output

      sca_core::sca_parameter<double> scale; // scale coefficient

      virtual const char* kind() const;

      explicit sca_source( sc_core::sc_module_name, double scale_ = 1.0 )
        : inp( "inp" ), y( "y" ), scale( "scale", scale_ )
      { implementation-defined }
    };

  } // namespace sca_de

  typedef sca_lsf::sca_de::sca_source sca_de_source;

} // namespace sca_lsf
```

4.2.1.22.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_de::sca_source**".

4.2.1.23. sca_lsf::sca_de::sca_sink, sca_lsf::sca_de_sink

4.2.1.23.1. Description

The class **sca_lsf::sca_de::sca_sink** shall implement a primitive module for the LSF MoC that realizes a scaled conversion from an LSF signal to a discrete-event signal. The value of the LSF input signal $x(t)$ shall be scaled with coefficient *scale* and written to the discrete-event output port *outp*.

4.2.1.23.2. Class definition

```
namespace sca_lsf {

  namespace sca_de {

    class sca_sink : public sca_lsf::sca_module
    {
     public:
      sca_lsf::sca_in x; // LSF input

      sc_core::sc_out<double> outp; // discrete-event output

      sca_core::sca_parameter<double> scale; // scale coefficient

      virtual const char* kind() const;

      explicit sca_sink( sc_core::sc_module_name, double scale_ = 1.0 )
        : x( "x" ), outp( "outp" ), scale( "scale", scale_ )
      { implementation-defined }
    };

  } // namespace sca_de

  typedef sca_lsf::sca_de::sca_sink sca_de_sink;

} // namespace sca_lsf
```

4.2.1.23.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_de::sca_sink**".

4.2.1.24. sca_lsf::sca_de::sca_mux, sca_lsf::sca_de_mux

4.2.1.24.1. Description

The class **sca_lsf::sca_de::sca_mux** shall implement a primitive module for the LSF MoC that realizes the selection of one of two LSF signals by a discrete-event control signal (multiplexer). The primitive shall contribute the following equation to the equation system:

$$y(t) = \begin{cases} x_1(t) & ctrl = false \\ x_2(t) & ctrl = true \end{cases}$$

where *ctrl* is the discrete-event control signal, $x_1(t)$ and $x_2(t)$ are the LSF input signals, and $y(t)$ is the LSF output signal.

4.2.1.24.2. Class definition

```
namespace sca_lsf {

  namespace sca_de {

    class sca_mux : public sca_lsf::sca_module
    {
     public:
      sca_lsf::sca_in  x1; // LSF inputs
      sca_lsf::sca_in  x2;

      sca_lsf::sca_out y;  // LSF output

      sc_core::sc_in<bool> ctrl; // discrete-event control

      virtual const char* kind() const;

      explicit sca_mux( sc_core::sc_module_name )
        : x1( "x1" ), x2( "x2" ), y( "y" ), ctrl( "ctrl" )
      { implementation-defined }
    };

  } // namespace sca_de

  typedef sca_lsf::sca_de::sca_mux sca_de_mux;
```



```
} // namespace sca_lsf
```

4.2.1.24.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_de::sca_mux**".

4.2.1.25. sca_lsf::sca_de::sca_demux, sca_lsf::sca_de_demux

4.2.1.25.1. Description

The class **sca_lsf::sca_de::sca_demux** shall implement a primitive module for the LSF MoC that realizes the routing of an LSF input signal to either one of two LSF output signals controlled by a discrete-event signal (demultiplexer). The primitive shall contribute the following equations to the equation system:

$$y_1(t) = \begin{cases} x(t) & ctrl = false \\ 0 & ctrl = true \end{cases}$$

$$y_2(t) = \begin{cases} 0 & ctrl = false \\ x(t) & ctrl = true \end{cases}$$

where *ctrl* is the discrete-event control signal, $x(t)$ is the LSF input signal, and $y_1(t)$ and $y_2(t)$ are the LSF output signals.

4.2.1.25.2. Class definition

```
namespace sca_lsf {

  namespace sca_de {

    class sca_demux : public sca_lsf::sca_module
    {
     public:
      sca_lsf::sca_in  x;  // LSF input

      sca_lsf::sca_out y1; // LSF outputs
      sca_lsf::sca_out y2;

      sc_core::sc_in<bool> ctrl; // discrete-event control

      virtual const char* kind() const;

      explicit sca_demux( sc_core::sc_module_name )
        : x( "x" ), y1( "y1" ), y2( "y2" ), ctrl( "ctrl" )
      { implementation-defined }
    };

  } // namespace sca_de

  typedef sca_lsf::sca_de::sca_demux sca_de_demux;

} // namespace sca_lsf
```

4.2.1.25.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_lsf::sca_de::sca_demux**".

4.2.2. Hierarchical LSF composition and port binding

The hierarchical composition of LSF modules shall use modules derived from class **sc_core::sc_module** and the constructor or its equivalent macro definitions. A hierarchical module can include modules and ports of different models of computation. Port binding rules shall follow IEEE Std 1666-2005 as well as the following specific rules:

- A port of class **sca_lsf::sca_in** shall only be bound to a primitive channel of class **sca_lsf::sca_signal** or to a port of class **sca_lsf::sca_in** or **sca_lsf::sca_out** of the parent module.

- A port of class **sca_lsf::sca_out** shall only be bound to a primitive channel of class **sca_lsf::sca_signal** or to port of class **sca_lsf::sca_out** of the parent module.
- A port of class **sca_lsf::sca_in** or **sca_lsf::sca_out** shall be bound to exactly one primitive channel of class **sca_lsf::sca:signal** throughout the whole hierarchy.
- A primitive channel of class **sca_lsf::sca_signal** shall have exactly one primitive port of class **sca_lsf::sca_out** bound to it and may have one or more primitive ports of class **sca_lsf::sca_in** bound to it throughout the whole hierarchy.

Predefined LSF primitive modules using ports of other models of computation shall follow the port binding rules of the corresponding models of computation.

4.2.3. LSF MoC elaboration and simulation

An implementation of the LSF MoC in a SystemC AMS class library shall include a public shell consisting of the predefined classes, functions, and so forth that can be used directly by an application. An implementation shall also include an LSF solver that implements the functionality of the LSF class library. The underlying semantics of the LSF solver are defined in this subclause.

The execution of a SystemC AMS application that includes LSF modules consists of elaboration followed by simulation. Elaboration results in one or more equation systems based on the contributions of the connected LSF modules. Simulation solves the equation systems repetitively. In addition to providing support for elaboration and simulation, the LSF solver may also provide implementation-specific functionality beyond the scope of this standard. As an example of such functionality, the LSF solver may report information on the LSF module composition and equation setup.

4.2.3.1. LSF elaboration

The primary purpose of LSF elaboration is to create internal data structures and equations for the LSF solver to support the semantics of LSF simulation. The LSF elaboration as described in this clause and in the following subclauses shall execute in a **sc_core::sc_module::end_of_elaboration** callback.

The actions stated in the following subclauses shall occur, in the given order, during LSF elaboration and only during LSF elaboration. The description of such actions use the concept of an LSF cluster, which is a set of LSF modules connected by channels of class **sca_lsf::sca_signal**.

LSF elaboration shall lock the parameter values of the predefined primitive modules. (See 3.2.6).

4.2.3.1.1. Timestep calculation and propagation

The timestep for every LSF cluster shall be derived from the timestep of a connected TDF cluster or set by the member function **set_timestep** of an LSF primitive module derived from class **sca_lsf::sca_module** in the corresponding LSF cluster. The timestep shall be propagated within the LSF cluster to all primitive modules and to all ports of class **sca_tdf::sca_in** and **sca_tdf::sca_out**, if any.

It shall be an error if a timestep value is not assigned to at least one LSF module or if inconsistent timesteps are defined in an LSF cluster.

After successful LSF elaboration, all assigned timestep values shall be overridden by the propagated timestep values.

NOTE—An LSF cluster could be considered as one TDF module, which could be connected to TDF modules in a hierarchical composition by the ports of class **sca_tdf::sca_in** and **sca_tdf::sca_out** of the predefined LSF primitive modules. In this case, the LSF cluster is included in the timestep calculation of the TDF cluster and must comply to the same rules (see 4.1.3.1.2).

4.2.3.1.2. LSF equation system setup and solvability check

For each LSF cluster, an equation system shall be setup by combining:

1. the contributing equations of each of the predefined LSF primitive modules in the cluster.

2. the equations implied by the connected ports of class **sca_lsf::sca_in** and **sca_lsf::sca_out** that express the equality of the values conveyed by the ports.

It shall be an error if any of the equation systems is numerically singular.

4.2.3.2. LSF simulation

This subclause defines the process of time-domain simulation of LSF descriptions. The simulation of a cluster of LSF modules is done by a repetitive solving of the underlying equation systems.

4.2.3.2.1. LSF initialization

For each LSF cluster:

1. all LSF signals shall be set to zero.
2. for all LSF signals consistent initial conditions shall be calculated in agreement with the initial conditions set by the predefined primitives.

4.2.3.2.2. Time-domain simulation

The solver shall at least provide results at the calculated timestep distances.

4.2.3.2.3. Synchronization with TDF MoC

Synchronization with the TDF MoC shall be done exclusively by using the predefined LSF primitive modules containing ports of class **sca_tdf::sca_in** and **sca_tdf::sca_out**.

The LSF solver reads repetitively samples from ports of class **sca_tdf::sca_in** for all calculated timesteps of the LSF cluster. Consecutive reads shall be interpreted as forming a continuous-time signal.

The LSF solver writes repetitively samples to ports of class **sca_tdf::sca_out** for all calculated timesteps of the LSF cluster.

4.2.3.2.4. Synchronization with SystemC kernel

Synchronization with the SystemC kernel shall be done exclusively by using the predefined LSF primitive modules containing ports of class **sc_core::sc_in** and **sc_core::sc_out**.

The LSF solver reads repetitively values from ports of class **sc_core::sc_in** at each first delta cycle of the corresponding SystemC time for all calculated timesteps of the LSF cluster. The value is assumed as constant until the next value is read.

The LSF solver writes repetitively values to ports of class **sc_core::sc_out** at each first delta cycle of the corresponding SystemC time for all calculated timesteps of the LSF cluster.

4.2.3.3. Running elaboration and simulation

The implementation shall use the same elaboration and simulation semantics as defined in IEEE Std 1666-2005 (SystemC Language Reference Manual).

NOTE—LSF modules can be instantiated in the **sc_main** context and the elaboration and simulation can be controlled by the function **sc_core::sc_start**.

4.3. Electrical linear networks model of computation

The ELN model of computation shall define the behavior of conservative continuous-time systems consisting of linear networks based on electrical primitives. The resulting differential and algebraic equation system, which is determined by the set of connected predefined ELN primitive modules, shall be solved during simulation. The mathematical relation defined in each ELN primitive module shall contribute to this overall equation system. The predefined ELN primitive modules shall serve as a basic set of electrical linear network primitives as defined in this subclause.

For ELN primitive modules with exactly two terminals, the voltage across the primitive is defined in volt and the current through the primitive is defined in ampere

Current tracing for ELN primitive modules shall be supported for the primitives having at least two terminals as defined in this subclause. The current, which is traced, is defined as the current in ampere flowing through the ELN primitive from terminal *p* to terminal *n*.

Voltage tracing shall be supported by the primitive channels of class **sca_eln::sca_node** and **sca_eln::sca_node_ref**. The voltage, which is traced, is defined as the voltage in volt across the electrical node of class **sca_eln::sca_node** or **sca_eln::sca_node_ref** and the corresponding electrical reference node of class **sca_eln::sca_node_ref**.

An implementation may support current tracing of ELN primitive modules with more than two terminals. These modules shall be derived from class **sca_util::***sca_traceable_object*[†].

4.3.1. ELN class definitions

All names used in the ELN class definitions shall be placed in the namespace **sca_eln**.

4.3.1.1. sca_eln::sca_module

4.3.1.1.1. Description

The class **sca_eln::sca_module** shall define the base class for all ELN primitive modules. An application shall not derive from this class directly, but shall use the predefined primitive modules as defined in the following clauses.

4.3.1.1.2. Class definition

```
namespace sca_eln {

  class sca_module : public sca_core::sca_module
  {
   public:
    virtual const char* kind() const;

   protected:
    sca_module();
    virtual ~sca_module();
  };
} // namespace sca_eln
```

4.3.1.2. sca_eln::sca_node_if

4.3.1.2.1. Description

The class **sca_eln::sca_node_if** shall define an interface proper for the primitive channels of class **sca_eln::sca_node** and **sca_eln::sca_node_ref**. The interface class member functions are implementation-defined.

4.3.1.2.2. Class definition

```
namespace sca_eln {

  class sca_node_if : public sca_core::sca_interface
  {
   protected:
    sca_node_if();

   private:
    // Other members
    implementation-defined

    // Disabled
    sca_node_if( const sca_eln::sca_node_if& );
    sca_eln::sca_node_if& operator= ( const sca_eln::sca_node_if& );
  };

} // namespace sca_eln
```


4.3.1.3. sca_eln::sca_terminal

4.3.1.3.1. Description

The class **sca_eln::sca_terminal** shall define a port class for the ELN MoC.

4.3.1.3.2. Class definition

```
namespace sca_eln {

  class sca_terminal : public sca_core::sca_port< sca_eln::sca_node_if >
  {
   public:
    sca_terminal();
    explicit sca_terminal( const char* name_ );

    virtual const char* kind() const;

   private:
    // Other members
    implementation-defined

    // Disabled
    sca_terminal( const sca_eln::sca_terminal& );
  };

} // namespace sca_eln
```

4.3.1.3.3. Constructors

```
sca_terminal();

explicit sca_terminal( const char* name_ );
```

The constructor for class **sca_eln::sca_terminal** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sca_core::sca_port** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_terminal**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sca_core::sca_port**.

4.3.1.3.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string “**sca_eln::sca_terminal**”.

4.3.1.4. sca_eln::sca_node

4.3.1.4.1. Description

The class **sca_eln::sca_node** shall define a primitive channel for the ELN MoC. It shall be used for connecting ELN primitive modules using ports of class **sca_eln::sca_terminal**. The primitive channel shall represent an electrical node. An application shall not access the associated interface directly.

4.3.1.4.2. Class definition

```
namespace sca_eln {

  class sca_node : public sca_eln::sca_node_if,
                   public sca_core::sca_prim_channel
  {
   public:
    sca_node();
    explicit sca_node( const char* name_ );

    virtual const char* kind() const;

   private:
    // Disabled
    sca_node( const sca_eln::sca_node& );
```



```
  };

} // namespace sca_eln
```

4.3.1.4.3. Constructors

```
sca_node();

explicit sca_node( const char* name_ );
```

The constructor for class **sca_eln::sca_node** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sca_core::sca_prim_channel** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_node**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sca_core::sca_prim_channel**.

4.3.1.4.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string “**sca_eln::sca_node**”.

4.3.1.5. sca_eln::sca_node_ref

4.3.1.5.1. Description

The class **sca_eln::sca_node_ref** shall define a primitive channel for the ELN MoC. It shall be used for connecting ELN primitive modules using ports of class **sca_eln::sca_terminal**. The primitive channel shall represent an electrical reference node, a node which shall always hold a voltage of zero volt. An application shall not access the associated interface directly.

4.3.1.5.2. Class definition

```
namespace sca_eln {

  class sca_node_ref : public sca_eln::sca_node_if,
                       public sca_core::sca_prim_channel
  {
   public:
    sca_node_ref();
    explicit sca_node_ref( const char* name_ );

    virtual const char* kind() const;

   private:
    // Disabled
    sca_node_ref( const sca_eln::sca_node_ref& );
  };

} // namespace sca_eln
```

4.3.1.5.3. Constructors

```
sca_node_ref();

explicit sca_node_ref( const char* name_ );
```

The constructor for class **sca_eln::sca_node_ref** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sca_core::sca_prim_channel** to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_core::sc_gen_unique_name**(“**sca_node_ref**”) to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sca_core::sca_prim_channel**.

4.3.1.5.4. kind

```
virtual const char* kind() const;
```


The member function **kind** shall return the string "**sca_eln::sca_node_ref**".

4.3.1.6. sca_eln::sca_r

4.3.1.6.1. Description

The class **sca_eln::sca_r** shall implement a primitive module for the ELN MoC that represents a resistor. The primitive shall contribute the following equation to the equation system:

$$v_{p,n}(t) = i_{p,n}(t) \cdot value$$

where *value* is the resistance in ohm, $v_{p,n}(t)$ is the voltage across the resistor between terminals *p* and *n*, and $i_{p,n}(t)$ is the current through the resistor flowing from terminal *p* to terminal *n*.

4.3.1.6.2. Class definition

```
namespace sca_eln {

  class sca_r : public sca_eln::sca_module,
                public sca_util::sca_traceable_object†
  {
   public:
    sca_eln::sca_terminal p;
    sca_eln::sca_terminal n;

    sca_core::sca_parameter<double> value;

    virtual const char* kind() const;

    explicit sca_r( sc_core::sc_module_name, double value_ = 1.0 )
      : p( "p" ), n( "n" ), value( "value", value_ )
    { implementation-defined }
  };

} // namespace sca_eln
```

4.3.1.6.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_r**".

4.3.1.7. sca_eln::sca_c

4.3.1.7.1. Description

The class **sca_eln::sca_c** shall implement a primitive module for the ELN MoC that represents a capacitor. The primitive shall contribute the following equation to the equation system:

$$i_{p,n}(t) = \frac{d\left(value \cdot v_{p,n}(t) + q_0\right)}{dt}$$

where *value* is the capacitance in farad, q_0 is the initial charge in coulomb, $v_{p,n}(t)$ is the voltage across the capacitor between terminals *p* and *n*, and $i_{p,n}(t)$ is the current through the capacitor flowing from terminal *p* to terminal *n*.

4.3.1.7.2. Class definition

```
namespace sca_eln {

  class sca_c : public sca_eln::sca_module,
                public sca_util::sca_traceable_object†
  {
   public:
    sca_eln::sca_terminal p;
    sca_eln::sca_terminal n;

    sca_core::sca_parameter<double> value;
    sca_core::sca_parameter<double> q0;

    virtual const char* kind() const;
```

```
    explicit sca_c( sc_core::sc_module_name, double value_ = 1.0, double q0_ = 0.0 )
      : p( "p" ), n( "n" ), value( "value", value_ ), q0( "q0", q0_ )
    { implementation-defined }
  };

} // namespace sca_eln
```

4.3.1.7.3. Constraint of usage

The argument *value* shall not be numerically zero.

4.3.1.7.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_c**".

4.3.1.8. sca_eln::sca_l

4.3.1.8.1. Description

The class **sca_eln::sca_l** shall implement a primitive module for the ELN MoC that represents an inductor. The primitive shall contribute the following equation to the equation system:

$$v_{p,n}(t) = \frac{d\left(value \cdot i_{p,n}(t) + phi_0\right)}{dt}$$

where *value* is the inductance in henry, phi_0 is the initial magnetic flux in weber, $v_{p,n}(t)$ is the voltage across the inductor between terminals *p* and *n*, and $i_{p,n}(t)$ is the current through the inductor flowing from terminal *p* to terminal *n*.

4.3.1.8.2. Class definition

```
namespace sca_eln {

  class sca_l : public sca_eln::sca_module,
                public sca_util::sca_traceable_object†
  {
   public:
    sca_eln::sca_terminal p;
    sca_eln::sca_terminal n;

    sca_core::sca_parameter<double> value;
    sca_core::sca_parameter<double> phi0;

    virtual const char* kind() const;

    explicit sca_l( sc_core::sc_module_name, double value_ = 1.0, double phi0_ = 0.0 )
      : p( "p" ), n( "n" ), value( "value", value_ ), phi0( "phi0", phi0_ )
    { implementation-defined }
  };

} // namespace sca_eln
```

4.3.1.8.3. Constraint of usage

The argument *value* shall not be numerically zero.

4.3.1.8.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_l**".

4.3.1.9. sca_eln::sca_vcvs

4.3.1.9.1. Description

The class **sca_eln::sca_vcvs** shall implement a primitive module for the ELN MoC that represents a voltage controlled voltage source. The primitive shall contribute the following equation to the equation system:

$$v_{np,nn}(t) = value \cdot v_{ncp,ncn}(t)$$

where *value* is the scale coefficient, $v_{ncp,ncn}(t)$ is the control voltage across terminals *ncp* and *ncn*, and $v_{np,nn}(t)$ is the voltage across terminals *np* and *nn*.

4.3.1.9.2. Class definition

```
namespace sca_eln {

  class sca_vcvs : public sca_eln::sca_module
  {
   public:
    sca_eln::sca_terminal ncp;
    sca_eln::sca_terminal ncn;

    sca_eln::sca_terminal np;
    sca_eln::sca_terminal nn;

    sca_core::sca_parameter<double> value;

    virtual const char* kind() const;

    explicit sca_vcvs( sc_core::sc_module_name, double value_ = 1.0 )
      : ncp( "ncp" ), ncn( "ncn" ), np( "np" ), nn( "nn" ), value( "value", value_ )
    { implementation-defined }
  };

} // namespace sca_eln
```

4.3.1.9.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_vcvs**".

4.3.1.10. sca_eln::sca_vccs

4.3.1.10.1. Description

The class **sca_eln::sca_vccs** shall implement a primitive module for the ELN MoC that represents a voltage controlled current source. The primitive shall contribute the following equation to the equation system:

$$i_{np,nn}(t) = value \cdot v_{ncp,ncn}(t)$$

where *value* is the scale coefficient in siemens, $v_{ncp,ncn}(t)$ is the control voltage across terminals *ncp* and *ncn*, and $i_{np,nn}(t)$ is the current flowing through the primitive from terminal *np* to terminal *nn*.

4.3.1.10.2. Class definition

```
namespace sca_eln {

  class sca_vccs : public sca_eln::sca_module
  {
   public:
    sca_eln::sca_terminal ncp;
    sca_eln::sca_terminal ncn;

    sca_eln::sca_terminal np;
    sca_eln::sca_terminal nn;

    sca_core::sca_parameter<double> value;

    virtual const char* kind() const;

    explicit sca_vccs( sc_core::sc_module_name, double value_ = 1.0 )
      : ncp( "ncp" ), ncn( "ncn" ), np( "np" ), nn( "nn" ), value( "value", value_ )
    { implementation-defined }
  };

} // namespace sca_eln
```

4.3.1.10.3. kind

```
virtual const char* kind() const;
```


The member function **kind** shall return the string "**sca_eln::sca_vccs**".

4.3.1.11. sca_eln::sca_ccvs

4.3.1.11.1. Description

The class **sca_eln::sca_ccvs** shall implement a primitive module for the ELN MoC that represents a current controlled voltage source. The primitive shall contribute the following equations to the equation system:

$$v_{np,nn}(t) = value \cdot i_{ncp,ncn}(t)$$

$$v_{ncp,ncn}(t) = 0$$

where *value* is the scale coefficient in ohm, $i_{ncp,ncn}(t)$ is the current flowing through the primitive from terminal *ncp* to terminal *ncn*, $v_{np,nn}(t)$ is the voltage across terminals *np* and *nn*, and $v_{ncp,ncn}(t)$ is the voltage across terminals *ncp* and *ncn*.

4.3.1.11.2. Class definition

```
namespace sca_eln {

  class sca_ccvs : public sca_eln::sca_module
  {
   public:
    sca_eln::sca_terminal ncp;
    sca_eln::sca_terminal ncn;

    sca_eln::sca_terminal np;
    sca_eln::sca_terminal nn;

    sca_core::sca_parameter<double> value;

    virtual const char* kind() const;

    explicit sca_ccvs( sc_core::sc_module_name, double value_ = 1.0 )
      : ncp( "ncp" ), ncn( "ncn" ), np( "np" ), nn( "nn" ), value( "value", value_ )
    { implementation-defined }
  };

} // namespace sca_eln
```

4.3.1.11.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_ccvs**".

4.3.1.12. sca_eln::sca_cccs

4.3.1.12.1. Description

The class **sca_eln::sca_cccs** shall implement a primitive module for the ELN MoC that represents a current controlled current source. The primitive shall contribute the following equations to the equation system:

$$i_{np,nn}(t) = value \cdot i_{ncp,ncn}(t)$$

$$v_{ncp,ncn}(t) = 0$$

where *value* is the scale coefficient, $i_{ncp,ncn}(t)$ is the current flowing through the primitive from terminal *ncp* to terminal *ncn*, $i_{np,nn}(t)$ is the current flowing through the primitive from terminal *np* to terminal *nn*, and $v_{ncp,ncn}(t)$ is the voltage across terminals *ncp* and *ncn*.

4.3.1.12.2. Class definition

```
namespace sca_eln {

  class sca_cccs : public sca_eln::sca_module
  {
   public:
    sca_eln::sca_terminal ncp;
```



```
    sca_eln::sca_terminal ncn;

    sca_eln::sca_terminal np;
    sca_eln::sca_terminal nn;

    sca_core::sca_parameter<double> value;

    virtual const char* kind() const;

    explicit sca_cccs( sc_core::sc_module_name, double value_ = 1.0 )
      : ncp( "ncp" ), ncn( "ncn" ), np( "np" ), nn( "nn" ), value( "value", value_ )
    { implementation-defined }
  };

} // namespace sca_eln
```

4.3.1.12.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_cccs**".

4.3.1.13. sca_eln::sca_nullor

4.3.1.13.1. Description

The class **sca_eln::sca_nullor** shall implement a primitive module for the ELN MoC that represents a nullor. The primitive shall contribute the following equations to the equation system:

$$v_{nip,nin}(t) = 0$$

$$i_{nip,nin}(t) = 0$$

where $v_{nip,nin}(t)$ is the voltage across terminals *nip* and *nin*, and $i_{nip,nin}(t)$ is the current flowing through the primitive from terminal *nip* to terminal *nin*.

NOTE—A nullor (a nullator - norator pair) corresponds to an ideal operational amplifier (an amplifier with an infinite gain).

4.3.1.13.2. Class definition

```
namespace sca_eln {

  class sca_nullor : public sca_eln::sca_module
  {
   public:
    sca_eln::sca_terminal nip;
    sca_eln::sca_terminal nin;

    sca_eln::sca_terminal nop;
    sca_eln::sca_terminal non;

    virtual const char* kind() const;

    explicit sca_nullor( sc_core::sc_module_name )
      : nip( "nip" ), nin( "nin" ), nop( "nop" ), non( "non" )
    { implementation-defined }
  };

} // namespace sca_eln
```

4.3.1.13.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_nullor**".

4.3.1.14. sca_eln::sca_gyrator

4.3.1.14.1. Description

The class **sca_eln::sca_gyrator** shall implement a primitive module for the ELN MoC that represents a gyrator. The primitive shall contribute the following equations to the equation system:

$$i_{p_1,n_1}(t) = g_2 \cdot v_{p_2,n_2}(t)$$

$$i_{p_2,n_2}(t) = -g_1 \cdot v_{p_1,n_1}(t)$$

where g_1 and g_2 are the gyration conductances in siemens (ampere / volt), $v_{p_2,n_2}(t)$ is the voltage across terminals p_2 and n_2, $v_{p_1,n_1}(t)$ is the voltage across terminals p_1 and n_1, $i_{p_1,n_1}(t)$ is the current flowing through the primitive from terminal p_1 to terminal n_1, and $i_{p_2,n_2}(t)$ is the current flowing through the primitive from terminal p_2 to terminal n_2.

4.3.1.14.2. Class definition

```
namespace sca_eln {

  class sca_gyrator : public sca_eln::sca_module
  {
   public:
    sca_eln::sca_terminal p1;
    sca_eln::sca_terminal n1;

    sca_eln::sca_terminal p2;
    sca_eln::sca_terminal n2;

    sca_core::sca_parameter<double> g1;
    sca_core::sca_parameter<double> g2;

    virtual const char* kind() const;

    explicit sca_gyrator( sc_core::sc_module_name, double g1_ = 1.0, double g2_ = 1.0 )
      : p1( "p1" ), n1( "n1" ), p2( "p2" ), n2( "n2" ), g1( "g1", g1_ ), g2( "g2", g2_ )
    { implementation-defined }
  };

} // namespace sca_eln
```

4.3.1.14.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_gyrator**".

4.3.1.15. sca_eln::sca_ideal_transformer

4.3.1.15.1. Description

The class **sca_eln::sca_ideal_transformer** shall implement a primitive module for the ELN MoC that represents an ideal transformer. The primitive shall contribute the following equations to the equation system:

$$v_{p1,n1}(t) = ratio \cdot v_{p2,n2}(t)$$

$$i_{p2,n2}(t) = ratio \cdot i_{p1,n1}(t)$$

where *ratio* is the transformation ratio, $v_{p_2,n_2}(t)$ is the voltage across terminals p_2 and n_2, $v_{p_1,n_1}(t)$ is the voltage across terminals p_1 and n_1, $i_{p_1,n_1}(t)$ is the current flowing through the primitive from terminal p_1 to terminal n_1, and $i_{p_2,n_2}(t)$ is the current flowing through the primitive from terminal p_2 to terminal n_2.

4.3.1.15.2. Class definition

```
namespace sca_eln {

  class sca_ideal_transformer : public sca_eln::sca_module
  {
   public:
    sca_eln::sca_terminal p1;
    sca_eln::sca_terminal n1;

    sca_eln::sca_terminal p2;
    sca_eln::sca_terminal n2;

    sca_core::sca_parameter<double> ratio;

    virtual const char* kind() const;
```



```
    explicit sca_ideal_transformer( sc_core::sc_module_name, double ratio_ = 1.0 )
      : p1( "p1" ), n1( "n1" ), p2( "p2" ), n2( "n2" ), ratio( "ratio", ratio_ )
    { implementation-defined }
  };

} // namespace sca_eln
```

4.3.1.15.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_ideal_transformer**".

4.3.1.16. sca_eln::sca_transmission_line

4.3.1.16.1. Description

The class **sca_eln::sca_transmission_line** shall implement a primitive module for the ELN MoC that represents a transmission line. The primitive shall contribute the following equations to the equation system:

$$v_{a_1,b_1}(t) = \begin{cases} z_0 \cdot i_{a_1,b_1}(t) & t < delay \\ e^{-delta_0 \cdot delay}\left(v_{a_2,b_2}(t-delay) + z_0 \cdot i_{a_2,b_2}(t-delay)\right) + z_0 \cdot i_{a_1,b_1}(t) & t \geq delay \end{cases}$$

$$v_{a_2,b_2}(t) = \begin{cases} z_0 \cdot i_{a_2,b_2}(t) & t < delay \\ e^{-delta_0 \cdot delay}\left(v_{a_1,b_1}(t-delay) + z_0 \cdot i_{a_1,b_1}(t-delay)\right) + z_0 \cdot i_{a_2,b_2}(t) & t \geq delay \end{cases}$$

where z_0 is the characteristic impedance of the transmission line in ohm, *delay* is the transmission delay in seconds and $delta_0$ is the dissipation factor in 1/seconds. $v_{a_1,b_1}(t)$ is the voltage across terminals a_1 and b_1, $v_{a_2,b_2}(t)$ is the voltage across terminals a_2 and b_2, $i_{a_1,b_1}(t)$ is the current flowing through the primitive from terminal a_1 to terminal b_1, and $i_{a_2,b_2}(t)$ is the current flowing through the primitive from terminal a_2 to terminal b_2.

4.3.1.16.2. Class definition

```
namespace sca_eln {

  class sca_transmission_line : public sca_eln::sca_module
  {
   public:
    sca_eln::sca_terminal a1;
    sca_eln::sca_terminal b1;

    sca_eln::sca_terminal a2;
    sca_eln::sca_terminal b2;

    sca_core::sca_parameter<double> z0;
    sca_core::sca_parameter<sca_util::sca_time> delay;
    sca_core::sca_parameter<double> delta0;

    virtual const char* kind() const;

    explicit sca_transmission_line( sc_core::sc_module_name,
                                    double z0_     = 100.0,
                                    sca_util::sca_time delay_ = sc_core::SC_ZERO_TIME,
                                    double delta0_ = 0.0 )
      : a1( "a1" ), b1( "b1" ),
        a2( "a2" ), b2( "b2" ),
        z0( "z0", z0_ ),
        delay( "delay", delay_ ),
        delta0( "delta0", delta0_ )
     { implementation-defined }
   };

} // namespace sca_eln
```

4.3.1.16.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_transmission_line**".

4.3.1.17. sca_eln::sca_vsource

4.3.1.17.1. Description

The class **sca_eln::sca_vsource** shall implement a primitive module for the ELN MoC that realizes a voltage source. In time-domain simulation, the primitive shall contribute the following equation to the equation system:

$$v_{p,n}(t) = \begin{cases} init\_value & t < delay \\ offset + amplitude \cdot \sin(2\pi \cdot frequency \cdot (t - delay) + phase) & t \geq delay \end{cases}$$

where *t* is the time, *delay* is the initial delay in seconds, *init_value* is the inital voltage in volt, *offset* is the offset voltage in volt, *amplitude* is the source amplitude in volt, *frequency* is the source frequency in hertz, *phase* is the source phase in radians, π is the pi constant, and $v_{p,n}(t)$ is the output voltage across terminals *p* and *n*. Voltage source parameters shall be set to zero by default.

In small-signal frequency-domain simulation, the primitive shall contribute the following equation to the equation system:

$$v_{p,n}(f) = ac\_amplitude \cdot \{\cos(ac\_phase) + j \cdot \sin(ac\_phase)\}$$

where *f* is the simulation frequency in hertz, *ac_amplitude* is the small-signal amplitude in volt, and *ac_phase* is the small-signal phase in radian.

In small-signal frequency-domain noise simulation, the primitive shall contribute the following equation to the equation system:

$$v_{p,n}(f) = ac\_noise\_amplitude$$

where *f* is the simulation frequency in hertz, and *ac_noise_amplitude* is the small-signal noise amplitude in volt.

4.3.1.17.2. Class definition

```
namespace sca_eln {

  class sca_vsource : public sca_eln::sca_module,
                      public sca_util::sca_traceable_object†
  {
   public:
    sca_eln::sca_terminal p;
    sca_eln::sca_terminal n;

    sca_core::sca_parameter<double> init_value;
    sca_core::sca_parameter<double> offset;
    sca_core::sca_parameter<double> amplitude;
    sca_core::sca_parameter<double> frequency;
    sca_core::sca_parameter<double> phase;
    sca_core::sca_parameter<sca_core::sca_time> delay;
    sca_core::sca_parameter<double> ac_amplitude;
    sca_core::sca_parameter<double> ac_phase;
    sca_core::sca_parameter<double> ac_noise_amplitude;

    virtual const char* kind() const;

    explicit sca_vsource( sc_core::sc_module_name,
                          double init_value_   = 0.0,
                          double offset_       = 0.0,
                          double amplitude_    = 0.0,
                          double frequency_    = 0.0,
                          double phase_        = 0.0,
                          sca_core::sca_time delay_ = sc_core::SC_ZERO_TIME,
                          double ac_amplitude_ = 0.0,
                          double ac_phase_     = 0.0,
                          double ac_noise_amplitude_ = 0.0 )
      : p( "p" ),
        n( "n" ),
        init_value( "init_value", init_value_ ),
        offset( "offset", offset_ ),
        amplitude( "amplitude", amplitude_ ),
        frequency( "frequency", frequency_ ),
```



```
        phase( "phase", phase_ ),
        delay( "delay", delay_ ),
        ac_amplitude( "ac_amplitude", ac_amplitude_ ),
        ac_phase( "ac_phase", ac_phase_ ),
        ac_noise_amplitude( "ac_noise_amplitude", ac_noise_amplitude_ )
    { implementation-defined }

  };

} // namespace sca_eln
```

4.3.1.17.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_vsource**".

4.3.1.18. sca_eln::sca_isource

4.3.1.18.1. Description

The class **sca_eln::sca_isource** shall implement a primitive module for the ELN MoC that realizes a current source. In time-domain simulation, the primitive shall contribute the following equation to the equation system:

$$i_{p,n}(t) = \begin{cases} init\_value & t < delay \\ offset + amplitude \cdot \sin(2\pi \cdot frequency \cdot (t - delay) + phase) & t \geq delay \end{cases}$$

where *t* is the time, *delay* is the initial delay in seconds, *init_value* is the initial current in ampere, *offset* is the offset current in ampere, *amplitude* is the source amplitude in ampere, *frequency* is the source frequency in hertz, *phase* is the source phase in radians, π is the pi constant, and $i_{p,n}(t)$ is the output current through the primitive from terminal *p* to terminal *n*. Current source parameters shall be set to zero by default.

In small-signal frequency-domain simulation, the primitive shall contribute the following equation to the equation system:

$$i_{p,n}(f) = ac\_amplitude \cdot \{\cos(ac\_phase) + j \cdot \sin(ac\_phase)\}$$

where *f* is the simulation frequency, *ac_amplitude* is the small-signal amplitude in ampere, and *ac_phase* is the small-signal phase in radian.

In small-signal frequency-domain noise simulation, the primitive shall contribute the following equation to the equation system:

$$i_{p,n}(f) = ac\_noise\_amplitude$$

where *f* is the simulation frequency, and *ac_noise_amplitude* is the small-signal noise amplitude in ampere.

4.3.1.18.2. Class definition

```
namespace sca_eln {

  class sca_isource : public sca_eln::sca_module,
                      public sca_util::sca_traceable_object†
  {
   public:
    sca_eln::sca_terminal p;
    sca_eln::sca_terminal n;

    sca_core::sca_parameter<double> init_value;
    sca_core::sca_parameter<double> offset;
    sca_core::sca_parameter<double> amplitude;
    sca_core::sca_parameter<double> frequency;
    sca_core::sca_parameter<double> phase;
    sca_core::sca_parameter<sca_core::sca_time> delay;
    sca_core::sca_parameter<double> ac_amplitude;
    sca_core::sca_parameter<double> ac_phase;
    sca_core::sca_parameter<double> ac_noise_amplitude;

    virtual const char* kind() const;
```



```
    explicit sca_isource( sc_core::sc_module_name,
                          double init_value_   = 0.0,
                          double offset_       = 0.0,
                          double amplitude_    = 0.0,
                          double frequency_    = 0.0,
                          double phase_        = 0.0,
                          sca_core::sca_time delay_ = sc_core::SC_ZERO_TIME,
                          double ac_amplitude_ = 0.0,
                          double ac_phase_     = 0.0,
                          double ac_noise_amplitude_ = 0.0 )
      : p( "p" ),
        n( "n" ),
        init_value( "init_value", init_value_ ),
        offset( "offset", offset_ ),
        amplitude( "amplitude", amplitude_ ),
        frequency( "frequency", frequency_ ),
        phase( "phase", phase_ ),
        delay( "delay", delay_ ),
        ac_amplitude( "ac_amplitude", ac_amplitude_ ),
        ac_phase( "ac_phase", ac_phase_ ),
        ac_noise_amplitude( "ac_noise_amplitude", ac_noise_amplitude_ )
    { implementation-defined }

  };

} // namespace sca_eln
```

4.3.1.18.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_isource**".

4.3.1.19. sca_eln::sca_tdf::sca_r, sca_eln::sca_tdf_r

4.3.1.19.1. Description

The class **sca_eln::sca_tdf::sca_r** shall implement a primitive module for the ELN MoC that represents a resistor, whose resistance is controlled by a TDF input signal. The primitive shall contribute the following equation to the equation system:

$$v_{p,n}(t) = scale \cdot inp \cdot i_{p,n}(t)$$

where *scale* is the constant scale coefficient, *inp* is the TDF input signal, $v_{p,n}(t)$ is the voltage across terminals *p* and *n*, and $i_{p,n}(t)$ is the current flowing through the primitive from terminal *p* to terminal *n*. The product of *scale* and *inp* shall be interpreted as the resistance in ohm.

4.3.1.19.2. Class definition

```
namespace sca_eln {

  namespace sca_tdf {

    class sca_r : public sca_eln::sca_module,
                  public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      ::sca_tdf::sca_in<double> inp;

      sca_core::sca_parameter<double> scale;

      virtual const char* kind() const;

      explicit sca_r( sc_core::sc_module_name, double scale_ = 1.0 )
        : p( "p" ), n( "n" ), inp( "inp" ), scale( "scale", scale_ )
      { implementation-defined }
    };

  } // namespace sca_tdf

  typedef sca_eln::sca_tdf::sca_r sca_tdf_r;
```



```
} // namespace sca_eln
```

4.3.1.19.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_tdf::sca_r**".

4.3.1.20. sca_eln::sca_tdf::sca_c, sca_eln::sca_tdf_c

4.3.1.20.1. Description

The class **sca_eln::sca_tdf::sca_c** shall implement a primitive module for the ELN MoC that represents a capacitor, whose capacitance is controlled by a TDF input signal. The primitive shall contribute the following equation to the equation system:

$$i_{p,n}(t) = scale \cdot \frac{d\left(inp \cdot v_{p,n}(t) + q_0\right)}{dt}$$

where *scale* is the constant scale coefficient, *inp* is the TDF input signal, q_0 is the initial charge in coulomb, $v_{p,n}(t)$ is the voltage across terminals *p* and *n*, and $i_{p,n}(t)$ is the current flowing through the primitive from terminal *p* to terminal *n*. The product of *scale* and *inp* shall be interpreted as the capacitance in farad.

4.3.1.20.2. Class definition

```
namespace sca_eln {

  namespace sca_tdf {

    class sca_c : public sca_eln::sca_module,
                  public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      ::sca_tdf::sca_in<double> inp;

      sca_core::sca_parameter<double> scale;
      sca_core::sca_parameter<double> q0;

      virtual const char* kind() const;

      explicit sca_c( sc_core::sc_module_name, double scale_ = 1.0, double q0_ = 0.0 )
        : p( "p" ), n( "n" ), inp( "inp" ), scale( "scale", scale_ ), q0( "q0", q0_ )
      { implementation-defined }
    };

  } // namespace sca_tdf

  typedef sca_eln::sca_tdf::sca_c sca_tdf_c;

} // namespace sca_eln
```

4.3.1.20.3. Constraint of usage

The TDF input signal *inp* shall not be zero.

4.3.1.20.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_tdf::sca_c**".

4.3.1.21. sca_eln::sca_tdf::sca_l, sca_eln::sca_tdf_l

4.3.1.21.1. Description

The class **sca_eln::sca_tdf::sca_l** shall implement a primitive module for the ELN MoC that represents an inductor, whose inductance is controlled by a TDF input signal. The primitive shall contribute the following equation to the equation system:

$$v_{p,n}(t) = scale \cdot \frac{d(inp \cdot i_{p,n}(t) + phi_0)}{dt}$$

where *scale* is the constant scale coefficient, *inp* is the TDF input signal, phi_0 is the initial magnetic flux in weber, $v_{p,n}(t)$ is the voltage across terminals *p* and *n*, and $i_{p,n}(t)$ is the current flowing through the primitive from terminal *p* to terminal *n*. The product of *scale* and *inp* shall be interpreted as the inductance in henry.

4.3.1.21.2. Class definition

```
namespace sca_eln {

  namespace sca_tdf {

    class sca_l : public sca_eln::sca_module,
                  public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      ::sca_tdf::sca_in<double> inp;

      sca_core::sca_parameter<double> scale;
      sca_core::sca_parameter<double> phi0;

      virtual const char* kind() const;

      explicit sca_l( sc_core::sc_module_name, double scale_ = 1.0, double phi0_ = 0.0 )
        : p( "p" ), n( "n" ), inp( "inp" ), scale( "scale", scale_ ), phi0( "phi0", phi0_ )
      { implementation-defined }
    };

  } // namespace sca_tdf

  typedef sca_eln::sca_tdf::sca_l sca_tdf_l;

} //namespace sca_eln
```

4.3.1.21.3. Constraint of usage

The TDF input signal *inp* shall not be zero.

4.3.1.21.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_tdf::sca_l**".

4.3.1.22. sca_eln::sca_tdf::sca_rswitch, sca_eln::sca_tdf_rswitch

4.3.1.22.1. Description

The class **sca_eln::sca_tdf::sca_rswitch** shall implement a primitive module for the ELN MoC that represents a switch, which is controlled by a TDF control signal. The primitive shall contribute the following equation to the equation system:

$$v_{p,n}(t) = \begin{cases} r_{on} \cdot i_{p,n}(t) & ctrl \neq off\_state \\ r_{off} \cdot i_{p,n}(t) & ctrl = off\_state \end{cases}$$

where *ctrl* is the TDF control signal, r_{off} is the resistance of the switch in ohm under the condition that *off_state* is equal to the TDF control signal, and r_{on} is the resistance of the switch in ohm under the condition that *off_state* is not equal to the TDF control signal. $v_{p,n}(t)$ is the voltage across terminals *p* and *n*, and $i_{p,n}(t)$ is the current flowing through the primitive from terminal *p* to terminal *n*.

4.3.1.22.2. Class definition

```
namespace sca_eln {

  namespace sca_tdf {

    class sca_rswitch : public sca_eln::sca_module,
```



```
                      public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      ::sca_tdf::sca_in<bool> ctrl;

      sca_core::sca_parameter<double> ron;
      sca_core::sca_parameter<double> roff;
      sca_core::sca_parameter<bool>   off_state;

      virtual const char* kind() const;

      explicit sca_rswitch( sc_core::sc_module_name, double ron_ = 0.0,
                                                     double roff_ = sca_util::SCA_INFINITY,
                                                     bool off_state_ = false )
        : p( "p" ), n( "n" ), ctrl( "ctrl" ),
          ron( "ron", ron_ ), roff( "roff", roff_ ),
          off_state( "off_state", off_state_ )
      { implementation-defined }
    };

  } // namespace sca_tdf

  typedef sca_eln::sca_tdf::sca_rswitch sca_tdf_rswitch;

} // namespace sca_eln
```

4.3.1.22.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_tdf::sca_rswitch**".

4.3.1.23. sca_eln::sca_tdf::sca_vsource, sca_eln::sca_tdf_vsource

4.3.1.23.1. Description

The class **sca_eln::sca_tdf::sca_vsource** shall implement a primitive module for the ELN MoC that realizes the scaled conversion of a TDF signal to an ELN voltage source. The primitive shall contribute the following equation to the equation system:

$$v_{p,n}(t) = scale \cdot inp$$

where *scale* is the constant scale coefficient, *inp* is the TDF input signal that shall be interpreted as a continuous-time signal, and $v_{p,n}(t)$ is the voltage across terminals *p* and *n*. The product of *scale* and *inp* shall be interpreted as the voltage in volt.

4.3.1.23.2. Class definition

```
namespace sca_eln {

  namespace sca_tdf {

    class sca_vsource : public sca_eln::sca_module,
                        public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      ::sca_tdf::sca_in<double> inp;

      sca_core::sca_parameter<double> scale;

      virtual const char* kind() const;

      explicit sca_vsource( sc_core::sc_module_name, double scale_ = 1.0 )
        : p( "p" ), n( "n" ), inp( "inp" ), scale( "scale", scale_ )
      { implementation-defined }
    };

  } // namespace sca_tdf

  typedef sca_eln::sca_tdf::sca_vsource sca_tdf_vsource;
```

```
} // namespace sca_eln
```

4.3.1.23.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_tdf::sca_vsource**".

4.3.1.24. sca_eln::sca_tdf::sca_isource, sca_eln::sca_tdf_isource

4.3.1.24.1. Description

The class **sca_eln::sca_tdf::sca_isource** shall implement a primitive module for the ELN MoC that realizes the scaled conversion of a TDF signal to an ELN current source. The primitive shall contribute the following equation to the equation system:

$$i_{p,n}(t) = scale \cdot inp$$

where *scale* is the constant scale coefficient, *inp* is the TDF input signal that shall be interpreted as a continuous-time signal, and $i_{p,n}(t)$ is the current flowing through the primitive from terminal *p* to terminal *n*. The product of *scale* and *inp* shall be interpreted as the current in ampere.

4.3.1.24.2. Class definition

```
namespace sca_eln {

  namespace sca_tdf {

    class sca_isource : public sca_eln::sca_module,
                        public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      ::sca_tdf::sca_in<double> inp;

      sca_core::sca_parameter<double> scale;

      virtual const char* kind() const;

      explicit sca_isource( sc_core::sc_module_name, double scale_ = 1.0 )
        : p( "p" ), n( "n" ), inp( "inp" ), scale( "scale", scale_ )
      { implementation-defined }
    };

  } // namespace sca_tdf

  typedef sca_eln::sca_tdf::sca_isource sca_tdf_isource;

} // namespace sca_eln
```

4.3.1.24.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_tdf::sca_isource**".

4.3.1.25. sca_eln::sca_tdf::sca_vsink, sca_eln::sca_tdf_vsink

4.3.1.25.1. Description

The class **sca_eln::sca_tdf::sca_vsink** shall implement a primitive module for the ELN MoC that realizes a scaled conversion from an ELN voltage to a TDF output signal. The value of the voltage across terminals *p* and *n* shall be scaled with coefficient *scale* and written to a TDF output port *outp*.

4.3.1.25.2. Class definition

```
namespace sca_eln {

  namespace sca_tdf {
```



```
    class sca_vsink : public sca_eln::sca_module,
                      public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      ::sca_tdf::sca_out<double> outp;

      sca_core::sca_parameter<double> scale;

      virtual const char* kind() const;

      explicit sca_vsink( sc_core::sc_module_name, double scale_ = 1.0 )
        : p( "p" ), n( "n" ), outp( "outp" ), scale( "scale", scale_ )
      { implementation-defined }
    };

  } // namespace sca_tdf

  typedef sca_eln::sca_tdf::sca_vsink sca_tdf_vsink;

} // namespace sca_eln
```

4.3.1.25.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_tdf::sca_vsink**".

4.3.1.26. sca_eln::sca_tdf::sca_isink, sca_eln::sca_tdf_isink

4.3.1.26.1. Description

The class **sca_eln::sca_tdf::sca_isink** shall implement a primitive module for the ELN MoC that realizes a scaled conversion from an ELN current to a TDF output signal. The value of the current flowing through the primitive from terminal *p* to terminal *n* shall be scaled with coefficient *scale* and written to a TDF output port *outp*. The primitive shall contribute the following equation to the equation system:

$$v_{p,n}(t) = 0$$

where $v_{p,n}(t)$ is the voltage across terminals *p* and *n*.

4.3.1.26.2. Class definition

```
namespace sca_eln {

  namespace sca_tdf {

    class sca_isink : public sca_eln::sca_module,
                      public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      ::sca_tdf::sca_out<double> outp;

      sca_core::sca_parameter<double> scale;

      virtual const char* kind() const;

      explicit sca_isink( sc_core::sc_module_name, double scale_ = 1.0 )
        : p( "p" ), n( "n" ), outp( "outp" ), scale( "scale", scale_ )
      { implementation-defined }
    };

  } // namespace sca_tdf

  typedef sca_eln::sca_tdf::sca_isink sca_tdf_isink;

} // namespace sca_eln
```

4.3.1.26.3. kind

```
virtual const char* kind() const;
```


The member function **kind** shall return the string "**sca_eln::sca_tdf::sca_isink**".

4.3.1.27. sca_eln::sca_de::sca_r, sca_eln::sca_de_r

4.3.1.27.1. Description

The class **sca_eln::sca_de::sca_r** shall implement a primitive module for the ELN MoC that represents a resistor, whose resistance is controlled by a discrete-event input signal. The primitive shall contribute the following equation to the equation system:

$$v_{p,n}(t) = scale \cdot inp \cdot i_{p,n}(t)$$

where *scale* is the constant scale coefficient, *inp* is the discrete-event input signal, $v_{p,n}(t)$ is the voltage across terminals *p* and *n*, and $i_{p,n}(t)$ is the current flowing through the primitive from terminal *p* to terminal *n*. The product of *scale* and *inp* shall be interpreted as the resistance in ohm.

4.3.1.27.2. Class definition

```
namespace sca_eln {

  namespace sca_de {

    class sca_r : public sca_eln::sca_module,
                  public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      sc_core::sc_in<double> inp;

      sca_core::sca_parameter<double> scale;

      virtual const char* kind() const;

      explicit sca_r( sc_core::sc_module_name, double scale_ = 1.0 )
        : p( "p" ), n( "n" ), inp( "inp" ), scale( "scale", scale_ )
      { implementation-defined }
    };

  } // namespace sca_de

  typedef sca_eln::sca_de::sca_r sca_de_r;

} // namespace sca_eln
```

4.3.1.27.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_de::sca_r**".

4.3.1.28. sca_eln::sca_de::sca_c, sca_eln::sca_de_c

4.3.1.28.1. Description

The class **sca_eln::sca_de::sca_c** shall implement a primitive module for the ELN MoC that represents a capacitor, whose capacitance is controlled by a discrete-event input signal. The primitive shall contribute the following equation to the equation system:

$$i_{p,n}(t) = scale \cdot \frac{d\left(inp \cdot v_{p,n}(t) + q_0\right)}{dt}$$

where *scale* is the constant scale coefficient, *inp* is the discrete-event input signal, q_0 is the initial charge in coulomb, $v_{p,n}(t)$ is the voltage across terminals *p* and *n*, and $i_{p,n}(t)$ is the current flowing through the primitive from terminal *p* to terminal *n*. The product of *scale* and *inp* shall be interpreted as the capacitance in farad.

4.3.1.28.2. Class definition

```
namespace sca_eln {
```



```
  namespace sca_de {

    class sca_c : public sca_eln::sca_module,
                  public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      sc_core::sc_in<double> inp;

      sca_core::sca_parameter<double> scale;
      sca_core::sca_parameter<double> q0;

      virtual const char* kind() const;

      explicit sca_c( sc_core::sc_module_name, double scale_ = 1.0, double q0_ = 0.0 )
        : p( "p" ), n( "n" ), inp( "inp" ), scale( "scale", scale_ ), q0( "q0", q0_ )
      { implementation-defined }
    };

  } // namespace sca_de

  typedef sca_eln::sca_de::sca_c sca_de_c;

} // namespace sca_eln
```

4.3.1.28.3. Constraint of usage

The discrete-event input signal *inp* shall not be zero.

4.3.1.28.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_de::sca_c**".

4.3.1.29. sca_eln::sca_de::sca_l, sca_eln::sca_de_l

4.3.1.29.1. Description

The class **sca_eln::sca_de::sca_l** shall implement a primitive module for the ELN MoC that represents an inductor, whose inductance is controlled by a discrete-event input signal. The primitive shall contribute the following equation to the equation system:

$$v_{p,n}(t) = scale \cdot \frac{d(inp \cdot i_{p,n}(t) + phi_0)}{dt}$$

where *scale* is the constant scale coefficient, *inp* is the discrete-event input signal, phi_0 is the initial magnetic flux in weber, $v_{p,n}(t)$ is the voltage across terminals *p* and *n*, and $i_{p,n}(t)$ is the current flowing through the primitive from terminal *p* to terminal *n*. The product of *scale* and *inp* shall be interpreted as the inductance in henry.

4.3.1.29.2. Class definition

```
namespace sca_eln {

  namespace sca_de {

    class sca_l : public sca_eln::sca_module,
                  public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      sc_core::sc_in<double> inp;

      sca_core::sca_parameter<double> scale;
      sca_core::sca_parameter<double> phi0;

      virtual const char* kind() const;
```

```
      explicit sca_l( sc_core::sc_module_name, double scale_ = 1.0, double phi0_ = 0.0 )
        : p( "p" ), n( "n" ), inp( "inp" ), scale( "scale", scale_ ), phi0( "phi0", phi0_ )
      { implementation-defined }
    };

  } // namespace sca_de

  typedef sca_eln::sca_de::sca_l sca_de_l;

} // namespace sca_eln
```

4.3.1.29.3. Constraint of usage

The discrete-event input signal *inp* shall not be zero.

4.3.1.29.4. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_de::sca_l**".

4.3.1.30. sca_eln::sca_de::sca_rswitch, sca_eln::sca_de_rswitch

4.3.1.30.1. Description

The class **sca_eln::sca_de::sca_rswitch** shall implement a primitive module for the ELN MoC that represents a switch, which is controlled by a discrete-event control signal. The primitive shall contribute the following equation to the equation system:

$$v_{p,n}(t) = \begin{cases} r_{on} \cdot i_{p,n}(t) & ctrl \neq off\_state \\ r_{off} \cdot i_{p,n}(t) & ctrl = off\_state \end{cases}$$

where *ctrl* is the discrete-event control signal, r_{off} is the resistance of the switch in ohm under the condition that *off_state* is equal to the discrete-event control signal, and r_{on} is the resistance of the switch in ohm under the condition that *off_state* is not equal to the discrete-event control signal. $v_{p,n}(t)$ is the voltage across terminals *p* and *n*, and $i_{p,n}(t)$ is the current flowing through the primitive from terminal *p* to terminal *n*.

4.3.1.30.2. Class definition

```
namespace sca_eln {

  namespace sca_de {

    class sca_rswitch : public sca_eln::sca_module,
                        public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      sc_core::sc_in<bool> ctrl;

      sca_core::sca_parameter<double> ron;
      sca_core::sca_parameter<double> roff;
      sca_core::sca_parameter<bool>   off_state;

      virtual const char* kind() const;

      explicit sca_rswitch( sc_core::sc_module_name, double ron_ = 0.0,
                                                     double roff_ = sca_util::SCA_INFINITY,
                                                     bool   off_state_ = false )
        : p( "p" ), n( "n" ), ctrl( "ctrl" ),
          ron( "ron", ron_ ), roff( "roff", roff_ ),
          off_state( "off_state", off_state_ )
      { implementation-defined }
    };

  } // namespace sca_de

  typedef sca_eln::sca_de::sca_rswitch sca_eln::sca_de_rswitch;

} // namespace sca_eln
```


4.3.1.30.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_de::sca_rswitch**".

4.3.1.31. sca_eln::sca_de::sca_vsource, sca_eln::sca_de_vsource

4.3.1.31.1. Description

The class **sca_eln::sca_de::sca_vsource** shall implement a primitive module for the ELN MoC that realizes the scaled conversion of a discrete-event signal to an ELN voltage source. The primitive shall contribute the following equation to the equation system:

$$v_{p,n}(t) = scale \cdot inp$$

where *scale* is the constant scale coefficient, *inp* is the discrete-event input signal that shall be interpreted as a discrete-time signal, and $v_{p,n}(t)$ is the voltage across terminals *p* and *n*. The product of *scale* and *inp* shall be interpreted as the voltage in volt.

4.3.1.31.2. Class definition

```
namespace sca_eln {

  namespace sca_de {

    class sca_vsource : public sca_eln::sca_module,
                        public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      sc_core::sc_in<double> inp;

      sca_core::sca_parameter<double> scale;

      virtual const char* kind() const;

      explicit sca_vsource( sc_core::sc_module_name, double scale_ = 1.0 )
        : p( "p" ), n( "n" ), inp( "inp" ), scale( scale_ )
      { implementation-defined }
    };

  } // namespace sca_de

  typedef sca_eln::sca_de::sca_vsource sca_de_vsource;

} // namespace sca_eln
```

4.3.1.31.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_de::sca_vsource**".

4.3.1.32. sca_eln::sca_de::sca_isource, sca_eln::sca_de_isource

4.3.1.32.1. Description

The class **sca_eln::sca_de::sca_isource** shall implement a primitive module for the ELN MoC that realizes the scaled conversion of a discrete-event signal to an ELN current source. The primitive shall contribute the following equation to the equation system:

$$i_{p,n}(t) = scale \cdot inp$$

where *scale* is the constant scale coefficient, *inp* is the discrete-event input signal that shall be interpreted as a discrete-time signal, and $i_{p,n}(t)$ is the current flowing through the primitive from terminal *p* to terminal *n*. The product of *scale* and *inp* shall be interpreted as the current in ampere.

4.3.1.32.2. Class definition

```
namespace sca_eln {

  namespace sca_de {

    class sca_isource : public sca_eln::sca_module,
                        public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      sc_core::sc_in<double> inp;

      sca_core::sca_parameter<double> scale;

      virtual const char* kind() const;

      explicit sca_isource( sc_core::sc_module_name, double scale_ = 1.0 )
        : p( "p" ), n( "n" ), inp( "inp" ), scale( "scale", scale_ )
      { implementation-defined }
    };

  } // namespace sca_de

  typedef sca_eln::sca_de::sca_isource sca_de_isource;

} // namespace sca_eln
```

4.3.1.32.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_de::sca_isource**".

4.3.1.33. sca_eln::sca_de::sca_vsink, sca_eln::sca_de_vsink

4.3.1.33.1. Description

The class **sca_eln::sca_de::sca_vsink** shall implement a primitive module for the ELN MoC that realizes a scaled conversion from an ELN voltage to a discrete-event output signal. The value of the voltage across terminals *p* and *n* shall be scaled with coefficient *scale* and written to a discrete-event output port *outp*.

4.3.1.33.2. Class definition

```
namespace sca_eln {

  namespace sca_de {

    class sca_vsink : public sca_eln::sca_module,
                      public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      sc_core::sc_out<double> outp;

      sca_core::sca_parameter<double> scale;

      virtual const char* kind() const;

      explicit sca_vsink( sc_core::sc_module_name, double scale_ = 1.0 )
        : p( "p" ), n( "n" ), outp( "outp" ), scale( "scale", scale_ )
      { implementation-defined }
    };

  } // namespace sca_de

  typedef sca_eln::sca_de::sca_vsink sca_de_vsink;

} // namespace sca_eln
```

4.3.1.33.3. kind

```
virtual const char* kind() const;
```


The member function **kind** shall return the string "**sca_eln::sca_de::sca_vsink**".

4.3.1.34. sca_eln::sca_de::sca_isink, sca_eln::sca_de_isink

4.3.1.34.1. Description

The class **sca_eln::sca_de::sca_isink** shall implement a primitive module for the ELN MoC that realizes a scaled conversion from an ELN current to a discrete-event output signal. The value of the current flowing through the primitive from terminal *p* to terminal *n* shall be scaled with coefficient *scale* and written to a discrete-event output port *outp*. The primitive shall contribute the following equation to the equation system:

$$v_{p,n}(t) = 0$$

where $v_{p,n}(t)$ is the voltage across terminals *p* and *n*.

4.3.1.34.2. Class definition

```
namespace sca_eln {

  namespace sca_de {

    class sca_isink : public sca_eln::sca_module,
                      public sca_util::sca_traceable_object†
    {
     public:
      sca_eln::sca_terminal p;
      sca_eln::sca_terminal n;

      sc_core::sc_out<double> outp;

      sca_core::sca_parameter<double> scale;

      virtual const char* kind() const;

      explicit sca_isink( sc_core::sc_module_name, double scale_ = 1.0 )
        : p( "p" ), n( "n" ), outp( "outp" ), scale( "scale", scale_ )
      { implementation-defined }
    };

  } // namespace sca_de

  typedef sca_eln::sca_de::sca_isink sca_de_isink;

} // namespace sca_eln
```

4.3.1.34.3. kind

```
virtual const char* kind() const;
```

The member function **kind** shall return the string "**sca_eln::sca_de::sca_isink**".

4.3.2. Hierarchical ELN composition and port binding

The hierarchical composition of ELN modules shall use modules derived from class **sc_core::sc_module** and the constructor or its equivalent macro definitions. A hierarchical module can include modules and ports of different models of computation. Port binding rules shall follow IEEE Std 1666-2005 as well as the following specific rules:

- A port of class **sca_eln::sca_terminal** shall only be bound to a primitive channel of class **sca_eln::sca_node**, **sca_eln::sca_node_ref** or to a port of class **sca_eln::sca_terminal** of the parent module.
- A port of class **sca_eln::sca_terminal** shall be bound to exactly one primitive channel of class **sca_eln::sca_node** or **sca_eln::sca_node_ref** throughout the whole hierarchy.
- A primitive channel of class **sca_eln::sca_node** or **sca_eln::sca_node_ref** shall have one or more primitive ports of class **sca_eln::sca_terminal** bound to it throughout the whole hierarchy.
- For each cluster of connected predefined ELN primitive modules, at least one port of class **sca_eln::sca_terminal** shall be bound to a primitive channel of class **sca_eln::sca_node_ref**.

Predefined ELN primitive modules with ports of other models of computation shall follow the port binding rules of the corresponding models of computation.

4.3.3. ELN MoC elaboration and simulation

An implementation of the ELN MoC in a SystemC AMS class library shall include a public shell consisting of the predefined classes, functions, and so forth that can be used directly by an application. An implementation shall also include an ELN solver that implements the functionality of the ELN class library. The underlying semantics of the ELN solver are defined in this subclause.

The execution of a SystemC AMS application that includes ELN modules consists of elaboration followed by simulation. Elaboration results in one or more equation systems setup by the contributions of the ELN modules. Simulation solves the equation systems repetitively. In addition to providing support for elaboration and simulation, the ELN solver may also provide implementation-specific functionality beyond the scope of this standard. As an example of such functionality, the ELN solver may report information on the ELN module composition and equation setup.

4.3.3.1. ELN elaboration

The primary purpose of ELN elaboration is to create internal data structures and setup equations for the ELN solver to support the semantics of ELN simulation. The ELN elaboration as described in this clause and in the following subclauses shall execute in one **sc_core::sc_module::end_of_elaboration** callback. The actions stated in the following subclauses shall occur, in the given order, during ELN elaboration and only during ELN elaboration. The description of such actions use the concept of an ELN cluster, which is a set of ELN modules connected by channels of class **sca_eln::sca_node**.

ELN elaboration shall lock the parameter values of the predefined ELN primitive modules. (See 3.2.6).

NOTE—Connections by channels of class **sca_eln::sca_node_ref** are ignored for building ELN clusters.

4.3.3.1.1. Timestep calculation and propagation

The timestep for every ELN cluster shall be derived from the timestep of a connected TDF cluster or set by the member function **set_timestep** of an ELN primitive module derived from class **sca_eln::sca_module** of the corresponding ELN cluster. The timestep shall be propagated within the ELN cluster to all primitive modules and to all ports of class **sca_tdf::sca_in** and **sca_tdf::sca_out**, if any.

It shall be an error if a timestep value is not assigned to at least one ELN module or if inconsistent timesteps are defined in an ELN cluster.

After successful ELN elaboration, all assigned timestep values shall be overridden by the propagated timestep values.

NOTE—An ELN cluster can be considered as one TDF module, which could be connected to TDF modules in a hierarchical composition by the ports of class **sca_tdf::sca_in** and **sca_tdf::sca_out** of the predefined ELN primitive modules. The ELN cluster is included in the timestep calculation of the TDF cluster and must comply to the same rules (see 4.1.3.1.2).

4.3.3.1.2. ELN equation system setup and solvability check

For each ELN cluster, an equation system shall be setup by combining:

1. the contributing equations of each of the predefined ELN primitive modules in the cluster.

2. the equations implied by the Kirchhoff's Laws.

It shall be an error if any of the equation systems is numerically singular.

For each port of class **sca_eln::sca_terminal**, the voltage across the terminal and the corresponding reference node of class **sca_eln::sca_node_ref** shall be defined due to Kirchhoff's Voltage Law. It shall be an error if this voltage is undefined.

4.3.3.2. ELN simulation

This subclause defines the process of time-domain simulation of ELN descriptions. The simulation of a cluster of ELN modules is done by a repetitive solving of the underlying equation systems.

4.3.3.2.1. ELN initialization

The ELN initialization phase calculates consistent initial conditions for the equation systems.

4.3.3.2.2. Time-domain simulation

The solver shall provide results at least at the calculated timestep distances.

4.3.3.2.3. Synchronization with TDF MoC

Synchronization with the TDF MoC shall be done exclusively by using the predefined ELN primitive modules containing ports of class **sca_tdf::sca_in** and **sca_tdf::sca_out**.

The ELN solver reads repetitively samples from ports of class **sca_tdf::sca_in** for all calculated timesteps of the ELN cluster. Consecutive reads shall be interpreted as forming a continuous-time signal.

The ELN solver writes repetitively samples to ports of class **sca_tdf::sca_out** for all calculated timesteps of the ELN cluster.

4.3.3.2.4. Synchronization with SystemC kernel

Synchronization with the SystemC kernel shall be done exclusively by using the predefined ELN primitive modules containing ports of class **sc_core::sc_in** and **sc_core::sc_out**.

The ELN solver reads repetitively values from ports of class **sc_core::sc_in** at each first delta cycle of the corresponding SystemC time for all calculated timesteps of the ELN cluster. The value is assumed as constant until the next value is read.

The ELN solver writes repetitively values to ports of class **sc_core::sc_out** at each first delta cycle of the corresponding SystemC time for all calculated timesteps of the ELN cluster.

4.3.3.3. Running elaboration and simulation

The implementation shall use the same elaboration and simulation semantics as defined in IEEE Std 1666-2005 (SystemC Language Reference Manual).

NOTE—ELN modules can be instantiated in the **sc_main** context and the elaboration and simulation can be controlled by the function **sc_core::sc_start**.

This page is intentionally left blank.

5. Predefined analyses

5.1. Time-domain analysis

The time-domain analysis shall be applicable to all descriptions supported by the predefined models of computation as defined in Clause 4. The analysis shall compute the time-domain behavior of the overall system, possibly composed by different models of computation and including descriptions as defined in IEEE Std 1666-2005 (SystemC Language Reference Manual).

5.1.1. Elaboration and simulation

The execution of a time-domain analysis consists of elaboration followed by simulation (see 4.1.3, 4.2.3, 4.3.3). The elaboration and simulation shall use the same semantics as defined in IEEE Std 1666-2005 (SystemC Language Reference Manual).

NOTE—TDF, LSF, and ELN modules can be instantiated in the **sc_main** context and the elaboration and simulation can be controlled by the function **sc_core::sc_start**.

5.2. Small-signal frequency-domain analyses

The small-signal frequency-domain analyses shall be applicable to all descriptions supported by the predefined models of computation defined in Clause 4. The analyses shall compute the small-signal frequency-domain behavior of the overall system, possibly composed of modules from different models of computation. The system description shall be mapped to a linear complex equation system.

Two kinds of small-signal frequency-domain analysis shall be supported:

1. Small-signal frequency-domain analysis shall solve for each frequency point the linear complex equation system including all small-signal frequency-domain source contributions.
2. Small-signal frequency-domain noise analysis shall solve 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, shall be set to zero.

All functions used in the small-signal frequency-domain and noise analysis shall be placed in the namespace **sca_ac_analysis**.

5.2.1. Elaboration and simulation

The execution of a small-signal frequency-domain or noise simulation consists of elaboration followed by simulation. For starting a small-signal frequency-domain or noise analysis, dedicated functions shall be used (see 5.2.1.3). While performing the analysis, the state of the time-domain simulation shall not be changed.

5.2.1.1. Elaboration

The small-signal frequency-domain elaboration shall be performed if one of the dedicated start functions is executed (see 5.2.1.3). In the case a time-domain elaboration has not yet been performed (due **sc_core::sc_start** has not yet been executed) the implementation shall perform a time-domain elaboration first.

The implementation shall setup one complex linear frequency dependent equation system by composing the equation system contributions of TDF, LSF, and ELN descriptions.

5.2.1.2. Simulation

The linear complex equation system for the chosen analysis kind, shall be solved for each frequency point and in dependency of the analysis kind.

5.2.1.3. Running elaboration and simulation

The implementation shall provide the function **sca_ac_analysis::sca_ac_start** and **sca_ac_analysis::sca_ac_noise_start** for running small-signal frequency-domain elaboration and simulation.

When called, functions **sca_ac_analysis::sca_ac_start** and **sca_ac_analysis::aca_ac_noise_start** shall first run elaboration as described in Subclause 5.1, if not yet performed.

5.2.1.3.1. sca_ac_analysis::sca_ac_start

```
namespace sca_ac_analysis {

  enum sca_ac_scale { SCA_LOG, SCA_LIN };

  void sca_ac_start( double start_freq, double stop_freq, unsigned long npoints,
                sca_ac_analysis::sca_ac_scale scale = sca_ac_analysis::SCA_LOG );

  void sca_ac_start( const sca_util::sca_vector<double>& frequencies );

} // namespace sca_ac_analysis
```

The functions **sca_ac_analysis::sca_ac_start** shall perform a small-signal frequency-domain simulation. The first function shall calculate the frequency domain behavior at *npoints* frequencies. If *npoints* is greater than zero, the first frequency point in hertz shall be *start_freq*. If *npoints* is greater than one, the last frequency point in hertz shall be *stop_freq*. If *scale* is **sca_ac_analysis::SCA_LOG**, the remaining frequency points shall be distributed logarithmically and if *scale* is **sca_ac_analysis::SCA_LIN**, the remaining points shall be distributed linear.

The second function shall calculate the small-signal frequency-domain behavior at the frequency points given by the vector *frequencies*.

5.2.1.3.2. sca_ac_analysis::sca_ac_noise_start

```
namespace sca_ac_analysis {

  void sca_ac_noise_start( double start_freq, double stop_freq, unsigned long npoints,
                           sca_ac_analysis::sca_ac_scale scale = sca_ac_analysis::SCA_LOG );

  void sca_ac_noise_start( const sca_util::sca_vector<double>& frequencies );

} // namespace sca_ac_analysis
```

The functions **sca_ac_analysis::sca_ac_noise_start** shall perform a small-signal frequency-domain noise simulation. The first function shall calculate the frequency-domain noise behavior at *npoints* frequencies. If *npoints* is greater than zero, the first frequency point in hertz shall be *start_freq*. If *npoints* is greater than one, the last frequency point in hertz shall be *stop_freq*. If *scale* is **sca_ac_analysis::SCA_LOG**, the remaining frequency points shall be distributed logarithmically and if *scale* is **sca_ac_analysis::SCA_LIN**, the remaining points shall be distributed linear.

The second function shall calculate the frequency-domain noise behavior at the frequency points given by the vector *frequencies*.

5.2.2. Small-signal frequency-domain analysis of TDF descriptions

The small-signal frequency-domain and noise representation of a TDF description shall contribute the following complex equation system:

$$A(f) \cdot x + b(f) + b_{noise}(f) + c(f,x) = 0$$

where $A(f)$ is a complex matrix of the frequency f that shall include contributions of modules derived from class **sca_tdf::sca_module**. Each module derived from class **sca_tdf::sca_module** can provide the implementation of the member function **ac_processing** (see 4.1.1.1.8) or the corresponding registered member function (see 4.1.1.1.10). The contributions shall describe linear complex functions between ports of class **sca_tdf::sca_in** and ports of class **sca_tdf::sca_out**.

x is a complex vector representing the small-signal frequency-domain values of the ports of class **sca_tdf::sca_out**.

$b(f)$ and $b_{noise}(f)$ are complex frequency dependent vectors, which represent the contributions to the ports of class **sca_tdf::sca_out** independent from the ports of class **sca_tdf::sca_in**.

For small-signal frequency-domain analysis, the independent contribution $b(f)$ shall be provided to the equation system by using the function **sca_ac_analysis::sca_ac** for accessing the port of class **sca_tdf::sca_out**. In this case the contribution $b_{noise}(f)$ shall be set to zero.

For small-signal frequency-domain noise analysis, the independent contribution $b_{noise}(f)$ shall be provided to the equation system by using the function **sca_ac_analysis::sca_ac_noise** for accessing the port of class **sca_tdf::sca_out**. In this case the contribution $b(f)$ shall be set to zero.

$c(f,x)$ is a vector of contributions from interaction with LSF or ELN primitives and may depend on TDF small-signal frequency-domain values of the ports of class **sca_tdf::sca_out**.

The implementation shall permit the access to time-domain values and complex frequency-domain values at ports. The access to complex frequency-domain values shall be done by the functions **sca_ac_analysis::sca_ac** or **sca_ac_analysis::sca_ac_noise** (see 5.2.2.1 and 5.2.2.2), while the time-domain values shall be accessible by using the member functions to read from a port of class **sca_tdf::sca_de::sca_in** or **sca_tdf::sca_in**.

If no value of type **sca_util::sca_complex** has been assigned to ports of class **sca_tdf::sca_out**, using the functions **sca_ac_analysis::sca_ac** or **sca_ac_analysis::sca_ac_noise** respectively, the implementation shall set these values to zero.

NOTE—It is not defined in which order and how often the member functions **sca_tdf::sca_module::ac_processing** are executed.

5.2.2.1. sca_ac_analysis::sca_ac

```
namespace sca_ac_analysis {

  template<class T>
  const sca_util::sca_complex& sca_ac( const sca_tdf::sca_in<T>& );

  template<class T>
  sca_util::sca_complex& sca_ac( const sca_tdf::sca_out<T>& );

} // namespace sca_ac_analysis
```

The function **sca_ac_analysis::sca_ac** applied to ports of class **sca_tdf::sca_in** shall return a const reference to a value of type **sca_util::sca_complex** of the corresponding port.

The function **sca_ac_analysis::sca_ac** applied to ports of class **sca_tdf::sca_out** shall return a reference to a value of type **sca_util::sca_complex** to allow assignment of a contribution to this port.

It shall be an error if the functions are called outside the context of the member function **sca_tdf::sca_module::ac_processing** or its equivalent registered member function.

NOTE—The values of type **sca_util::sca_complex** read using the function **sca_ac_analysis::sca_ac** from the ports of class **sca_tdf::sca_in** are implementation-defined.

5.2.2.2. sca_ac_analysis::sca_ac_noise

```
namespace sca_ac_analysis {

  template<class T>
  const sca_util::sca_complex& sca_ac_noise( const sca_tdf::sca_in<T>& );

  template<class T>
  sca_util::sca_complex& sca_ac_noise( const sca_tdf::sca_out<T>& );

} // namespace sca_ac_analysis
```

The function **sca_ac_analysis::sca_ac_noise** applied to ports of class **sca_tdf::sca_in** shall return a const reference to a value of type **sca_util::sca_complex** of the corresponding port.

The function **sca_ac_analysis::sca_ac_noise** applied to port of class **sca_tdf::sca_out** shall return a reference to a value of type **sca_util::sca_complex** to allow assignment of a noise contribution to this port.

It shall be an error if the function is called outside the context of the member function **sca_tdf::sca_module::ac_processing** or its equivalent registered member function.

5.2.2.3. sca_ac_analysis::sca_ac_is_running

```
namespace sca_ac_analysis {

  bool sca_ac_is_running();

} // namespace sca_ac_analysis
```

The function **sca_ac_analysis::sca_ac_is_running** shall return true while performing a small-signal frequency-domain or a small-signal noise simulation and false otherwise.

5.2.2.4. sca_ac_analysis::sca_ac_noise_is_running

```
namespace sca_ac_analysis {

  bool sca_ac_noise_is_running();

} // namespace sca_ac_analysis
```

The function **sca_ac_analysis::sca_ac_noise_is_running** shall return true while performing a small-signal frequency-domain noise simulation and false otherwise.

5.2.2.5. sca_ac_analysis::sca_ac_f

```
namespace sca_ac_analysis {

  double sca_ac_f();

} // namespace sca_ac_analysis
```

The function **sca_ac_analysis::sca_ac_f** shall return the current frequency in hertz.

It shall be an error if the function is called outside the context of the member function **sca_tdf::sca_module::ac_processing** or its equivalent registered member function.

5.2.2.6. sca_ac_analysis::sca_ac_w

```
namespace sca_ac_analysis {

  double sca_ac_w();

} // namespace sca_ac_analysis
```

The function **sca_ac_analysis::sca_ac_w** shall return the current angular frequency in radians per second (rad/s).

It shall be an error if the function is called outside the context of the member function **sca_tdf::sca_module::ac_processing** or its equivalent registered member function.

5.2.2.7. sca_ac_analysis::sca_ac_s

```
namespace sca_ac_analysis {

  sca_util::sca_complex sca_ac_s( long n = 1 );

} // namespace sca_ac_analysis
```

The function **sca_ac_analysis::sca_ac_s** shall return the complex value of the Laplace operator $s^n = (j\omega)^n$.

It shall be an error if the function is called outside the context of the member function **sca_tdf::sca_module::ac_processing** or its equivalent registered member function.

5.2.2.8. sca_ac_analysis::sca_ac_z

```
namespace sca_ac_analysis {

  sca_util::sca_complex sca_ac_z( long n, const sca_core::sca_time& tstep );

  sca_util::sca_complex sca_ac_z( long n = 1 );

} // namespace sca_ac_analysis
```

The functions **sca_ac_analysis::sca_ac_z** shall return the complex value of the *z* operator z^n ($e^{j\omega \cdot n \cdot tstep}$). If not specified, the argument *tstep* shall be set to the value returned by the member function **get_timestep** of the module of class **sca_tdf::sca_module**, in which the function is called.

It shall be an error if the function is called outside the context of the member function **sca_tdf::sca_module::ac_processing** or its equivalent registered member function.

5.2.2.9. sca_ac_analysis::sca_ac_delay

```
namespace sca_ac_analysis {

  sca_util::sca_complex sca_ac_delay( const sca_core::sca_time& delay );

} // namespace sca_ac_analysis
```

The function **sca_ac_analysis::sca_ac_delay** shall return the complex value of the continuous time delay ($e^{-j\omega \cdot delay}$).

It shall be an error if the function is called outside the context of the member function **sca_tdf::sca_module::ac_processing** or its equivalent registered member function.

5.2.2.10. sca_ac_analysis::sca_ac_ltf_nd

```
namespace sca_ac_analysis {

  sca_util::sca_complex sca_ac_ltf_nd( const sca_util::sca_vector<double>& num,
                                       const sca_util::sca_vector<double>& den,
                                       const sca_util::sca_complex& input = 1.0,
                                             double k = 1.0 );

  sca_util::sca_complex sca_ac_ltf_nd( const sca_util::sca_vector<double>& num,
                                       const sca_util::sca_vector<double>& den,
                                             sca_core::sca_time delay,
                                       const sca_util::sca_complex& input = 1.0,
                                             double k = 1.0 );

} // namespace sca_ac_analysis
```

The function **sca_ac_analysis::sca_ac_ltf_nd** shall return the complex value of the linear transfer function of the Laplace-domain variable *s* in numerator-denumerator form (see 4.1.4.3) with $s^i = (j\omega)^i$, multiplied by the complex value *input*.

It shall be an error if the function is called outside the context of the member function **sca_tdf::sca_module::ac_processing** or its equivalent registered member function.

5.2.2.11. sca_ac_analysis::sca_ac_ltf_zp

```
namespace sca_ac_analysis {

  sca_util::sca_complex sca_ac_ltf_zp( const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                                       const sca_util::sca_vector<sca_util::sca_complex>& poles,
                                       const sca_util::sca_complex& input = 1.0,
                                             double k = 1.0 );

  sca_util::sca_complex sca_ac_ltf_zp( const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                                       const sca_util::sca_vector<sca_util::sca_complex>& poles,
                                             sca_core::sca_time delay,
                                       const sca_util::sca_complex& input = 1.0,
                                             double k = 1.0 );

} // namespace sca_ac_analysis
```

The function **sca_ac_analysis::sca_ac_ltf_zp** shall return the complex value of the linear transfer function of the Laplace-domain variable *s* in zero-pole form (see 4.1.4.4) with $s^i = (j\omega)^i$, multiplied by the complex value *input*.

It shall be an error if the function is called outside the context of the member function **sca_tdf::sca_module::ac_processing** or its equivalent registered member function.

5.2.2.12. sca_ac_analysis::sca_ac_ss

```
namespace sca_ac_analysis {

  sca_util::sca_vector<sca_util::sca_complex> sca_ac_ss(
                                          const sca_util::sca_matrix<double>& a,
                                          const sca_util::sca_matrix<double>& b,
                                          const sca_util::sca_matrix<double>& c,
                                          const sca_util::sca_matrix<double>& d,
                                          const sca_util::sca_vector<sca_util::sca_complex>& input );

  sca_util::sca_vector<sca_util::sca_complex> sca_ac_ss(
                                          const sca_util::sca_matrix<double>& a,
                                          const sca_util::sca_matrix<double>& b,
                                          const sca_util::sca_matrix<double>& c,
                                          const sca_util::sca_matrix<double>& d );

  sca_util::sca_vector<sca_util::sca_complex> sca_ac_ss(
                                          const sca_util::sca_matrix<double>& a,
                                          const sca_util::sca_matrix<double>& b,
                                          const sca_util::sca_matrix<double>& c,
                                          const sca_util::sca_matrix<double>& d,
                                          sca_core::sca_time delay,
                                          const sca_util::sca_vector<sca_util::sca_complex>& input );

  sca_util::sca_vector<sca_util::sca_complex> sca_ac_ss(
                                          const sca_util::sca_matrix<double>& a,
                                          const sca_util::sca_matrix<double>& b,
                                          const sca_util::sca_matrix<double>& c,
                                          const sca_util::sca_matrix<double>& d,
                                          sca_core::sca_time delay );

} // namespace sca_ac_analysis
```

The functions **sca_ac_analysis::sca_ac_ss** shall return the complex vector *y* of the state-space equation system with $(d/dt)^i = (j\omega)^i$. The function with the complex vector *input* as argument shall multiply the complex vector *y* with *input*. It shall be an error if the matrix and vector sizes are inconsistent (see 4.1.4.5) .

It shall be an error if the function is called outside the context of the member function **sca_tdf::sca_module::ac_processing** or its equivalent registered member function.

5.2.3. Small-signal frequency-domain analysis of LSF descriptions

The implementation of the LSF primitive modules shall define their small-signal frequency-domain behavior.

Therefore the equation system for each LSF cluster (see 4.2.3.1.2) shall be transformed from time-domain to small-signal frequency-domain by replacing a derivation *d/dt* by $j\omega$, an integral by $1/(j\omega)$ respectively and a *delay* by $e^{-j\omega \cdot delay}$. The resulting equation systems shall be contributed to the overall equation system.

5.2.4. Small-signal frequency-domain analysis of ELN descriptions

The implementation of the ELN primitive modules shall define their small-signal frequency-domain behavior.

Therefore the equation system for each ELN cluster (see 4.3.3.1.2) shall be transformed from time-domain to small-signal frequency-domain by replacing a derivation *d/dt* by $j\omega$, and a *delay* by $e^{-j\omega \cdot delay}$. The resulting equation system shall be contributed to the overall equation system.

6. Utility definitions

6.1. AMS trace files

An AMS trace file records the simulation results for AMS signals and nodes. At least the tabular and the VCD trace file format shall be supported. The VCD format can only support tracing for time-domain simulation.

A VCD trace file can only be created and opened by calling function **sca_util::sca_create_vcd_trace_file** and a tabular trace file by calling function **sca_util::sca_create_tabular_trace_file**. A trace file may be opened during elaboration or at any time during simulation. Values can only be traced by calling function **sca_util::sca_trace**. A trace file shall be opened before values can be traced to that file, and values shall not be traced to a given trace file if one or more delta cycles have elapsed since opening the file. A VCD trace file shall be closed by calling function **sca_util::sca_close_vcd_trace_file**. A tabular trace file shall be closed by calling function **sca_util::sca_close_tabular_trace_file**. A trace file shall not be closed by these functions before the final delta cycle of the simulation.

An implementation may support other trace file formats by providing alternatives to the functions **sca_util::sca_create_vcd_trace_file**, **sca_util::sca_create_tabular_trace_file**, **sca_util::sca_close_vcd_trace_file**, and **sca_util::sca_close_tabular_trace_file**.

6.1.1. Class definition and function declarations

All names used in the class definitions and function declarations for tracing shall be placed in the namespace **sca_util**.

6.1.1.1. sca_util::sca_trace_mode_base

6.1.1.1.1. Description

The class **sca_util::sca_trace_mode_base** shall define the base class for trace mode manipulators. The manipulators, which shall be derived from this base class, are predefined. An application shall not create an instance and shall not derive from this class.

Instances of derived classes can only be used as argument to the member function **set_mode** of class **sca_util::sca_trace_file** (see 6.1.1.2.5).

An implementation shall at least support the trace mode manipulators as defined in this subclause.

6.1.1.1.2. Class Definition

```
namespace sca_util {

  class sca_trace_mode_base
  {
   public:
    virtual ~sca_trace_mode_base();
  };

  enum sca_ac_fmt { SCA_AC_REAL_IMAG, SCA_AC_MAG_RAD, SCA_AC_DB_DEG };

  class sca_ac_format : public sca_util::sca_trace_mode_base
  {
   public:
    sca_ac_format( sca_util::sca_ac_fmt format = sca_util::SCA_AC_REAL_IMAG );
  };

  enum sca_noise_fmt { SCA_NOISE_SUM, SCA_NOISE_ALL };

  class sca_noise_format : public sca_util::sca_trace_mode_base
  {
   public:
    sca_noise_format( sca_util::sca_noise_fmt format = sca_util::SCA_NOISE_SUM );
  };

  class sca_decimation : public sca_util::sca_trace_mode_base
  {
   public:
```

```
    sca_decimation( unsigned long n );
  };

  class sca_sampling : public sca_util::sca_trace_mode_base
  {
   public:
    sca_sampling( const sca_core::sca_time& tstep,
                  const sca_core::sca_time& toffset = sc_core::SC_ZERO_TIME );
    sca_sampling( double tstep, sc_core::sc_time_unit tstep_unit,
                  double toffset = 0.0, sc_core::sc_time_unit toffset_unit = sc_core::SC_SEC );
  };

  enum sca_multirate_fmt { SCA_INTERPOLATE, SCA_DONT_INTERPOLATE, SCA_HOLD_SAMPLE };

  class sca_multirate : public sca_util::sca_trace_mode_base
  {
   public:
    sca_multirate( sca_util::sca_multirate_fmt format = sca_util::SCA_INTERPOLATE );
  };

} // namespace sca_util
```

6.1.1.1.3. Trace mode classes

If an instance of class **sca_util::sca_ac_format** is passed to the member function **set_mode** of class **sca_util::sca_trace_file**, the format for writing the results of a small-signal frequency-domain or noise simulation shall be set. If **sca_util::SCA_AC_REAL_IMAG** is passed as argument to create an instance of class **sca_util::sca_ac_format**, the results shall be written as real and imaginary part. The signal names shall be extended by *.real* and *.imag*. If **sca_util::SCA_AC_MAG_RAD** is passed as argument to create an instance of class **sca_util::sca_ac_format**, the results shall be written as magnitude value and phase in radian. The signal names shall be extended by *.mag* and *.rad*. If **sca_util::SCA_AC_DB_DEG** is passed as argument to create an instance of class **sca_util::sca_ac_format**, the results shall be written as the magnitude in decibel (dB) and phase in degree. The signal names shall be extended by *.db* and *.deg*. The magnitude (DB) of the signal relative to a reference level of one (1) shall be expressed in decibel according to the formula:

$$DB = 20 \cdot \log_{10}(\textit{magnitude})$$

If an instance of class **sca_util::sca_noise_format** is passed to the member function **set_mode** of class **sca_util::sca_trace_file**, the format for writing the results of a small-signal frequency-domain noise simulation shall be set. If **sca_util::SCA_NOISE_SUM** is passed as argument to create an instance of class **sca_util::sca_noise_format**, the contributions of all noise sources of a small-signal frequency-domain noise simulation shall be summed arithmetically. In this case only the magnitude or real value corresponding to the specified format shall be written. If **sca_util::SCA_NOISE_ALL** is passed as argument to create an instance of class **sca_util::sca_noise_format**, the contributions of all noise sources of a small-signal frequency-domain noise simulation shall be written separately. The name shall be extended by the instance name followed by the corresponding format specifier (e.g. *.db* or *.deg*).

If an instance of class **sca_util::sca_decimation** is passed to the member function **set_mode** of class **sca_util::sca_trace_file**, only every *n*-th line of the results of a time-domain simulation shall be written to the tabular trace file, where *n* is the argument that shall be assigned while creating the passed instance. The default value of *n* is 1. It shall be an error, if *n* is equal to zero.

If an instance of class **sca_util::sca_sampling** is passed to the member function **set_mode** of class **sca_util::sca_trace_file**, the traced signals shall be sampled starting at time offset *toffset* with the sampling period equal to the timestep *tstep*. The arguments *toffset* and *tstep* shall be assigned while creating the passed instance. Only the sampled values shall be written to the tabular trace file. If *tstep* is set to the value **sc_core::SC_ZERO_TIME**, no sampling shall be done. The default value of timestep *tstep* and time offset *toffset* is **sc_core::SC_ZERO_TIME**.

In case multiple signals are written to the same tabular trace file, the trace file shall use the timestep of the signal with the smallest timestep. If these signals have different rates and thus use different timesteps, the class **sca_util::sca_multirate** shall be used to define if and how values are inserted in case there is no sample available at a particular point in time. The instance of class **sca_util::sca_multirate** shall be passed as argument to the member function **set_mode** of

class **sca_util::sca_trace_file**. If **sca_util::SCA_INTERPOLATE** is passed as argument to create an instance of class **sca_util::sca_multirate**, the inserted value shall be the result of interpolation of the last available and the next available value in time. If **sca_util::SCA_DONT_INTERPOLATE** is passed as argument to create an instance of class **sca_util::sca_multirate**, the symbol '*****' shall be inserted. If **sca_util::SCA_HOLD_SAMPLE** is passed as argument to create an instance of class **sca_util::sca_multirate**, the inserted value shall be equal to the last available value of the signal.

6.1.1.2. sca_util::sca_trace_file

6.1.1.2.1. Description

The class **sca_util::sca_trace_file** shall define the abstract base class, from which the classes that provide file handles for VCD, tabular, or other implementation-defined trace file formats are derived. An application shall not construct objects of class **sca_util::sca_trace_file**, but may define pointers and references to this type.

6.1.1.2.2. Class definition

```
namespace sca_util {

  class sca_trace_file
  {
   public:
    void enable();
    void disable();

    void set_mode( const sca_util::sca_trace_mode_base& );

    void reopen( const std::string& name,
                 std::ios_base::openmode mode=std::ios_base::out | std::ios_base::trunc );
    void reopen( std::ostream& outstream );

    // Other members
    implementation-defined
  };

} // namespace sca_util
```

6.1.1.2.3. enable

```
void enable();
```

The member function **enable** shall enable tracing to the trace file. After construction of an object of class **sca_util::sca_trace_file**, tracing shall be enabled by default. Tracing shall start at the simulation time, returned by function **sc_core::sc_time_stamp**, at which the member function **enable** is called.

6.1.1.2.4. disable

```
void disable();
```

The member function **disable** shall disable tracing to the trace file. Values shall be written to the trace file up to the simulation time, returned by function **sc_core::sc_time_stamp**, at which the member function **disable** is called. Tracing shall stop until the member function **enable** is called.

6.1.1.2.5. set_mode

```
void set_mode( const sca_util::sca_trace_mode_base& );
```

The member function **set_mode** shall change the mode of the trace file of the corresponding instance derived from class **sca_util::sca_trace_mode_base**.

6.1.1.2.6. reopen

```
void reopen( const std::string& name,
                 std::ios_base::openmode mode=std::ios_base::out | std::ios_base::trunc );

void reopen( std::ostream& outstream );
```

The member function **reopen** shall write all values up to the simulation time, returned by function **sc_core::sc_time_stamp**, at which the member function is **reopen** is called. If a trace file was opened, it

shall be closed. If arguments *name* and *mode* are provided, a new trace file using the *name* and *mode* shall be opened. If the argument *outstream* is provided, tracing shall be continued using this stream. The member function shall return true, if all actions were successful, and false otherwise. It shall be an error if the trace file cannot be closed and opened.

6.1.1.3. sca_util::sca_create_vcd_trace_file

```
namespace sca_util {

   sca_util::sca_trace_file* sca_create_vcd_trace_file( const char* name );

   sca_util::sca_trace_file* sca_create_vcd_trace_file( std::ostream& os );

} // namespace sca_util
```

The function **sca_util::sca_create_vcd_trace_file** shall create a new object handle of class **sca_util::sca_trace_file**. If a file name is passed as a character string, it shall be opened. If the file name has no extension, the file name shall be constructed by appending the character string ".vcd" to the character string passed as an argument to the function. If a pointer to an object of class std::ostream is passed, the traces shall be written to output stream managed by that object. The function shall return a pointer to the **sca_util::sca_trace_file** handle.

VCD tracing for the classes **sca_tdf::sca_signal_if<T>**, **sca_tdf::sca_in<T>**, **sca_tdf::sca_out<T>**, **sc_core::sc_signal_in_if<T>**, **sc_core::sc_port<sc_core::sc_signal_in_if<T> >**, **sc_core::sc_port<sc_core::sc_signal_inout_if<T> >** and **sca_tdf::sca_trace_variable<T>** shall be supported for at least the data types **T** defined in function **sc_core::sc_trace**, except those using a reference or a pointer to an object. In case the data type uses a reference to an object or a pointer to an object, the tracing functionality is implementation-defined.

6.1.1.4. sca_util::sca_close_vcd_trace_file

```
namespace sca_util {

   void sca_close_vcd_trace_file( sca_util::sca_trace_file* tf );

} // namespace sca_util
```

Depending whether the passed **sca_util::sca_trace_file** argument is associated with a VCD trace file or an output stream, the function **sca_util::sca_close_vcd_trace_file** shall close the VCD trace file or detach the output stream.

6.1.1.5. sca_util::sca_create_tabular_trace_file

```
namespace sca_util {

   sca_util::sca_trace_file* sca_create_tabular_trace_file( const char* name );

   sca_util::sca_trace_file* sca_create_tabular_trace_file( std::ostream& os );

} // namespace sca_util
```

The function **sca_util::sca_create_tabular_trace_file** shall create a new object handle of class **sca_util::sca_trace_file**. If a file name is passed as a character string, it shall be opened. If the file name has no extension, the file name shall be constructed by appending the character string ".dat" to the character string passed as an argument to the function. If a pointer to an object of class std::ostream is passed, the traces shall be written to output stream managed by that object. The function shall return a pointer to the **sca_util::sca_trace_file** handle.

6.1.1.5.1. Format for time-domain simulations

The first line of a tabular trace file shall start with *%time* followed by the signal names separated by one or more spaces. The order of the signal names is defined by the order, in which they are added to the trace file of class **sca_util::sca_trace_file** using **sca_util::sca_trace**. Each of the following lines shall contain the simulation results for one point in time. The first column contains the time as type double in seconds, the following columns the simulation results of the corresponding values, signals, and nodes in the order of the signal names as reflected in the first line. Columns shall be separated by one or more spaces. The

simulation results shall be written to the file using **operator**<< of the value or signals of type double for **sca_eln::sca_node** and **sca_eln::sca_node_ref**. For each calculation time of each value, signal, or node added by **sca_util::sca_trace** to the current **sca_util::sca_trace_file**, a new line shall be written to the file.

6.1.1.5.2. Format for small-signal frequency-domain and noise simulations

The first line of a tabular trace file shall start with *%frequency* followed by the signal names separated by one or more spaces. The order of the signal names is defined by the order, in which they are added to the trace file of class **sca_util::sca_trace_file** using **sca_util::sca_trace**. Dependent on the trace mode settings, the signal names shall be extended by the corresponding format specifier (see 6.1.1.1.3).

Each written line shall contain the simulation result of one frequency point. The first column shall contain the frequency in hertz (Hz) written by the **operator**<< of the type double. The following columns shall contain the simulation results written by the **operator**<< of the type double in the order of the signal names as reflected in the first line. Columns shall be separated by one or more spaces.

6.1.1.6. sca_util::sca_close_tabular_trace_file

```
namespace sca_util {

  void sca_close_tabular_trace_file( sca_util::sca_trace_file* tf );

} // namespace sca_util
```

Depending whether the passed **sca_util::sca_trace_file** argument is associated with a tabular trace file or an output stream, the function **sca_util::sca_close_tabular_trace_file** shall close the tabular trace file or detach the output stream.

6.1.1.7. sca_util::sca_write_comment

```
namespace sca_util {

  void sca_write_comment( sca_util::sca_trace_file* tf, const std::string& comment );

} // namespace sca_util
```

The function **sca_util::sca_write_comment** shall write the string given as the second argument to the trace file given by the first argument, as a comment, at the simulation time, at which the function is called.

In a tabular trace file, the string shall be interpreted as a single comment line, which starts with the symbol '%' and ends with the newline character(s).

NOTE—In a VCD file, comments are enclosed between the tags "$comment" and "$end".

6.1.1.8. sca_util::*sca_traceable_object*†

6.1.1.8.1. Description

The class **sca_util**::*sca_traceable_object*† shall be the base class for all objects, which can be traced.

6.1.1.8.2. Class definition

```
namespace sca_util {

  class sca_traceable_object†
  {
    implementation-defined
  };

} // namespace sca_util
```

6.1.1.8.3. Constraint on usage

The class shall not be instantiated by an application.

6.1.1.9. sca_util::sca_trace

```
namespace sca_util {
```

```
  void sca_trace( sca_util::sca_trace_file* tf,
                  const sca_util::sca_traceable_object†& obj,
                  const std::string& str );

  void sca_trace( sca_util::sca_trace_file* tf,
                  const sca_util::sca_traceable_object†* obj,
                  const std::string& str );

  template<class T>
  void sca_trace( sca_util::sca_trace_file* tf,
                  const sc_core::sc_signal_in_if<T>& value,
                  const std::string& str );

  template<class T>
  void sca_trace( sca_util::sca_trace_file* tf,
                  const sc_core::sc_port<sc_core::sc_signal_in_if<T> >& value,
                  const std::string& str );

  template<class T>
  void sca_trace( sca_util::sca_trace_file* tf,
                  const sc_core::sc_port<sc_core::sc_signal_inout_if<T> >& value,
                  const std::string& str );

} // namespace sca_util
```

The function **sca_util::sca_trace** shall trace the value passed as the second argument to the trace file passed as the first argument, using the string passed as the third argument to identify the value in the trace file.

6.2. Data types and constants

All native C++ types are supported within a SystemC AMS application. The AMS class library shall provide additional data type classes, which shall be used as data containers to setup the equation systems and for communication as part of the predefined models of computation.

The following data type classes shall be supported:

- Complex numbers, which are objects of class **sca_util::sca_complex**. A complex number shall be represented by a real and imaginary part of type double.
- Matrix types, which are objects of class **sca_util::sca_matrix**. A matrix type shall represent a two dimensional data container of an arbitrary type.
- Vector types, which are objects of class **sca_util::sca_vector**. A vector type shall represent a one-dimensional data container of an arbitrary type.

The following symbolic constants shall be supported:

- **sca_util::SCA_INFINITY**, representing infinity.
- **sca_util::SCA_COMPLEX_J**, representing the imaginary number *j*.

NOTE—An application may create data type objects using the classes defined in this clause, as long as the data type is supported within the predefined model of computation and its associated classes. The set of member functions for these data type classes is restricted, as the primary use of these class definitions is to act as data container only.

6.2.1. Class definition and function declarations

All names used in the class definitions and function declarations for data types shall be placed in the namespace **sca_util**.

6.2.1.1. sca_util::sca_complex

6.2.1.1.1. Description

The type **sca_util::sca_complex** shall provide a type for complex numbers.

6.2.1.1.2. Class definition

```
namespace sca_util {
```



```
  typedef std::complex<double> sca_complex;

} // namespace sca_util
```

6.2.1.2. sca_util::sca_matrix

6.2.1.2.1. Description

The class **sca_util::sca_matrix** shall provide a type for handling two dimensional matrices of an arbitrary type.

6.2.1.2.2. Class definition

```
namespace sca_util {

  template<class T>
  class sca_matrix
  {
   public:
    sca_matrix();
    sca_matrix( unsigned long n_rows_, unsigned long n_cols_ );
    sca_matrix( const sca_util::sca_matrix<T>& matrix );

    void resize( unsigned long n_rows_, unsigned long n_cols_ );
    void set_auto_resizable();
    void unset_auto_resizable();
    bool is_auto_resizable() const;
    unsigned long n_rows() const;
    unsigned long n_cols() const;

    T& operator() ( unsigned long row, unsigned long col );
    const T& operator() ( unsigned long row, unsigned long col ) const;

    const sca_util::sca_matrix<T>& operator= (
                                  const sca_util::sca_matrix<T>& matrix );
    const sca_util::sca_matrix<T>& operator= (
                                  sca_core::sca_assign_from_proxy†<sca_util::sca_matrix<T> >& );


    const std::string to_string() const;
    void print( std::ostream& = std::cout ) const;
  };

  template<class T>
  std::ostream& operator<< ( std::ostream&, const sca_util::sca_matrix<T>& );

} // namespace sca_util
```

6.2.1.2.3. Template parameter T

The argument passed as template parameter **T** shall be either a C++ type for which the predefined semantics for assignment are adequate (for example, a fundamental type or a pointer), or a type **T** that obeys each of the following rules:

a. The following stream operator shall be defined and should copy the state of the object given as the second argument to the stream given as the first argument. The way in which the state information is formatted is undefined by this standard.

```
std::ostream& operator<< ( std::ostream&, const T& );
```

b. If the default assignment semantics are inadequate (in the sense given in this subclause), the following assignment operator should be defined for the type **T**. In either case (default assignment or explicit operator), the semantics of assignment should be sufficient to assign the state of an object of type **T** such that the value of the left operand is indistinguishable from the value of the right operand.

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

c. If any constructor for type **T** exists, a default constructor for type **T** shall be defined.

6.2.1.2.4. Constructors

```
sca_matrix();

sca_matrix( unsigned long n_rows_, unsigned long n_cols_ );
```

```
sca_matrix( const sca_util::sca_matrix<T>& matrix );
```

The constructor **sca_matrix()** shall construct a matrix of size zero. The constructor **sca_matrix**(unsigned long *n_rows_*, unsigned long *n_cols_*) shall construct a matrix of *n_rows_* rows and *n_cols_* columns. For creating the elements, the default constructor of the corresponding type shall be used. For matrices of type short, unsigned short, long, unsigned long, int, unsigned int, **sc_dt::int64**, and **sc_dt::uint64**, the elements shall be initialized with zero. Matrices of type float, double, and **sca_util::sca_complex** shall be initialized with 0.0. The constructor **sca_matrix**(const **sca_util::sca_matrix**<T>& *matrix*) shall construct a matrix with the same dimension as matrix *matrix* and shall copy all elements from matrix *matrix* to the new matrix including the automatic resizable mode state. For copying the elements, their copy constructor shall be used.

6.2.1.2.5. resize

```
void resize( unsigned long n_rows_, unsigned long n_cols_ );
```

The member function **resize** shall resize the matrix to *n_rows_* rows and *n_cols_* columns. After resizing, the values of still existing elements shall remain unchanged. New elements shall be created by the default constructor. New element for matrices of type short, unsigned short, long, unsigned long, int, unsigned int, **sc_dt::int64** and **sc_dt::uint64** shall be initialized with zero. New elements for matrices of type float, double, and **sca_util::sca_complex** shall be initialized with 0.0. Elements, which are no longer required, shall be destroyed by their destructor.

6.2.1.2.6. set_auto_resizable

```
void set_auto_resizable();
```

The member function **set_auto_resizable** sets the matrix in the automatic resizable mode. In this mode the matrix shall be resized automatically using the member function **resize**, if an element outside of the current dimensions is accessed. In this case the matrix shall be resized to the minimum dimension for that the accessed element exists. After construction, a matrix shall be set to be automatically resizable.

6.2.1.2.7. unset_auto_resizable

```
void unset_auto_resizable();
```

The member function **unset_auto_resizable** deactivates the automatic resizable mode. After calling the member function **unset_auto_resizable**, it shall be an error to access an element outside of the current matrix dimensions.

6.2.1.2.8. is_auto_resizable

```
bool is_auto_resizable() const;
```

The member function **is_auto_resizable** shall return true, if the matrix is automatic resizable, and false otherwise.

6.2.1.2.9. n_rows

```
unsigned long n_rows() const;
```

The member function **n_rows** shall return the number of rows in the matrix.

6.2.1.2.10. n_cols

```
unsigned long n_cols() const;
```

The member function **n_cols** shall return the number of columns in the matrix.

6.2.1.2.11. operator()

```
T& operator() ( unsigned long row, unsigned long col )
```

The **operator()** shall assign a value to the matrix element at row *row* and column *col*.


```
const T& operator() ( unsigned long row, unsigned long col ) const;
```

The **operator()** shall return a reference to the matrix element at row *row* and column *col*. The behavior for accessing a non-existing element shall depend on the mode returned by the member function **is_auto_resizable**.

The returned reference is only guaranteed to be valid until the end of the full expression containing the call.

6.2.1.2.12. operator=

```
const sca_util::sca_matrix<T>& operator= ( const sca_util::sca_matrix<T>& matrix );
```

The **operator=** shall copy matrix *matrix*. The dimensions, the automatic resizable mode state, and all elements shall be copied. For copying the elements, the copy constructor shall be used.

```
const sca_util::sca_matrix<T>& operator= (
                                    sca_core::sca_assign_from_proxy†<sca_util::sca_matrix<T> >& );
```

The **operator=** shall copy the values the values made available by the object of class **sca_core::***sca_assign_from_proxy*† to the object of class **sca_util::sca_matrix**.

6.2.1.2.13. to_string

```
const std::string to_string() const;
```

The member function **to_string** shall perform the conversion of the current matrix to an object of class std::string. Conversion shall be done by calling **operator**<< (std::ostream&, const T&). The values in the columns shall be separated by one or more spaces and the rows shall be separated by a new line.

6.2.1.2.14. print

```
void print( std::ostream& = std::cout ) const;
```

The member function **print** shall print the current matrix to the stream passed as an argument by calling **operator**<< (std::ostream&, const T&). The values in the columns shall be separated by one or more spaces and the rows shall be separated by a new line.

6.2.1.2.15. operator<<

```
template<class T>
std::ostream& operator<< ( std::ostream&, const sca_util::sca_matrix<T>& );
```

The **operator**<< shall write the current matrix to the stream by calling **operator**<< of the element type. The values in the columns shall be separated by one or more spaces and the rows shall be separated by a new line.

6.2.1.3. sca_util::sca_vector

6.2.1.3.1. Description

The class **sca_util::sca_vector** shall provide a type for handling vectors of an arbitrary type.

6.2.1.3.2. Class definition

```
namespace sca_util {

  template<class T>
  class sca_vector
  {
   public:
    sca_vector();
    explicit sca_vector( unsigned long len );
    sca_vector( const sca_util::sca_vector<T>& vector );

    void resize( unsigned long len );
    void set_auto_resizable();
    void unset_auto_resizable();
    bool is_auto_resizable() const;
    unsigned long length() const;
```

```
    T& operator() ( unsigned long pos ) const;
    const T& operator() ( unsigned long pos ) const;

    sca_util::sca_vector<T>& operator= ( const sca_util::sca_vector<T>& vector );
    sca_util::sca_vector<T>& operator= ( sca_core::sca_assign_from_proxy†<sca_util::sca_vector<T> >& );

    const std::string to_string() const;
    void print( std::ostream& = std::cout ) const;
  };

  template<class T>
  std::ostream& operator<< ( std::ostream&, const sca_util::sca_vector<T>& );

} // namespace sca_util
```

6.2.1.3.3. Template parameter T

The argument passed as template parameter **T** shall be either a C++ type for which the predefined semantics for assignment are adequate (for example, a fundamental type or a pointer), or a type **T** that obeys each of the following rules:

a. The following stream operator shall be defined and should copy the state of the object given as the second argument to the stream given as the first argument. The way in which the state information is formatted is undefined by this standard.

```
std::ostream& operator<< ( std::ostream&, const T& );
```

b. If the default assignment semantics are inadequate (in the sense given in this subclause), the following assignment operator should be defined for the type **T**. In either case (default assignment or explicit operator), the semantics of assignment should be sufficient to assign the state of an object of type **T** such that the value of the left operand is indistinguishable from the value of the right operand.

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

c. If any constructor for type **T** exists, a default constructor for type **T** shall be defined.

6.2.1.3.4. Constructors

```
sca_vector();

explicit sca_vector( unsigned long len );

sca_vector( const sca_util::sca_vector<T>& vector );
```

The constructor **sca_vector()** shall construct a vector of size zero. The constructor **sca_vector**(unsigned long *len*) shall construct a vector with a length *len*. For creating the elements, the default constructor of the corresponding type shall be used. For vectors of type short, unsigned short, long, unsigned long, int, unsigned int, **sc_dt::int64**, and **sc_dt::uint64**, the elements shall be initialized with zero. Vectors of type float, double, and **sca_util::sca_complex** shall be initialized with 0.0. The constructor **sca_vector**(const **sca_util::sca_vector**<T>& *vector*) shall construct a vector with the same dimension as vector *vector*, and shall copy all elements from vector *vector* to the new vector including the automatic resizable mode state. For copying the elements, their copy constructor shall be used.

6.2.1.3.5. resize

```
void resize( unsigned long len );
```

The member function **resize** shall resize the vector to the length *len*. After resizing, the values of still existing elements shall remain unchanged. New elements shall be created by the default constructor. New elements for vectors of type short, unsigned short, long, unsigned long, int, unsigned int, **sc_dt::int64**, and **sc_dt::uint64** shall be initialized with zero. New elements for vectors of type float, double, and **sca_util::sca_complex** shall be initialized with 0.0. Elements which are no longer required shall be destroyed by their destructor.

6.2.1.3.6. set_auto_resizable

```
void set_auto_resizable();
```

The member function **set_auto_resizable** sets the vector in the automatic resizable mode. In this mode the vector shall be resized automatically using the member function **resize**, if an element outside of the

current dimensions is accessed. In this case the vector shall be resized to the minimum dimension for that the accessed element exists. After construction a vector shall be set to be automatically resizable.

6.2.1.3.7. unset_auto_resizable

```
void unset_auto_resizable();
```

The member function **unset_auto_resizable** deactivates the automatic resizable mode. After calling the member function **unset_auto_resizable**, it shall be an error to access an element outside of the current vector dimensions.

6.2.1.3.8. is_auto_resizable

```
bool is_auto_resizable() const;
```

The member function **is_auto_resizable** shall return true, if the vector is automatic resizable, and false otherwise.

6.2.1.3.9. length

```
unsigned long length() const;
```

The member function **length** returns the length of the vector.

6.2.1.3.10. operator()

```
T& operator() ( unsigned long pos );
```

The **operator()** shall assign a value to the vector element at position *pos*.

```
const T& operator() ( unsigned long pos ) const;
```

The **operator()** shall return a reference to the vector element at position *pos*. The behavior for accessing a non-existing element shall depend on the mode returned by the member function **set_auto_resizable**.

The returned reference is only guaranteed to be valid until the end of the full expression containing the call.

6.2.1.3.11. operator=

```
sca_util::sca_vector<T>& operator= ( const sca_util::sca_vector<T>& vector );
```

The **operator=** shall copy vector *vector*. The dimensions, the automatic resizable mode and all elements shall be copied. For copying the elements, the copy constructor shall be used.

```
sca_util::sca_vector<T>& operator= ( sca_core::sca_assign_from_proxy†<sca_util::sca_vector<T> >& );
```

The **operator=** shall copy the values the values made available by the object of class **sca_core::***sca_assign_from_proxy*† to the object of class **sca_util::sca_vector**.

6.2.1.3.12. to_string

```
const std::string to_string() const;
```

The member function **to_string** shall perform the conversion of the current vector to an object of class std::string. Conversion shall be done by calling **operator<<** (std::ostream&, const T&). The values in the vector shall be separated by one or more spaces.

6.2.1.3.13. print

```
void print( std::ostream& = std::cout ) const;
```

The member function **print** shall print the current vector to the stream passed as an argument by calling **operator<<** (std::ostream&, const T&). The values in the vector shall be separated by one or more spaces.

6.2.1.3.14. operator<<

```
template<class T>
```



```
std::ostream& operator<< ( std::ostream&, const sca_util::sca_vector<T>& );
```

The **operator<<** shall write the current vector to the stream by calling **operator<<** of the element type. The values in the vector shall be separated by one or more spaces.

6.2.1.4. sca_util::sca_create_vector

```
namespace sca_util {

  template<class T>
  sca_util::sca_vector<T> sca_create_vector( const T& a0 );

  template<class T>
  sca_util::sca_vector<T> sca_create_vector( const T& a0, const T& a1 );

  ...

  template<class T>
  sca_util::sca_vector<T> sca_create_vector( const T& a0, const T& a1,  ... const T& a15 );

} // namespace sca_util
```

6.2.1.4.1. Description

The functions **sca_util::sca_create_vector** shall create an object of class **sca_util::sca_vector** of the size according to the number of arguments. The vector elements shall be initialized by the values of the arguments. The first argument shall initialize the first vector element, the second argument shall initialize the second vector element, and so on till the last argument. An implementation shall provide at least functions to initialize vectors of 1 up to 16 arguments.

6.2.2. Definition of constants

6.2.2.1. sca_util::SCA_INFINITY

6.2.2.1.1. Description

The constant **sca_util::SCA_INFINITY** shall be interpreted as infinity, if representable as a dedicated data type, which is implementation-defined, else as if it were a floating constant that is too large for the range of the data type to which the constant is assigned to.

6.2.2.1.2. Definition

```
namespace sca_util {

static const double SCA_INFINITY = implementation-defined

} // namespace sca_util
```

6.2.2.2. sca_util::SCA_COMPLEX_J

6.2.2.2.1. Description

The constant **sca_util::SCA_COMPLEX_J** shall represent the imaginary number *j*. This constant shall be defined using class **sca_util::sca_complex**, where value 0.0 is passed as first argument of type double and value 1.0 is passed as second argument of type double.

6.2.2.2.2. Definition

```
namespace sca_util {

static const sca_util::sca_complex SCA_COMPLEX_J = sca_util::sca_complex( 0.0, 1.0 );

} // namespace sca_util
```

6.3. Reporting information

An implementation shall provide information about the clustering and the solvers. The content of the information shall be implementation-defined. By default, all information shall be reported. The information

shall be reported by the member function **report** of the class **sc_core::sc_report_handler** with the severity **SC_INFO** and the message type "**SCA_INFORMATION**".

6.3.1. Class definition and function declarations

All names used in the class definitions and function declarations for reporting information shall be placed in the namespace **sca_util**.

6.3.1.1. sca_util::*sca_information_mask*†

The class **sca_util**::*sca_information_mask*† shall provide a mechanism to mask the reporting of information.

6.3.1.1.1. Class definition

```
namespace sca_util {

  class sca_information_mask†
  {
   public:
    sca_util::sca_information_mask† operator| ( const sca_information_mask†& ) const;

    implementation-defined
  };

} // namespace sca_util
```

6.3.1.1.1.1. operator|

```
sca_util::sca_information_mask† operator| ( const sca_information_mask†& ) const;
```

The **operator|** shall add the provided element to the mask.

6.3.1.2. sca_util::sca_information_on

```
namespace sca_util {

  void sca_information_on();
  void sca_information_on( sca_util::sca_information_mask† mask );

} // namespace sca_util
```

The function **sca_util::sca_information_on** without argument shall enable reporting of all information. The function with the *mask* argument shall enable reporting of all masked information. Reporting of other information shall be disabled.

6.3.1.3. sca_util::sca_information_off

```
namespace sca_util {

  void sca_information_off();
  void sca_information_off( sca_util::sca_information_mask† mask );

} // namespace sca_util
```

The function **sca_util::sca_information_off** without argument shall disable reporting of all information. The function with the *mask* argument shall disable reporting of all masked information. Reporting of other information shall be enabled.

6.3.2. Mask definitions

```
namespace sca_util {

  namespace sca_info {

    extern const sca_util::sca_information_mask† sca_module;
    extern const sca_util::sca_information_mask† sca_tdf_solver;
    extern const sca_util::sca_information_mask† sca_lsf_solver;
    extern const sca_util::sca_information_mask† sca_eln_solver;

  } // namespace sca_info
```

```
} // namespace sca_util
```

6.3.2.1. sca_util::sca_info::sca_module

The global constant variable **sca_util::sca_info::sca_module** shall define a mask that can be used to enable or disable reporting of AMS module information.

6.3.2.2. sca_util::sca_info::sca_tdf_solver

The global constant variable **sca_util::sca_info::sca_tdf_solver** shall define a mask that can be used to enable or disable reporting of TDF solver information.

6.3.2.3. sca_util::sca_info::sca_lsf_solver

The global constant variable **sca_util::sca_info::sca_lsf_solver** shall define a mask that can be used to enable or disable reporting of LSF solver information.

6.3.2.4. sca_util::sca_info::sca_eln_solver

The global constant variable **sca_util::sca_info::sca_eln_solver** shall define a mask that can be used to enable or disable reporting of ELN solver information.

6.4. Implementation information

6.4.1. Function declarations

All names used in the function declarations to report information on the implementation shall be placed in the namespace **sca_core**.

```
namespace sca_core {

  const char* sca_copyright();
  const char* sca_version();
  const char* sca_release();

} // namespace sca_core
```

6.4.1.1. sca_core::sca_copyright

```
const char* sca_core::sca_copyright();
```

The function **sca_core::sca_copyright** shall return an implementation-defined string. The intent is that this string should contain a legal copyright notice, which an application may print to the console window or to a log file.

6.4.1.2. sca_core::sca_version

```
const char* sca_core::sca_version();
```

The function **sca_core::sca_version** shall return an implementation-defined string. The intent is that this string should contain information concerning the version of the class library implementation of the SystemC AMS extensions, which an application may print to the console window or to a log file.

6.4.1.3. sca_core::sca_release

```
const char* sca_core::sca_release();
```

The function **sca_core::sca_release** shall return an implementation-defined string of the following form:

<*major#*>.<*minor#*>.<*patch*>-<*originator*>

where <*major#*> represents the major release number, <*minor#*> represents the minor release number, <*patch*> represents the patch level, and <*originator*> represents the originator of the implementation of the SystemC AMS extensions. The intent is that this string should be machine-readable by any application

that has an interest in checking the version or release number of the implementation of the SystemC AMS extensions.

The character set for each of these four fields shall be as follows:

a. The lower-case letters a-z

b. The upper-case letters A-Z

c. The decimal digits 0-9

d. The underscore character _

This page is intentionally left blank.

Annex A. Introduction to the SystemC AMS extensions

(Informative)

This clause is informative and is intended to aid the reader in the understanding of the structure and intent of the SystemC AMS extensions. The SystemC AMS extensions broaden the scope of SystemC towards design of analog/mixed-signal and signal processing systems at functional and architecture level considering the use cases executable specification, architecture exploration, integration verification, and virtual prototyping.

For this purpose, the extensions provide new models of computation:

- Timed Data Flow (TDF) as described in Subclause 4.1 enables the modeling and efficient simulation of signal processing algorithms and communication systems at functional and architecture level.
- Linear Signal Flow (LSF) as described in Subclause 4.2 enables the modeling of control systems or filter structures in continuous time using macro models connected by directed, real-valued signals.
- Electrical Linear Networks (ELN) as described in Subclause 4.3 enables the modeling of electrical loads, protection circuits, and buses at high frequencies using macro models for describing continuous-time relations between voltages and currents.

Furthermore, the extensions introduce small-signal frequency-domain analysis as described in Subclause 5.2 as a new kind of analysis that complements the time-domain analysis as available in SystemC, data types for computing with complex values, vectors, and matrices, and means for tracing AMS signals.

The AMS extensions are built on top of the SystemC standard. The architecture of the extensions is shown in Figure A.1. The extensions define new language constructs identified by the prefix **sca_**. They are declared in dedicated namespaces **sca_tdf**, **sca_lsf**, and **sca_eln** according to the underlying model of computation. By using namespaces, similar primitives as in SystemC are defined to denote ports, interfaces, signals, and modules. For example, a timed data flow input port is an object of class **sca_tdf::sca_in**.

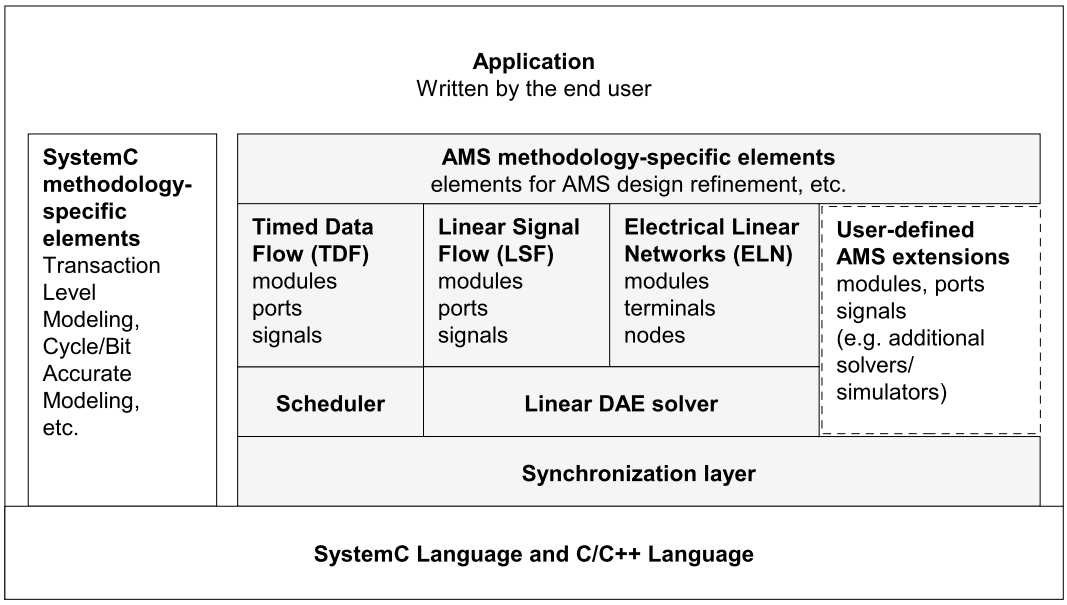

Figure A.1. AMS Extensions for the SystemC Language Standard

The TDF model of computation is suitable to describe the functional behavior of the signal processing parts of a system. TDF models consist of TDF modules derived from class **sca_tdf::sca_module** that are connected via TDF channels of class **sca_tdf::sca_signal** using TDF ports derived from class **sca_tdf::sca_in** and **sca_tdf::sca_out**. The behavior of a TDF module is specified by predefined member functions **set_attributes**, **initialize**, and **processing**:

- The member function **set_attributes** is used to specify attributes such as rates, delay, or timesteps of TDF ports and modules.

- The member function **initialize** is used to specify initial conditions. It is executed once when the simulation starts.
- The member function **processing** describes time-domain behavior of the module. It is executed at each activation of the TDF module.

TDF modules form contiguous structures called a TDF cluster. Clusters must not have cycles without delays, and each TDF signal must have one source. A cluster is activated in discrete time steps. At each timestep, the member function **processing** of each TDF module is executed in the data flow's direction, considering multiple data rates.

It is expected that there is at least one definition of the timestep value and, in the case of cycles, one definition of a delay value per cycle. TDF ports are single-rate by default. It is the task of the elaboration phase to compute and propagate consistent values for the timesteps to all TDF ports and modules. Before simulation, the scheduler determines a schedule that defines activation order of the TDF modules taking into account the rates, delays, and timesteps. During simulation, the member functions **processing** are executed at discrete time steps. Example A.1 shows a TDF model of a mixer, which member function **processing** will be executed with a timestep of 1µs.

```
SCA_TDF_MODULE(mixer) // TDF primitive module definition
{
  sca_tdf::sca_in<double>  rf_in, lo_in; // TDF input ports
  sca_tdf::sca_out<double> if_out;       // TDF output port

  void set_attributes()
  {
    set_timestep(1.0, SC_US);            // time between activations
  }

  void processing()                      // executed at each activation
  {
    if_out.write( rf_in.read() * lo_in.read() );
  }

  SCA_CTOR(mixer) {}

};
```

Example A.1. TDF module of a mixer

In addition to the pure algorithmic or procedural description of the member function **processing**, different kind of transfer functions can be embedded in TDF modules. Example A.2 gives the TDF model of a gain controlled low pass filter by instantiating a class that computes a continuous-time Laplace transfer function (LTF). The coefficients are stored in a vector of the class **sca_util::sca_vector** and are set in the member function **initialize**. The transfer function is computed in the member function **processing** by the LTF object at discrete time points using fixed-size time steps.

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

  sca_tdf::sc_in<double>       gain;          // converter port for SystemC input
  sca_tdf::sca_ltf_nd          ltf;           // computes transfer function
  sca_util::sca_vector<double> num, den;      // coefficients

  void initialize()
  {
    num(0) = 1.0;
    den(0) = 1.0;
    den(1) = 1.0 /( 2.0 * M_PI * 1.0e4 ); // M_PI = 3.1415...
  }

  void processing()
  {
    out.write( ltf( num, den, in, gain.read() ) );
  }

  SCA_CTOR(lp_filter_tdf) {}

};
```

Example A.2. Continuous-time Laplace transfer function embedded in a TDF module

TDF modules can be instantiated and connected like SystemC modules. However, TDF modules are primitive modules and no other modules or SystemC processes may be instantiated inside. To set up a hierarchical structure, TDF modules must be instantiated in SystemC modules (**sc_core::sc_module**).

Predefined converter ports (**sca_tdf::sc_out** or **sca_tdf::sc_in**) can establish a connection to a SystemC channel, e.g., **sc_core::sc_signal**<T>, to read or write values during the first delta cycle of the current SystemC timestep. Example A.3 illustrates the usage of such a converter port in a TDF module to model a simple A/D converter with an output port, to which a SystemC channel can be bound.

```
SCA_TDF_MODULE(ad_converter) // Very simple AD converter
{
  sca_tdf::sca_in<double>             in_tdf; // TDF port
  sca_tdf::sc_out<sc_dt::sc_int<12> > out_de; // converter port to discrete-event domain

  void processing()
  {
    out_de.write( static_cast<sc_dt::sc_int<12> >( in_tdf.read() ) );
  }

  SCA_CTOR(ad_converter) { }

};
```

Example A.3. TDF model of a simple A/D converter

LSF models are specified by instantiating a structure of signal flow primitives such as adders, integrators, differentiators, or transfer functions. These primitives are part of the language definition. The primitives are connected via signals of the class **sca_lsf::sca_signal**. To access LSF signals from the TDF or discrete event domain, converter modules have to be instantiated. The instantiation of the LSF primitive modules can be done in a regular **SC_MODULE** using the standard SystemC rules.

```
SC_MODULE(lp_filter_lsf)  // Hierarchical models use SC_MODULE class
{
  sca_tdf::sca_in<double>  in;        // Input from TDF MoC
  sca_tdf::sca_out<double> out;       // Output to TDF MoC

  sca_lsf::sca_sub* sub1;             // Subtractor
  sca_lsf::sca_dot* dot1;             // Differentiator
  sca_lsf::sca_tdf_source* tdf2lsf1;  // converter TDF -> LSF
  sca_lsf::sca_tdf_sink*   lsf2tdf1;  // converter LSF -> TDF

  sca_lsf::sca_signal in_lsf, sig, out_lsf;       // LSF signals

  lp_filter_lsf(sc_module_name, double fc=1.0e3)  // Constructor with parameters
  {
    tdf2lsf1 = new sca_lsf::sca_tdf_source("tdf2lsf1");  // TDF->LSF converter
    tdf2lsf1->inp(in);                // Port to signal binding like in SystemC
    tdf2lsf1->y(in_lsf);

    sub1 = new sca_lsf::sca_sub("sub1");
    sub1->x1(in_lsf);
    sub1->x2(sig);
    sub1->y(out_lsf);

    dot1 = new sca_lsf::sca_dot("dot1",1.0/(2.0*M_PI*fc));  // M_PI=3.1415...
    dot1->x(out_lsf);
    dot1->y(sig);

    lsf2tdf1 = new sca_lsf::sca_tdf_sink("lsf2tdf1");       // LSF->TDF converter
    lsf2tdf1->x(out_lsf);
    lsf2tdf1->outp(out);
  }

};
```

Example A.4. LSF model of a low pass filter structure with converter modules

ELN models are specified by instantiating predefined network primitives such as resistors or capacitors. As with LSF, the ELN primitive modules can be instantiated in a regular **SC_MODULE**. To access the voltages or currents in the network, converter modules have to be used. The SystemC AMS extensions provide converter modules that translate these voltages and currents to double-precision floating point values in timed data flow and discrete-event models and vice versa. Example A.5 gives the ELN model of a low-pass filter implemented with one resistor primitive and one capacitor primitive. Since it is intended to use the

model in a TDF context, additional converter primitives are used for converting TDF data sample values to voltages and vice-versa.

```
SC_MODULE(lp_filter_eln)
{
  sca_tdf::sca_in<double>  in;             // Input from TDF MoC
  sca_tdf::sca_out<double> out;            // Output to TDF MoC

  sca_eln::sca_node     in_node, out_node; // node declarations
  sca_eln::sca_node_ref gnd;               // reference node

  sca_eln::sca_r           *r1;            // resistor
  sca_eln::sca_c           *c1;            // capacitor
  sca_eln::sca_tdf_vsource *v_in;          // converter TDF -> voltage
  sca_eln::sca_tdf_sink    *v_out;         // converter voltage -> TDF

  SC_CTOR(lp_filter_eln)
  {
    v_in = new sca_eln::sca_tdf_vsource("v_in", 1.0); // scale factor 1.0
    v_in->inp(in);                                    // TDF input
    v_in->p(in_node);                                 // is converted to voltage
    v_in->n(gnd);

    r1 = new sca_eln::sca_r("r1", 10e3);              // 10 kOhm resistor
    r1->p(in_node);
    r1->n(out_node);

    c1 = new sca_eln::sca_c("c1", 100e-6);            // 100 uF capacitor
    c1->p(out_node);
    c1->n(gnd);

    v_out = new sca_eln::sca_tdf_sink("v_out", 1.0);  // scale factor 1.0
    v_out->p(out_node);                               // filter output as voltage
    v_out->n(gnd);
    v_out->outp(out);                                 // here converted to a TDF signal
  }

};
```

Example A.5. ELN model of a low pass filter structure with converter modules

The TDF modules given in Example A.1 and Example A.2 can be instantiated and connected to form a hierarchical structure together with other SystemC modules. The TDF modules are connected as in SystemC with TDF signals of class **sca_tdf::sca_signal** and SystemC signals as shown in Example A.6.

```
SC_MODULE(frontend) // SC_MODULE used for hierarchical structure
{
  sca_tdf::sca_in<double> rf, loc_osc;          // use TDF ports to connect with
  sca_tdf::sca_out<double> if_out;              // TDF ports/signals in hierarchy
  sc_core::sc_in<sc_dt::sc_bv<3> > ctrl_config; // SystemC input agc_ctrl configuration

  sca_tdf::sca_signal<double> if_sig;           // TDF internal signal
  sc_core::sc_signal<double> ctrl_gain;         // SystemC internal signal

  mixer* mixer1;
  lp_filter_tdf* lpf1;
  agc_ctrl* ctrl1;

  SC_CTOR(frontend) {
    mixer1 = new mixer("mixer1");
    mixer1->rf_in(rf);
    mixer1->lo_in(loc_osc);
    mixer1->if_out(if_sig);

    lpf1 = new lp_filter_tdf("lpf1");
    lpf1->in(if_sig);
    lpf1->out(if_out);
    lpf1->gain(ctrl_gain);

    ctrl1 = new agc_ctrl("ctrl1");              // SystemC module
    ctrl1->out(ctrl_gain);
    ctrl1->config(ctrl_config);
  }

};
```

Example A.6. Structural description including TDF and SystemC modules.

Using the frontend module given in Example A.6 and other modules for a demodulator and testbench (implementation not shown here), the structure of the top-level model in **sc_main** is shown in Example A.7.


```
#include <systemc-ams>          // SystemC AMS extensions header

#include "frontend.h"    // definitions for subsystems
#include "demodulator.h"
#include "testbench.h"

int sc_main(int argc, char* argv[]) {

  sca_tdf::sca_signal<double> rf, loc_osc, d_in;
  sca_tdf::sca_signal<bool> symbol;
  sc_core::sc_signal< sc_dt::sc_bv<3> > ctrl_config;

  frontend fe1("fe1");
  fe1.loc_osc(loc_osc);
  fe1.ctrl_config(ctrl_config);
  fe1.rf(rf);
  fe1.if_out(d_in);

  demodulator demod1("demod1");
  demod1.in(d_in);
  demod1.out(symbol);

  testbench tb1("tb1");
  tb1.rf(rf);
  tb1.loc_osc(loc_osc);
  tb1.ctrl(ctrl_config);
  tb1.symbol(symbol);

  // tracing ...
  sca_util::sca_trace_file *tfp =
    sca_util::sca_create_tabular_trace_file("systemc_ams_example.dat");

  sca_util::sca_trace(tfp, rf, "rf");
  sca_util::sca_trace(tfp, loc_osc, "loc_osc");
  sca_util::sca_trace(tfp, fe1.if_sig, "fe1.if_sig");
  sca_util::sca_trace(tfp, ctrl_config, "ctrl_config");
  sca_util::sca_trace(tfp, fe1.ctrl_gain, "fe1.ctrl_gain");
  sca_util::sca_trace(tfp, d_in, "d_in");
  sca_util::sca_trace(tfp, symbol, "symbol");

  sc_core::sc_start(51.2, sc_core::SC_MS);

  sca_util::sca_close_tabular_trace_file(tfp);

  return 0;
};
```

Example A.7. Top-level model in sc_main

Besides means for modeling timed data flow, linear signal flow, and electrical linear networks, the SystemC AMS extensions provide additional means for tracing AMS signals similar to what SystemC already offers for discrete-event signals. Besides the VCD format, which is best suited for tracing transient digital signals, support for a tabular format is added, which facilitates the interface for further processing of the results with mathematically oriented tools.

The SystemC AMS extensions provide support for small-signal frequency-domain analyses for TDF, LSF, and ELN descriptions. For small-signal frequency-domain analysis, the function **sca_ac_analysis::sca_ac_start** (Subclause 5.2.1.3.1) is called with arguments that specify a set of frequencies, for which the transfer function is to be computed. Similarly, a small-signal frequency-domain noise analysis can be invoked using the function **sca_ac_analysis::sca_ac_noise_start** (Subclause 5.2.1.3.2). For LSF models and ELN models, the transfer function is part of the primitives description. It just has to be transformed from the internal time-domain representation to the frequency-domain representation. For TDF modules, the user can specify complex-valued transfer functions between inputs and outputs by overloading the function **sca_tdf::sca_module::ac_processing**. The amplitude at a frequency **sca_ac_analysis::sca_ac_f** can be read from the input ports, be processed using functions specified in Subclause 5.2, and propagated to the output ports. The results of the small-signal frequency-domain analyses can be traced to a tabular trace file (see 6.1.1.5).

This page is intentionally left blank.

Annex B. Glossary

(Informative)

The technical terms and phrases as defined in IEEE Std 1666-2005 (SystemC Language Reference Manual) also apply for the AMS extensions. In addition, this glossary contains brief, informal descriptions for a number of terms and phrases used in this standard. Where appropriate, the complete, formal definition of each term or phrase is given in the main body of the standard.

Each glossary entry contains either the clause number of the definition in the main body of the standard or an indication that a similar term is defined in the SystemC Language Reference Manual (IEEE Std 1666-2005).

B.1. application

A C++ program, written by an end user, that uses the SystemC class library, or the AMS extensions that is, uses classes, calls functions, uses macros, and so forth. 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. (See 2.1.2.) (See SystemC term.)

B.2. cluster

A cluster is a set of connected modules sharing the same model of computation.

B.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.

B.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*.

B.5. electrical linear networks, ELN

A model of computation that uses the electrical linear networks formalism for calculations. (See 4.3.)

B.6. frequency-domain processing

Frequency-domain processing can be embedded in timed data flow descriptions for analysis of small-signal frequency-domain behavior. The small-signal frequency-domain processing member function can be either the member function **sca_tdf::sca_module::ac_processing** or an application-defined member function, which shall be registered using the member function **sca_tdf::sca_module::register_ac_processing**.

B.7. hierarchical port

A port of a parent module.

B.8. implementation

A specific concrete implementation of the full SystemC AMS extensions, only the public shell of which need be exposed to the application (for example, parts may be pre-compiled and distributed as object code by a tool vendor). (See 2.1.2.) (See SystemC term.)

B.9. linear signal flow, LSF

A model of computation that uses the linear signal flow formalism for calculations and signal processing. (See 4.2.)

B.10. model of computation, MoC

A *model of computation* is a set of rules defining the behavior and interaction between AMS primitive modules instantiated within a module. (See 2.1.4.)

B.11. numerically singular

Numerically singular describes a situation, in which the solution of an equation system cannot be calculated.

B.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.

B.13. primitive port

A port of a primitive module.

B.14. proxy class

A class, whose only purpose is to extend the readability of certain statements that would otherwise 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 (unnamed) value returned by a function. A proxy class constructor shall not be called explicitly by an application to create a named object. (See SystemC term.)

B.15. rate

The rate defines the number of samples that are read or written per execution of a port of class **sca_tdf::sca_in**, **sca_tdf::sca_out**, **sca_tdf::sca_de::sca_in** or **sca_tdf::sca_de::sca_out** as part of the TDF model of computation. The rate of such a port shall have a positive, nonzero value.

B.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.

B.17. solver

A solver computes the solution of an equation system (e.g., a set of differential and algebraic equations).

B.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.

B.19. timed data flow, TDF

A model of computation that uses the timed data flow formalism for scheduling and signal processing. (See 4.1.)

B.20. time-domain processing

Time-domain processing is 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**.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

