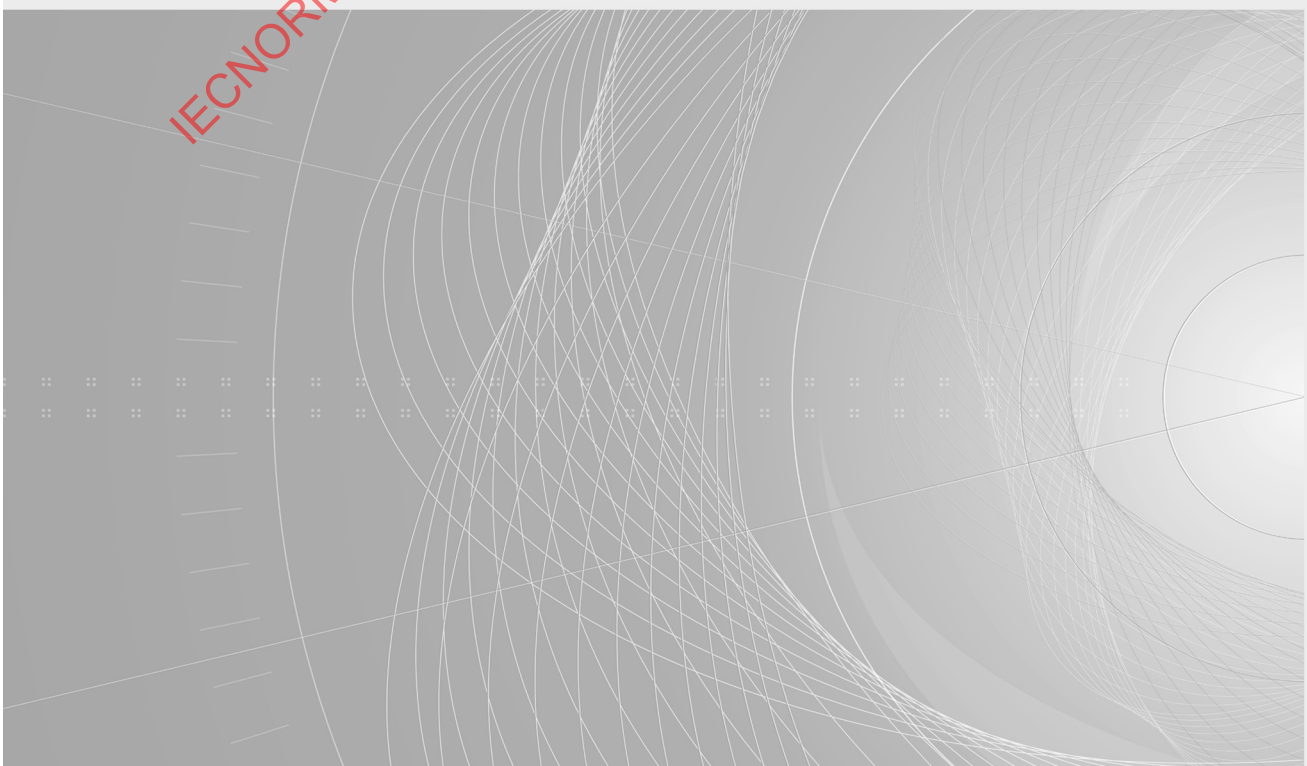
IEC 61691-8

Edition 1.0 2021-07

INTERNATIONAL STANDARD IEEE Std 1666.1™

Behavioural languages –
Part 8: Standard SystemC® Analog/Mixed-Signal Extensions Language Reference Manual



IEC 61691-8:2021-07(en) IEEE Std 1666.1-2016

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland
Tel.: +41 22 919 02 11
info@iec.ch
www.iec.ch

Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue
New York, NY 10016-5997
United States of America
stds.info@ieee.org
www.ieee.org

About the IEC
The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications
The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigendum or an amendment might have been published.

IEC publications search - webstore.iec.ch/advsearchform
The advanced search enables to find IEC publications by a variety of criteria (reference number, text, technical committee, …). It also gives information on projects, replaced and withdrawn publications.

IEC Just Published - webstore.iec.ch/justpublished
Stay up to date on all new IEC publications. Just Published details all new publications released. Available online and once a month by email.

IEC Customer Service Centre - webstore.iec.ch/csc
If you wish to give us your feedback on this publication or need further assistance, please contact the Customer Service Centre: sales@iec.ch.

IEC online collection - oc.iec.ch
Discover our powerful search engine and read freely all the publications previews. With a subscription you will always have access to up to date content tailored to your needs.

Electropedia - www.electropedia.org
The world's leading online dictionary on electrotechnology, containing more than 22 000 terminological entries in English and French, with equivalent terms in 18 additional languages. Also known as the International Electrotechnical Vocabulary (IEV) online.

IEC 61691-8

Edition 1.0 2021-07

INTERNATIONAL STANDARD IEEE Std 1666.1™

Behavioural languages – Part 8: Standard SystemC® Analog/Mixed-Signal Extensions Language Reference Manual

INTERNATIONAL ELECTROTECHNICAL COMMISSION

ICS 25.040.01; 35.060

ISBN 978-2-8322-9951-7

Contents

Behavioural languages – Part 8: Standard SystemC® Analog/Mixed-Signal Extensions Language Reference Manual

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC document(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation.

   IEEE Standards documents are developed within IEEE Societies and Standards Coordinating Committees of the IEEE Standards Association (IEEE SA) Standards Board. IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of IEEE and serve without compensation. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards. Use of IEEE Standards documents is wholly voluntary. *IEEE documents are made available for use subject to important notices and legal disclaimers (see* http://standards.ieee.org/ipr/disclaimers.html *for more information).*

   IEC collaborates closely with IEEE in accordance with conditions determined by agreement between the two organizations. This Dual Logo International Standard was jointly developed by the IEC and IEEE under the terms of that agreement.

2) The formal decisions of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees. The formal decisions of IEEE on technical matters, once consensus within IEEE Societies and Standards Coordinating Committees has been reached, is determined by a balanced ballot of materially interested parties who indicate interest in reviewing the proposed standard. Final approval of the IEEE standards document is given by the IEEE Standards Association (IEEE SA) Standards Board.

3) IEC/IEEE Publications have the form of recommendations for international use and are accepted by IEC National Committees/IEEE Societies in that sense. While all reasonable efforts are made to ensure that the technical content of IEC/IEEE Publications is accurate, IEC or IEEE cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.

4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications (including IEC/IEEE Publications) transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC/IEEE Publication and the corresponding national or regional publication shall be clearly indicated in the latter.

5) IEC and IEEE do not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC and IEEE are not responsible for any services carried out by independent certification bodies.

6) All users should ensure that they have the latest edition of this publication.

7) No liability shall attach to IEC or IEEE or their directors, employees, servants or agents including individual experts and members of technical committees and IEC National Committees, or volunteers of IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE SA) Standards Board, for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC/IEEE Publication or any other IEC or IEEE Publications.

8) Attention is drawn to the normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.

9) Attention is drawn to the possibility that implementation of this IEC/IEEE Publication may require use of material covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. IEC or IEEE shall not be held responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patent Claims or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility.

IEC 61691-8/IEEE Std 1666.1 was processed through IEC technical committee 91: Electronics assembly technology, under the IEC/IEEE Dual Logo Agreement. It is an International Standard.

The text of this International Standard is based on the following documents:

IEEE Std	FDIS	Report on voting
1666.1 (2016)	91/1712/FDIS	91/1724/RVD

Full information on the voting for its approval can be found in the report on voting indicated in the above table.

The language used for the development of this International Standard is English.

The IEC Technical Committee and IEEE Technical Committee have decided that the contents of this document will remain unchanged until the stability date indicated on the IEC website under webstore.iec.ch in the data related to the specific document. At this date, the document will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

IEEE Standard for Standard SystemC® Analog/Mixed-Signal Extensions Language Reference Manual

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 29 January 2016

IEEE-SA Standards Board

Abstract: The SystemC® Analog/Mixed-Signal (AMS) extensions are defined in this standard. SystemC AMS is an ANSI standard C++ class library for electronic system-level design and modeling for use by system architects and engineers who need to address complex heterogeneous systems that are a hybrid between analog, digital and software components. This standard provides a precise and complete definition of the SystemC AMS class library so that a SystemC AMS implementation can be developed with reference to this standard alone. The primary audiences for this standard are the implementors of the SystemC AMS class library, the implementors of tools supporting the class library, and the users of the class library.

Keywords: analog mixed signal, behavioral modeling, C++, computer languages, data flow simulation, digital systems, discrete event simulation, electronic design automation, electronic system level, electronic systems, electrical networks, hardware description language, hardware design, hardware verification, IEEE 1666™, IEEE 1666.1™, mixed-signal modeling, SystemC, SystemC AMS, signal flow modeling, system modeling, system-on-chip

Acknowledgment: Grateful acknowledgment is made to the Accellera Systems Initiative for the permission to use the following source material: Standard SystemC® AMS extensions 2.0 Language Reference Manual.

Important Notices and Disclaimers Concerning IEEE Standards Documents

IEEE documents are made available for use subject to important notices and legal disclaimers. These notices and disclaimers, or a reference to this page, appear in all standards and may be found under the heading "Important Notice" or "Important Notices and Disclaimers Concerning IEEE Standards Documents."

Notice and Disclaimer of Liability Concerning the Use of IEEE Standards Documents

IEEE Standards documents (standards, recommended practices, and guides), both full-use and trial-use, are developed within IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association ("IEEE-SA") Standards Board. IEEE ("the Institute") develops its standards through a consensus development process, approved by the American National Standards Institute ("ANSI"), which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and participate without compensation from IEEE. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims all warranties (express, implied and statutory) not included in this or any other document relating to the standard, including, but not limited to, the warranties of: merchantability; fitness for a particular purpose; non-infringement; and quality, accuracy, effectiveness, currency, or completeness of material. In addition, IEEE disclaims any and all conditions relating to: results; and workmanlike effort. IEEE standards documents are supplied "AS IS" and "WITH ALL FAULTS."

Use of an IEEE standard is wholly voluntary. The existence of an IEEE standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document, should rely upon his or her own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

IN NO EVENT SHALL IEEE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO: PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE PUBLICATION, USE OF, OR RELIANCE UPON ANY STANDARD, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND REGARDLESS OF WHETHER SUCH DAMAGE WAS FORESEEABLE.

Translations

The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE should be considered the approved IEEE standard.

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, or be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position of IEEE.

Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE. However, IEEE does not provide consulting information or advice pertaining to IEEE Standards documents. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to comments or questions except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in revisions to an IEEE standard is welcome to join the relevant IEEE working group.

Comments on standards should be submitted to the following address:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854 USA

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

IEEE draft and approved standards are copyrighted by IEEE under U.S. and international copyright laws. They are made available by IEEE and are adopted for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making these documents available for use and adoption by public authorities and private users, IEEE does not waive any rights in copyright to the documents.

Photocopies

Subject to payment of the appropriate fee, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, noncommercial use only. To arrange for payment of licensing fees, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Updating of IEEE Standards documents

Users of IEEE Standards documents should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect.

Every IEEE standard is subjected to review at least every ten years. When a document is more than ten years old and has not undergone a revision process, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE standard.

In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE-SA Website at http://ieeexplore.ieee.org/expel/standards.jsp or contact IEEE at the address listed previously. For more information about the IEEE-SA or IOWA's standards development process, visit the IEEE-SA Website at http://standards.ieee.org.

Errata

Errata, if any, for all IEEE standards can be accessed on the IEEE-SA Website at the following URL: http://standards.ieee.org/findstds/errata/index.html. Users are encouraged to check this URL for errata periodically.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE-SA Website at http://standards.ieee.org/about/sasb/patcom/patents.html. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Introduction

This introduction is not part of IEEE Std 1666.1™-2016, IEEE Standard for Standard SystemC® Analog/Mixed-Signal Extensions Language Reference Manual.

This document defines the SystemC Analog/Mixed-Signal (AMS) extensions, which is a C++ class library.

As the electronics industry builds more complex heterogeneous systems involving large numbers of components including analog, digital and software, there is an increasing need for a modeling language that can manage the complexity, heterogeneity, and size of these systems. SystemC AMS provides a mechanism for managing this complexity with its facility for modeling hardware and software together at multiple levels of abstraction. This capability is not available in traditional hardware description languages.

Stakeholders in SystemC AMS include Electronic Design Automation (EDA) companies who implement SystemC AMS class libraries and tools, integrated circuit (IC) suppliers who extend those class libraries and use SystemC AMS to model their intellectual property, and end users who use SystemC AMS to model their systems.

This standard is not intended to serve as a user's guide or to provide an introduction to SystemC AMS. Readers requiring a SystemC AMS tutorial or information on the intended use of SystemC AMS should consult the Accellera Systems Initiative Web site (http://www.accellera.org) to locate the supplemental material and training classes available.

IEEE Standard for Standard SystemC® Analog/Mixed-Signal Extensions Language Reference Manual

1. Overview

1.1 Scope

This standard defines the Analog/Mixed-Signal extensions for SystemC®[1], as an ANSI standard C++ class library based on SystemC for system and hardware design including analog/mixed-signal elements.

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™-2011[2] (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 neither 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] SystemC® is a registered trademark of the Accellera Systems Initiative.
[2] Information on references can be found in Clause 2.

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.

NOTE—It is possible to create a well-formed C++ program that is legal according to the C++ programming language standard but that violates this standard. An implementation is not obliged to detect every violation of this standard.[3]

1.5 Relationship with SystemC

This standard is built on IEEE Std 1666-2011 and extends it using the mechanisms provided by the C++ language, to provide an additional layer of analog/mixed-signal constructs. Consequently, an implementation and application may use the SystemC core language and predefined channels defined in the namespace **sc_core** and the SystemC data types defined in the namespace **sc_dt**, unless stated otherwise.

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, 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.

Readers should pay close attention to Clause 3. An understanding of the terminology and conventions 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 4.

Clause 5, Clause 6 and Clause 7 define the public interface to the SystemC AMS class library defining the predefined models of computation.

[3]Notes in text, tables, and figures are given for information only and do not contain requirements needed to implement the standard.

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 5, Clause 6, and Clause 7, using the framework and constraints provided by the class definitions starting in Clause 4.

Annex A is intended to aid the reader in the understanding of the structure and intent of the SystemC AMS class library.

Annex B is giving informal descriptions of the terms used in this standard.

2. Normative references

The following referenced documents are indispensable for the applications of this document (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained) For dated references, only the edition cited applies. For undated references, the latest edition of the referenced 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++[4]

IEEE Std 1666™-2011: IEEE Standard for Standard SystemC Language Reference Manual[5,6]

[4]ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch/). ISO/IEC publications are also available in the United States from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112, USA (http://global.ihs.com/). Electronic copies are available in the United States from the American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

[5]The IEEE standards or products referred to in this clause are trademarks of The Institute of Electrical and Electronics Engineers, Inc.

[6]IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (http://standards.ieee.org/).

3. Terminology and conventions used in this standard

3.1 Terminology

3.1.1 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*.

3.1.2 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, it uses the classes, functions, or macros defined in this standard.

3.1.3 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.

3.1.4 Specific technical terms

The specific technical terms as defined in IEEE Std 1666-2011 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 via AMS ports, which have no defined decoupling semantics. All modules have to be 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.

3.2 Syntactical conventions

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 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*†

3.2.5 Prefixes

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 defined in this standard.

3.3 Typographical conventions

The following typographical conventions are used in this standard:

a) The *italic* font is used for:
   1) Cross references to terms defined in 3.1, 3.2, and Annex B.
   2) Arguments of member functions in class definitions and in the text that are generally substituted with real values by the implementation or application.

b) 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.

c) The `constant-width` (Courier) font is used:
   1) For the SystemC AMS class definition including its member functions, data members and data types.
   2) To illustrate SystemC AMS language examples when the exact usage is depicted.
   3) 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.

3.4 Semantic conventions

3.4.1 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.

3.4.2 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 presumed that a destructor will perform any tasks necessary to release the resources allocated by the corresponding constructor.

3.4.3 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.

3.4.3.1 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.

3.4.3.2 Functions that return const char*

Certain functions have the return type **const char***; that is, they return a pointer to a null-terminated character string. The **const char*** pointer shall remain valid for the whole lifetime of the object associated to the member function, which returns this pointer.

3.4.4 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 within **sca_core**, **sca_tdf**, **sca_lsf**, **sca_eln**, **sca_ac_analysis**, and **sca_util** 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.

3.4.5 Non-compliant applications and errors

In the case where an application fails to meet an obligation imposed by this standard, the behavior of the SystemC AMS implementation shall be undefined in general. When this results in the violation of a diagnosable rule of the C++ standard, the C++ implementation shall 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 SystemC 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* being issued.

3.5 Notes and examples

Notes appear at the end of certain subclauses, designated by the uppercase word NOTE. Notes often describe the 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.

4. Core language definitions

4.1 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++.

4.1.1 #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 1666-2011 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**.

4.1.2 #include “systemc-ams.h”

The header file named **systemc-ams.h** shall add names from the namespace **sca_core, sca_ac_analysis**, **sca_util**, and **sca_tdf** 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;
using sca_util::SCA_UNDEFINED;
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_max_time;
using sca_core::sca_copyright;
using sca_core::sca_version;
using sca_core::sca_release;

// Using declarations for the following names in the sca_tdf namespace
using sca_tdf::SCA_CT_CUT;
using sca_tdf::SCA_DT_CUT;
using sca_tdf::SCA_NO_CUT;
```

NOTE—The header file **systemc-ams.h** is provided for backward compatibility with earlier versions of the SystemC AMS extensions and may be deprecated in future versions of this standard.

4.2 Base class definitions

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

4.2.1 sca_core::sca_module

4.2.1.1 Description

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

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

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

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

} // namespace sca_core
```

4.2.1.3 Constraints on usage

Any primitive module as defined in Clause 5, Clause 6 and Clause 7 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**:

a) All forms of member function **sc_core::sc_module::wait**.
b) All forms of member function **sc_core::sc_module::next_trigger**.
c) All forms of member function **sc_core::sc_module::reset_signal_is**.
d) All forms of member function **sc_core::sc_module::async_reset_signal_is**.
e) Member function **sc_core::sc_module::dont_initialize**.
f) 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**:

a) Macro **SC_CTOR**.
b) Macro **SC_HAS_PROCESS**.
c) Macro **SC_METHOD**.
d) Macro **SC_THREAD**.
e) Macro **SC_CTHREAD**.
f) Macro **SC_FORK**.
g) Macro **SC_JOIN**.

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

a) All forms of function **sc_core::wait**.
b) All forms of function **sc_core::next_trigger**.
c) Function **sc_core::sc_time_stamp**.
d) Function **sc_core::sc_delta_count**.
e) Function **sc_core::sc_get_current_process_handle**.
f) 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**:

a) Objects of class **sc_core::sc_event**.
b) Objects of class **sc_core::sc_process_handle**.
c) All objects which are derived from class **sc_core::sc_export_base**.
d) All objects which are derived from class **sc_core::sc_interface**.
e) All objects which are derived from class **sc_core::sc_module**.
f) 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 5, Clause 6, and Clause 7.

4.2.1.4 kind

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

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

4.2.1.5 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 5.3, 6.3, 7.3). 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, except in the context of the callbacks **set_attributes** and **change_attributes** of the current module derived from class **sca_tdf::sca_module** (see 5.1.1.6).

4.2.1.6 set_max_timestep

```
virtual void set_max_timestep( const sca_core::sca_time& );

virtual void set_max_timestep( double, sc_core::sc_time_unit );
```

The member function **set_max_timestep** shall define the maximum timestep or the latest first activation time after elaboration of the module according to the execution semantics of the associated predefined model of computation (see 5.3, 6.3, 7.3). If **set_max_timestep** is not called, an implementation shall set the maximum timestep to the value returned by function **sca_core::sca_max_time**. It shall be an error to call this function after the first **sc_core::sc_module::end_of_elaboration** callback has been executed, except in the context of the callbacks **set_attributes** and **change_attributes** of the current module derived from class **sca_tdf::sca_module** (see 5.1.1.6).

NOTE—The propagated maximum timestep defined by the member function **set_max_timestep** is always respected.

4.2.1.7 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.

4.2.2 sca_core::sca_interface

4.2.2.1 Description

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

4.2.2.2 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
```

4.2.2.3 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.

4.2.3 sca_core::sca_prim_channel

4.2.3.1 Description

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

4.2.3.2 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
```

4.2.3.3 Constraints on usage

Any primitive channel as defined in Clause 5, Clause 6, and Clause 7 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—Since the constructors are protected, class **sca_core::sca_prim_channel** cannot be instantiated directly. An application shall use only the channels defined in Clause 5, Clause 6, and Clause 7 and shall not directly derive any channel from this class.

4.2.3.4 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**.

4.2.3.5 kind

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

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

4.2.4 sca_core::sca_port

4.2.4.1 Description

The class **sca_core::sca_port** shall define the base class to derive ports for the predefined models of computation. It 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.

4.2.4.2 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
```

4.2.4.3 Template parameter IF

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

4.2.4.4 Constraints on usage

An application shall not use class **sca_core::sca_port** to instantiate ports, but shall use the ports defined in Clause 5, Clause 6, and Clause 7.

4.2.4.5 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**.

4.2.4.6 kind

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

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

4.2.5 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 1—The typedef **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 1666-2011.

NOTE 2—Since typedef **sca_core::sca_time** is an alias to **sc_core::sc_time**, an application can change the time resolution by calling the function **sc_core::sc_set_time_resolution** or query the time resolution by calling the function **sc_core::sc_get_time_resolution**.

4.2.6 sca_core::sca_max_time

The implementation shall provide a function **sca_core::sca_max_time** with the following declaration:

```
namespace sca_core { const sca_core::sca_time& sca_max_time(); }
```

The function **sca_core::sca_max_time** shall return the maximum value of type **sca_core::sca_time**, calculated after taking into account the time resolution. Since function **sca_core::sca_max_time** necessarily returns a reference to an object of type **sca_core::sca_time** that represents a non-zero time value, the time resolution cannot be modified after a call to **sca_core::sca_max_time**. Every call to **sca_core::sca_max_time** during a given simulation run shall return an object having the same value and representing the maximum simulation time. The actual value is implementation-defined. Whether each call to **sca_core::sca_max_time** returns a reference to the same object or a different object is implementation-defined.

4.2.7 sca_core::sca_parameter_base

4.2.7.1 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 access to the primitive module parameter values.

4.2.7.2 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& os = 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
```

4.2.7.3 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**.

4.2.7.4 kind

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

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

4.2.7.5 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.

4.2.7.6 print

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

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

4.2.7.7 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.

4.2.7.8 unlock

```
void unlock();
```

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

4.2.7.9 is_locked

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

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

4.2.7.10 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 the member function **print**(std::ostream).

4.2.8 sca_core::sca_parameter

4.2.8.1 Description

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

4.2.8.2 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& os = 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
```

4.2.8.3 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 not defined 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 shall 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.

4.2.8.4 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*.

4.2.8.5 kind

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

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

4.2.8.6 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 4.2.8.3.)

4.2.8.7 print

```
virtual void print( std::ostream& os = 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 4.2.8.3.)

4.2.8.8 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** have 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**.

4.2.8.9 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. It shall be an error if the member function **set** or **operator=** is called if the parameter is locked (see 4.2.7.9).

```
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.

4.2.9 sca_core::*sca_assign_from_proxy*†

4.2.9.1 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, whose type is passed to this base class as template parameter of type **T**.

4.2.9.2 Class definition

```
namespace sca_core {
```

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

} // namespace sca_core
```

4.2.9.3 Constraint on usage

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

4.2.10 sca_core::*sca_assign_to_proxy*†

4.2.10.1 Description

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

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

4.2.10.2 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
```

4.2.10.3 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**.

4.2.10.4 Constraint on usage

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

5. 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.

5.1 Class definitions

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

5.1.1 sca_tdf::sca_module

5.1.1.1 Description

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

5.1.1.2 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 change_attributes();
    virtual void initialize();
    virtual void reinitialize();
    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 );

    void request_next_activation( const sca_core::sca_time& );
    void request_next_activation( double, sc_core::sc_time_unit );
    void request_next_activation( const sc_core::sc_event& );
    void request_next_activation( const sca_core::sca_time&, const sc_core::sc_event& );
    void request_next_activation( double, sc_core::sc_time_unit, const sc_core::sc_event& );
    void request_next_activation( const sc_core::sc_event_or_list& );
    void request_next_activation( const sc_core::sc_event_and_list& );
    void request_next_activation( const sca_core::sca_time&, const sc_core::sc_event_or_list& );
    void request_next_activation( double, sc_core::sc_time_unit, const sc_core::sc_event_or_list& );
    void request_next_activation( const sca_core::sca_time&, const sc_core::sc_event_and_list& );
    void request_next_activation( double, sc_core::sc_time_unit, const sc_core::sc_event_and_list& );

    template<class T>
    void request_next_activation( const sca_tdf::sca_de::sca_in<T>& );

    void accept_attribute_changes();
    void reject_attribute_changes();
    void does_attribute_changes();
    void does_no_attribute_changes();

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

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

    bool is_dynamic() const;
    bool are_attribute_changes_allowed() const;
    bool are_attributes_changed() const;
    bool is_timestep_changed() const;

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

    virtual ~sca_module();
  };

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

} // namespace sca_tdf
```

5.1.1.3 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.

5.1.1.4 kind

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

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

5.1.1.5 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 5.3). The attributes can be defined using the member functions **set_timestep**, **set_max_timestep**, and **request_next_activation** of a TDF module and member functions **set_timestep**, **set_max_timestep**, **set_delay**, and **set_rate** for ports of classes **sca_tdf::sca_in**, **sca_tdf::sca_out**, **sca_tdf::sca_de::sca_in**, **sca_tdf::sca_de::sca_out**, and in addition **set_ct_delay** for ports of classes **sca_tdf::sca_out**<T, **sca_tdf::SCA_CT_CUT**, INTERP> and **sca_tdf::sca_out**<T, **sca_tdf::SCA_DT_CUT**>. The member function **set_attributes** shall be called during the elaboration phase (see 5.3.1.1). The application shall not call this member function.

5.1.1.6 change_attributes

```
virtual void change_attributes();
```

The member function **change_attributes** shall provide a context to change attributes of the TDF module and its ports. The attributes can be changed using the member functions **set_timestep**, **set_max_timestep**, or **request_next_activation** of a TDF module and member functions **set_timestep**, **set_max_timestep**, **set_delay**, and **set_rate** for ports of classes **sca_tdf::sca_in**, **sca_tdf::sca_out**, **sca_tdf::sca_de::sca_in**, **sca_tdf::sca_de::sca_out**, and in addition **set_ct_delay** for ports of classes **sca_tdf::sca_out**<T, **sca_tdf::SCA_CT_CUT**, INTERP> and **sca_tdf::sca_out**<T, **sca_tdf::SCA_DT_CUT**>. The member function shall only be called during simulation (see 5.3.2). The application shall not call this member function.

5.1.1.7 initialize

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

The member function **initialize** shall provide a context to set initial values to member variables and ports. In the context of this member function, the application can initialize the delay samples of all ports if their delay attribute has been set to a value greater than zero by using member function **initialize** of ports of classes **sca_tdf::sca_in**, **sca_tdf::sca_out**, **sca_tdf::sca_de::sca_in**, and **sca_tdf::sca_de::sca_out**. The member function shall only be called during simulation (see 5.3.2.1). The application shall not call this member function.

5.1.1.8 reinitialize

```
virtual void reinitialize();
```

The member function **reinitialize** shall provide a context to reinitialize values to member variables and ports. The member function shall be called after each cluster execution period (see 5.3.2.3). The application shall not call this member function.

NOTE—This member function may be used to reinitialize the delay samples of the associated ports by using the member function **initialize** in case of attribute changes.

5.1.1.9 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 5.1.1.11). 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 8.1). The application shall not call this member function.

5.1.1.10 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 5.1.1.12). If no application-defined member function is registered, this function shall be called during small-signal frequency-domain simulation (see 8.2). The application shall not call this member function.

5.1.1.11 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 5.1.1.9). 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.

5.1.1.12 register_ac_processing

```
void register_ac_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 5.1.1.10). 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.

5.1.1.13 request_next_activation

The member function **request_next_activation** shall override the propagated timestep defined by the member function **set_timestep** of the TDF modules and ports for the next module activation. The next module activation shall be no later than the time requested by this member function.

The time given as argument shall be taken to be relative to the time in the context of the member function **change_attributes** of the current module, in which the member function **request_next_activation** is called.

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

void request_next_activation( double, sc_core::sc_time_unit );
```

The next module activation is requested after the time given as an argument has elapsed.

```
void request_next_activation( const sc_core::sc_event& );
```

The next module activation is requested when the event, passed as an argument, is notified.

```
void request_next_activation( const sca_core::sca_time&, const sc_core::sc_event& );

void request_next_activation( double, sc_core::sc_time_unit, const sc_core::sc_event& );
```

The next module activation is requested after the time given as an argument has elapsed or when the given event is notified, whichever occurs first.

```
void request_next_activation( const sc_core::sc_event_or_list& );

void request_next_activation( const sc_core::sc_event_and_list& );
```

The next module activation is requested based on the event list, passed as argument.

```
void request_next_activation( const sca_core::sca_time&, const sc_core::sc_event_or_list& );

void request_next_activation( double, sc_core::sc_time_unit, const sc_core::sc_event_or_list& );


void request_next_activation( const sca_core::sca_time&, const sc_core::sc_event_and_list& );

void request_next_activation( double, sc_core::sc_time_unit, const sc_core::sc_event_and_list& );
```

The next module activation is requested after the time given as an argument has elapsed or in response to the event list, passed as argument, whichever is satisfied first.

```
template<class T>
void request_next_activation( const sca_tdf::sca_de::sca_in<T>& );
```

The next module activation is requested when the event, which is returned by the member function **default_event** of the port, passed as an argument, is notified.

5.1.1.14 accept_attribute_changes

```
void accept_attribute_changes();
```

The member function **accept_attribute_changes** shall mark a TDF module to accept attribute changes caused by other TDF modules, which belong to the same TDF cluster, after all **set_attributes** or **change_attributes** callbacks of the current cluster execution have been executed (see 5.3.2.3). It shall be an error if the member function is called outside the context of the member functions **set_attributes**, **change_attributes**, or the constructor of the current TDF module. If this member function is not called, a TDF module shall not accept attribute changes.

NOTE—A TDF module which accepts attribute changes is not allowed to change attributes itself, unless the member function **does_attribute_changes** has been called.

5.1.1.15 reject_attribute_changes

```
void reject_attribute_changes();
```

The member function **reject_attribute_changes** shall mark a TDF module to reject attribute changes made by other TDF modules, which belong to the same TDF cluster, after all **set_attributes** or **change_attributes** callbacks of the current cluster execution have been executed (see 5.3.2.3). It shall be an error if the TDF module is marked to reject attribute changes, and if other TDF modules, which belong to the same TDF cluster, change the attributes. It shall be an error if the member function is called outside the context of the member functions **set_attributes**, **change_attributes**, or the constructor of the current TDF module. If this member function is not called, a TDF module shall reject attribute changes.

NOTE—A TDF module which rejects attribute changes may still change attributes itself, as long as the member function **does_attribute_changes** has been called.

5.1.1.16 does_attribute_changes

```
void does_attribute_changes();
```

The member function **does_attribute_changes** shall mark a TDF module to allow it to make attribute changes after all **set_attributes** or **change_attributes** callbacks of the current cluster execution have been executed

(see 5.3.2.3). It shall be a warning if the TDF module is marked to allow making attribute changes and if there is no **change_attributes** callback implemented in the TDF module. It shall be an error if the member function is called outside the context of the member functions **set_attributes**, **change_attributes**, or the constructor of the current TDF module. If this member function is not called, a TDF module is not allowed to make attribute changes.

5.1.1.17 does_no_attribute_changes

```
void does_no_attribute_changes();
```

The member function **does_no_attribute_changes** shall mark a TDF module to disallow it to make attribute changes after all **set_attributes** or **change_attributes** callbacks of the current cluster execution have been executed (see 5.3.2.3). It shall be an error if the TDF module is marked to disallow making attribute changes, but changes its attributes or the ones of its ports. It shall be an error if the member function is called outside the context of the member functions **set_attributes**, **change_attributes**, or the constructor of the current TDF module. If this member function is not called, a TDF module is not allowed to make attribute changes.

5.1.1.18 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. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

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.

5.1.1.19 get_timestep

```
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 (see 5.3). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.1.20 get_max_timestep

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

The member function **get_max_timestep** shall return the maximum timestep of the module according to the execution semantics (see 5.3), ignoring the timesteps set for all TDF modules and TDF ports through the member function **set_timestep**. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

NOTE—The member function **request_next_activation** overrules the resolved timestep set by the member functions **set_timestep** of the TDF modules and needs to satisfy the resolved maximum timestep set by member functions **set_max_timestep** of the TDF modules. The member function **get_max_timestep** returns the maximum resolved timestep.

5.1.1.21 get_last_timestep

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

The member function **get_last_timestep** shall return the last non-zero module timestep of type **sca_core::sca_time** of the last module activation or before. For the first module activation, the member function shall return the propagated timestep. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.1.22 is_dynamic

```
bool is_dynamic() const;
```

The member function **is_dynamic** shall return true if at least one TDF module within the TDF cluster has been marked to allow making attribute changes using the member function **does_attribute_changes**. Otherwise, it shall return false. The state returned by the member function **is_dynamic** is updated after the execution of all member functions **change_attributes** of the current cluster (see 5.3.2.3). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.1.23 are_attribute_changes_allowed

```
bool are_attribute_changes_allowed() const;
```

The member function **are_attribute_changes_allowed** shall return true if changes to the attributes of the TDF module, in which the member function is called, and its ports are allowed. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module. By default, a TDF module is not allowed to make changes to its attributes.

5.1.1.24 are_attributes_changed

```
bool are_attributes_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **are_attributes_changed** shall return true if the timestep, delay, or rate of the TDF module, or its ports, have changed since the last activation of the callback **processing**. In the context of the callback **change_attributes**, the member function **are_attributes_changed** shall return true if the timestep, delay, or rate of the TDF module, or its ports, have changed between the last activation and before last activation of the callback **processing**. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.1.25 is_timestep_changed

```
bool is_timestep_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_timestep_changed** shall return true if the timestep of the TDF module, or its ports, have changed since the last activation of the callback **processing**. In the context of the callback **change_attributes**, the member function **is_timestep_changed** shall return true if the timestep of the TDF module, or its ports, have changed

between the last activation and before last activation of the callback **processing**. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

NOTE—A change in the propagated timestep (see 5.3.1.2) may change the TDF module or port timestep, which is then also detected by this member function.

5.1.1.26 Constructor

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

sca_module();
```

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

5.1.1.27 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
}
```

5.1.2 sca_tdf::sca_signal_if

5.1.2.1 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.

5.1.2.2 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
```

5.1.3 sca_tdf::sca_signal

5.1.3.1 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.

5.1.3.2 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* );

    virtual const char* kind() const;

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

} // namespace sca_tdf
```

5.1.3.3 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 not defined by this standard. The implementation shall use this operator for writing trace values in time-domain simulation (see 9.1).

```
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 shall 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 or reading from ports of type **T**.

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

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

5.1.3.4 Constructors

```
sca_signal();

explicit sca_signal( const char* );
```

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**.

5.1.3.5 kind

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

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

5.1.4 sca_tdf::sca_default_interpolator

5.1.4.1 Description

The class **sca_tdf::sca_default_interpolator** shall define the default interpolation mechanism for the continuous-time decoupling port of class **sca_tdf::sca_out**<T, **sca_tdf::SCA_CT_CUT**, INTERP>. The specialized classes **sca_tdf::sca_default_interpolator**<double> and **sca_tdf::sca_default_interpolator**<**sca_util::sca_complex**> shall provide a default interpolation mechanism by interpreting the signal as continuous in time. For all other types, the class shall keep the value of the last available time point.

5.1.4.2 Class definition

```
namespace sca_tdf {

  template<class T>
  class sca_default_interpolator
  {
   public:
    void store_value( const sca_core::sca_time&, const T& );
    T get_value( const sca_core::sca_time& ) const;
  };

  template<>
  class sca_default_interpolator<double>
  {
   public:
    void store_value( const sca_core::sca_time&, const double& );
    double get_value( const sca_core::sca_time& ) const;
  };

  template<>
  class sca_default_interpolator<sca_util::sca_complex>
  {
   public:
    void store_value( const sca_core::sca_time&, const sca_util::sca_complex& );
    sca_util::sca_complex get_value( const sca_core::sca_time& ) const;
  };

} // namespace sca_tdf
```

5.1.4.3 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) If the default assignment semantics are inadequate (in the sense given in this subclause), the following assignment operator shall 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& );
```

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

5.1.4.4 store_value

```
void store_value( const sca_core::sca_time&, const T& );

void store_value( const sca_core::sca_time&, const double& );

void store_value( const sca_core::sca_time&, const sca_util::sca_complex& );
```

An implementation shall use the member function **store_value** to store a time-value pair to the interpolator class.

5.1.4.5 get_value

```
T get_value( const sca_core::sca_time& ) const;

double get_value( const sca_core::sca_time& ) const;

sca_util::sca_complex get_value( const sca_core::sca_time& ) const;
```

An implementation shall use the member function **get_value** to return an interpolated value at a given time point given as argument.

5.1.5 sca_tdf::sca_in

5.1.5.1 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.

5.1.5.2 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* );

    void set_delay( unsigned long );
    void set_rate( unsigned long );
    void set_timestep( const sca_core::sca_time& );
    void set_timestep( double, sc_core::sc_time_unit );
    void set_max_timestep( const sca_core::sca_time& );
    void set_max_timestep( double, sc_core::sc_time_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( unsigned long sample_id = 0 ) const;
    sca_core::sca_time get_max_timestep() const;
    sca_core::sca_time get_last_timestep( unsigned long sample_id = 0 ) const;
```

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

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

    bool is_timestep_changed( unsigned long sample_id = 0 ) const;
    bool is_rate_changed() const;
    bool is_delay_changed() const;

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

} // namespace sca_tdf
```

5.1.5.3 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 not defined by this standard. The implementation shall use this operator for writing trace values in time-domain simulation (see 9.1).

```
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 shall 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 or reading from ports of type **T**.

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

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

5.1.5.4 Constructors

```
sca_in();

explicit sca_in( const char* );
```

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**.

5.1.5.5 set_delay

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

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 context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.5.6 set_rate

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

The member function **set_rate** shall define the number of samples that can be read during the execution of the member function **processing** of the current TDF module by using 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 context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.5.7 set_timestep

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

void set_timestep( double, sc_core::sc_time_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 5.3). It shall be an error if the member function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.5.8 set_max_timestep

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

void set_max_timestep( double, sc_core::sc_time_unit );
```

The member function **set_max_timestep** shall define the maximum timestep between two consecutive samples. If **set_max_timestep** is not called, an implementation shall set the maximum timestep to the value returned by function **sca_core::sca_max_time**. It shall be an error if the member function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.5.9 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.5.10 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.5.11 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 with index *sample_id*. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

The relationship shown in Equation (5.1) shall hold:

$$P.\mathbf{get\_time}(sample\_id) = M.\mathbf{get\_time}() + \frac{M.\mathbf{get\_timestep}() \cdot sample\_id}{P.\mathbf{get\_rate}()} \tag{5.1}$$

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 5.3).

NOTE—The relation is valid within the time resolution bound, which is returned by the function **sc_core::sc_get_time_resolution** (see 5.3.1.2).

5.1.5.12 get_timestep

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

The member function **get_timestep** shall return the timestep between the preceding and current sample with index *sample_id*. If the preceding sample is not available, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.5.13 get_max_timestep

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

The member function **get_max_timestep** shall return the maximum timestep between two consecutive samples according to the execution semantics (see 5.3), ignoring the timesteps set for all TDF modules and TDF ports through the member function **set_timestep**. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.5.14 get_last_timestep

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

The member function **get_last_timestep** shall return the timestep between the two samples preceding the sample with index *sample_id*. If the timestep between these two preceding samples is equal to **sc_core::SC_ZERO_TIME**, the member function shall return the last non-zero timestep. If one or both of the preceding samples are not available or if all preceding timesteps are all equal to **sc_core::SC_ZERO_TIME**, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.5.15 kind

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

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

5.1.5.16 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 functions **initialize** or **reinitialize** of the current TDF 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 the port requires that the port has been assigned a delay using the member function **set_delay**, which shall be called in the member functions **set_attributes** or **change_attributes** of the TDF module.

5.1.5.17 read_delayed_value

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

The member function **read_delayed_value** shall return a reference to the value of a delayed sample that is available at the port. The argument *sample_id* denotes the index of the delayed sample being read. The samples shall be indexed from zero to *P*.**get_delay**()–1, where *P* denotes the port. A *sample_id* of zero shall refer to the first delayed sample in time. It shall be an error if *sample_id* is greater than or equal to the port delay.

The member function shall only be called in the member function **reinitialize** of the current TDF module. Otherwise, it shall be an error. Consecutive reads with the same *sample_id* during the same module activation shall return the same delayed value.

5.1.5.18 is_timestep_changed

```
bool is_timestep_changed( unsigned long sample_id = 0 ) const;
```

The member function **is_timestep_changed** shall return true if the timestep of the sample with index *sample_id* of the TDF port, returned by its member function **get_timestep**(*sample_id*), has changed with respect to the preceding sample. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.5.19 is_rate_changed

```
bool is_rate_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed between the last activation and before last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall

be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.5.20 is_delay_changed

```
bool is_delay_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed between the last activation and before last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.5.21 read

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

operator const T& () const;

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

The member functions **read**, **operator const T&**, and **operator[]** shall return a reference to the value of a particular sample that is available at the port. 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 functions **read**, **operator const T&**, and **operator[]** shall only be called in the context of the member functions **processing** and **ac_processing** 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.

5.1.6 sca_tdf::sca_out

5.1.6.1 Description

The class **sca_tdf::sca_out** shall define a port class for the TDF MoC.

5.1.6.2 Class definition

```
namespace sca_tdf {

  enum sca_cut_policy
  {
    SCA_NO_CUT,
    SCA_CT_CUT,
    SCA_DT_CUT
  };

  template<class T, sca_tdf::sca_cut_policy CUT_POL = sca_tdf::SCA_NO_CUT,
           class INTERP = sca_tdf::sca_default_interpolator<T> >
  class sca_out implementation-defined ;

} // namespace sca_tdf
```

5.1.6.3 Constraint on usage

An application shall not instantiate the **sca_tdf::sca_out** class template with template parameter combinations matching none of the partial template specializations **sca_tdf::sca_out<T>**, **sca_tdf::sca_out**<T, **sca_tdf::SCA_CT_CUT**, INTERP> or **sca_tdf::sca_out**<T, **sca_tdf::SCA_DT_CUT>** (see 5.1.7, 5.1.8, and 5.1.9 respectively).

5.1.6.4 Template parameters

The first 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 not defined by this standard. The implementation shall use this operator for writing trace values in time-domain simulation (see 9.1).

```
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 shall 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 or reading from ports of type **T**.

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

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

The second argument passed as template parameter CUT_POL is an optional port decoupling policy of type **sca_tdf::sca_cut_policy**. The port decoupling policy argument determines the rules how sample time points to ports of class **sca_tdf::sca_in** shall be decoupled:

— The policy **sca_tdf::SCA_NO_CUT** means that the port shall not decouple TDF clusters. As such, it acts as a normal TDF port of class **sca_tdf::sca_out**<T>. For this type of port, the third template parameter INTERP shall be ignored.

— The policy **sca_tdf::SCA_CT_CUT** means that the port shall decouple TDF clusters using interpolation as defined by the third template parameter INTERP.

— The policy **sca_tdf::SCA_DT_CUT** means that the port shall decouple TDF clusters using a sample-and-hold regime. For this type of port, the third template parameter INTERP shall be ignored.

The third argument passed as template parameter INTERP is an optional interpolation mechanism for the type T. By default, the class **sca_tdf::sca_default_interpolator**<T> shall be used. If any constructor for type INTERP exists, a default constructor for type INTERP shall be defined. An application may define its own interpolator classes for other types. Such class shall provide the following public member functions:

```
void store_value( const sca_core::sca_time&, const T& );

T get_value( const sca_core::sca_time& ) const;
```

An implementation shall confirm that the time passed as argument to member function **store_value** is larger than the time passed as argument to the last call to member function **store_value**.

An implementation shall confirm that the time passed as argument to member function **get_value** is smaller than or equal to the time passed as argument to the last call to member function **store_value** and larger than or equal to the time passed as argument to the before last call of member function **store_value** (see 5.1.4).

5.1.7 sca_tdf::sca_out<T>

5.1.7.1 Description

The class **sca_tdf::sca_out<T>** 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.

5.1.7.2 Class definition

```
namespace sca_tdf {

  template<class T>
  class sca_out_base : public sca_core::sca_port< sca_tdf::sca_signal_if<T> >
  { implementation-defined };

  template<class T>
  class sca_out<T, sca_tdf::SCA_NO_CUT> : public sca_tdf::sca_out_base<T>
  {
   public:
    sca_out();
    explicit sca_out( const char* );

    void set_delay( unsigned long );
    void set_rate( unsigned long );
    void set_timestep( const sca_core::sca_time& );
    void set_timestep( double, sc_core::sc_time_unit );
    void set_max_timestep( const sca_core::sca_time& );
    void set_max_timestep( double, sc_core::sc_time_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( unsigned long sample_id = 0 ) const;
    sca_core::sca_time get_max_timestep() const;
    sca_core::sca_time get_last_timestep( unsigned long sample_id = 0 ) const;

    virtual const char* kind() const;

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

    bool is_timestep_changed( unsigned long sample_id = 0 ) const;
    bool is_rate_changed() const;
    bool is_delay_changed() const;

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

} // namespace sca_tdf
```

5.1.7.3 Constructors

```
sca_out();

explicit sca_out( const char* );
```

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**.

5.1.7.4 set_delay

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

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 context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.7.5 set_rate

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

The member function **set_rate** shall define the number of samples that can be written during the execution of the member function **processing** of the current TDF module by using 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 context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.7.6 set_timestep

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

void set_timestep( double, sc_core::sc_time_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 5.3). It shall be an error if the member function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.7.7 set_max_timestep

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

void set_max_timestep( double, sc_core::sc_time_unit );
```

The member function **set_max_timestep** shall define the maximum timestep between two consecutive samples. If **set_max_timestep** is not called, an implementation shall set the maximum timestep to the value returned by function **sca_core::sca_max_time**. It shall be an error if the member function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.7.8 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.7.9 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.7.10 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 with index *sample_id*. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

The relationship shown in Equation (5.2) shall hold:

$$P.\mathbf{get\_time}(sample\_id) = M.\mathbf{get\_time}() + \frac{M.\mathbf{get\_timestep}() \cdot sample\_id}{P.\mathbf{get\_rate}()} \tag{5.2}$$

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 5.3).

NOTE—The relation is valid within the time resolution bound, which is returned by the function **sc_core::sc_get_time_resolution** (see 5.3.1.2).

5.1.7.11 get_timestep

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

The member function **get_timestep** shall return the timestep between the preceding and current sample with index *sample_id*. If the preceding sample is not available, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.7.12 get_max_timestep

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

The member function **get_max_timestep** shall return the maximum timestep between two consecutive samples according to the execution semantics (see 5.3), ignoring the timesteps set for all TDF modules and TDF ports through the member function **set_timestep**. It shall be an error if the member function is called outside the

context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.7.13 get_last_timestep

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

The member function **get_last_timestep** shall return the timestep between the two samples preceding the sample with index *sample_id*. If the timestep between these two preceding samples is equal to **sc_core::SC_ZERO_TIME**, the member function shall return the last non-zero timestep. If one or both of the preceding samples are not available or if all preceding timesteps are all equal to **sc_core::SC_ZERO_TIME**, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.7.14 kind

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

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

5.1.7.15 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 functions **initialize** or **reinitialize** of the current TDF 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 the port requires that the port has been assigned a delay using the member function **set_delay**, which shall be called in the member functions **set_attributes** or **change_attributes** of the TDF module.

5.1.7.16 read_delayed_value

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

The member function **read_delayed_value** shall return a reference to the value of a delayed sample that is available at the port. The argument *sample_id* denotes the index of the delayed sample being read. The samples shall be indexed from zero to *P*.**get_delay**()–1, where *P* denotes the port. A *sample_id* of zero shall refer to the first delayed sample in time. It shall be an error if *sample_id* is greater than or equal to the port delay.

The member function shall only be called in the member function **reinitialize** of the current TDF module. Otherwise, it shall be an error. Consecutive reads with the same *sample_id* during the same module activation shall return the same delayed value.

5.1.7.17 is_timestep_changed

```
bool is_timestep_changed( unsigned long sample_id = 0 ) const;
```

The member function **is_timestep_changed** shall return true if the timestep of the sample with index *sample_id* of the TDF port, returned by its member function **get_timestep**(*sample_id*), has changed with respect to the preceding sample. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.7.18 is_rate_changed

```
bool is_rate_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed between the last activation and before last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.7.19 is_delay_changed

```
bool is_delay_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed between the last activation and before last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.7.20 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 functions **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= ( sca_tdf::sca_de::sca_in<T>& );
```

The **operator=** shall read the first value from the input port of class **sca_tdf::sca_in** or **sca_tdf::sca_de::sca_in** and write it to the first value of the output port.

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

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

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

The member functions **write**, **operator=**, and **operator[]** shall only be called in the context of the member function **processing** 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.

5.1.8 sca_tdf::sca_out<T, sca_tdf::SCA_CT_CUT, INTERP>

5.1.8.1 Description

The class **sca_tdf::sca_out**<T, **sca_tdf::SCA_CT_CUT**, INTERP> shall define a decoupling 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. The samples read by the connected ports of class **sca_tdf::sca_in** shall be interpreted as forming a continuous-time signal.

5.1.8.2 Class definition

```
namespace sca_tdf {

  template<class T, class INTERP>
  class sca_out<T, sca_tdf::SCA_CT_CUT, INTERP> : public sca_tdf::sca_out_base<T>
  {
   public:
    sca_out();
    explicit sca_out( const char* );

    void set_delay( unsigned long );
    void set_ct_delay( const sca_core::sca_time& );
    void set_ct_delay( double, sc_core::sc_time_unit );
    void set_rate( unsigned long );
    void set_timestep( const sca_core::sca_time& );
    void set_timestep( double, sc_core::sc_time_unit );
    void set_max_timestep( const sca_core::sca_time& );
    void set_max_timestep( double, sc_core::sc_time_unit );

    unsigned long get_delay() const;
    sca_core::sca_time get_ct_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( unsigned long sample_id = 0 ) const;
    sca_core::sca_time get_max_timestep() const;
    sca_core::sca_time get_last_timestep( unsigned long sample_id = 0 ) const;

    virtual const char* kind() const;

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

    bool is_timestep_changed( unsigned long sample_id = 0 ) const;
    bool is_rate_changed() const;
    bool is_delay_changed() const;

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

   private:
    // Disabled
    sca_out<T, sca_tdf::SCA_CT_CUT, INTERP>(
      const sca_tdf::sca_out<T, sca_tdf::SCA_CT_CUT, INTERP>& );
```

```
    sca_tdf::sca_out<T, sca_tdf::SCA_CT_CUT, INTERP>& operator= (
      const sca_tdf::sca_out<T, sca_tdf::SCA_CT_CUT, INTERP>& );
  };

} // namespace sca_tdf
```

5.1.8.3 Constraint on usage

A port of class **sca_tdf::sca_out**<T, **sca_tdf::SCA_CT_CUT**, INTERP> shall only be a member of a module derived from class **sca_tdf::sca_module**; otherwise, it shall be an error.

5.1.8.4 Constructors

```
sca_out();

explicit sca_out( const char* );
```

The constructor for class **sca_tdf::sca_out**<T, **sca_tdf::SCA_CT_CUT**, INTERP> 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_ct_cut**") to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sca_core::sca_port**.

5.1.8.5 set_delay

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

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. The total delay of the port shall be the sum of the delay set by the member functions **set_delay** and **set_ct_delay**. It shall be an error if the total delay of the port is smaller than one timestep. It shall be an error if the member function is called outside the constructor of the parent module or called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.8.6 set_ct_delay

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

void set_ct_delay( double, sc_core::sc_time_unit );
```

The member function **set_ct_delay** shall define the continuous-time delay to be inserted before the first output sample. If the member function is not called, the continuous-time delay is set to **sc_core::SC_ZERO_TIME**. The total delay of the port shall be the sum of the delay set by the member functions **set_delay** and **set_ct_delay**. It shall be an error if the total delay of the port is smaller than one timestep. It shall be an error if the member function is called outside the constructor of the parent module or called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.8.7 set_rate

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

The member function **set_rate** shall define the number of samples that can be written during the execution of the member function **processing** of the current TDF module by using 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 context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.8.8 set_timestep

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

void set_timestep( double, sc_core::sc_time_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 5.3). It shall be an error if the member function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.8.9 set_max_timestep

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

void set_max_timestep( double, sc_core::sc_time_unit );
```

The member function **set_max_timestep** shall define the maximum timestep between two consecutive samples. If **set_max_timestep** is not called, an implementation shall set the maximum timestep to the value returned by function **sca_core::sca_max_time**. It shall be an error if the member function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.8.10 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.8.11 get_ct_delay

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

The member function **get_ct_delay** shall return the continuous-time delay of type **sca_core::sca_time** set at the port. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.8.12 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.8.13 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 with index *sample_id*. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

The relationship shown in Equation (5.3) shall hold:

$$P.\textbf{get\_time}(sample\_id) = M.\textbf{get\_time}() + \frac{M.\textbf{get\_timestep}() \cdot sample\_id}{P.\textbf{get\_rate}()} \tag{5.3}$$

where *P* is an instance of a port of class **sca_tdf::sca_out<T, sca_tdf::SCA_CT_CUT, INTERP>** and *M* is the parent module derived from class **sca_tdf::sca_module** (see 5.3).

NOTE—The relation is valid within the time resolution bound, which is returned by the function **sc_core::sc_get_time_resolution** (see 5.3.1.2).

5.1.8.14 get_timestep

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

The member function **get_timestep** shall return the timestep between the preceding and current sample with index *sample_id*. If the preceding sample is not available, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.8.15 get_max_timestep

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

The member function **get_max_timestep** shall return the maximum timestep between two consecutive samples according to the execution semantics (see 5.3), ignoring the timesteps set for all TDF modules and TDF ports through the member function **set_timestep**. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.8.16 get_last_timestep

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

The member function **get_last_timestep** shall return the timestep between the two samples preceding the sample with index *sample_id*. If the timestep between these two preceding samples is equal to **sc_core::SC_ZERO_TIME**, the member function shall return the last non-zero timestep. If one or both of the preceding samples are not available or if all preceding timesteps are all equal to **sc_core::SC_ZERO_TIME**, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.8.17 kind

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

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

5.1.8.18 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 constructor of the parent module or in the member functions **initialize** or **reinitialize** of the current TDF 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**<T, **sca_tdf::SCA_CT_CUT**, INTERP> requires that the port has been assigned a delay using the member function **set_delay**, which shall be called in the member functions **set_attributes** or **change_attributes** of the TDF module.

5.1.8.19 set_initial_value

```
void set_initial_value( const T& );
```

The member function **set_initial_value** shall set the initial value of the TDF decoupling port to the value supplied as argument. It shall be an error if the member function is called outside the context of the member function **set_attributes** of the current TDF module.

5.1.8.20 read_delayed_value

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

The member function **read_delayed_value** shall return a reference to the value of a delayed sample that is available at the port. The argument *sample_id* denotes the index of the delayed sample being read. The samples shall be indexed from zero to *P*.**get_delay**()–1, where *P* denotes the port. A *sample_id* of zero shall refer to the first delayed sample in time. It shall be an error if *sample_id* is greater than or equal to the port delay.

The member function shall only be called in the member function **reinitialize** of the current TDF module. Otherwise, it shall be an error. Consecutive reads with the same *sample_id* during the same module activation shall return the same delayed value.

5.1.8.21 is_timestep_changed

```
bool is_timestep_changed( unsigned long sample_id = 0 ) const;
```

The member function **is_timestep_changed** shall return true if the timestep of the sample with index *sample_id* of the TDF port, returned by its member function **get_timestep**(*sample_id*), has changed with respect to the preceding sample. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.8.22 is_rate_changed

```
bool is_rate_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed between the last activation and before last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.8.23 is_delay_changed

```
bool is_delay_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed between the last activation and before last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.8.24 write

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

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

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

The member functions **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, sca_tdf::SCA_CT_CUT, INTERP>& operator= ( const sca_tdf::sca_in<T>& );

sca_tdf::sca_out<T, sca_tdf::SCA_CT_CUT, INTERP>& operator= ( sca_tdf::sca_de::sca_in<T>& );
```

The **operator=** shall read the first value from the input port of class **sca_tdf::sca_in** or **sca_tdf::sca_de::sca_in** and write it to the first value of the output port.

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

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

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

The member functions **write**, **operator=**, and **operator[]** shall only be called in the context of the member function **processing** 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.

5.1.9 sca_tdf::sca_out<T, sca_tdf::SCA_DT_CUT>

5.1.9.1 Description

The class **sca_tdf::sca_out<T, sca_tdf::SCA_DT_CUT>** shall define a decoupling 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. The samples read by the connected ports of class **sca_tdf::sca_in** shall always have the value of the last available time point.

5.1.9.2 Class definition

```
namespace sca_tdf {

  template<class T>
  class sca_out<T, sca_tdf::SCA_DT_CUT> : public sca_tdf::sca_out_base<T>
  {
   public:
    sca_out();
    explicit sca_out( const char* );

    void set_delay( unsigned long );
    void set_ct_delay( const sca_core::sca_time& );
    void set_ct_delay( double, sc_core::sc_time_unit );
    void set_rate( unsigned long );
    void set_timestep( const sca_core::sca_time& );
    void set_timestep( double, sc_core::sc_time_unit );
    void set_max_timestep( const sca_core::sca_time& );
    void set_max_timestep( double, sc_core::sc_time_unit );

    unsigned long get_delay() const;
    sca_core::sca_time get_ct_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( unsigned long sample_id = 0 ) const;
    sca_core::sca_time get_max_timestep() const;
    sca_core::sca_time get_last_timestep( unsigned long sample_id = 0 ) const;

    virtual const char* kind() const;

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

    bool is_timestep_changed( unsigned long sample_id = 0 ) const;
    bool is_rate_changed() const;
    bool is_delay_changed() const;

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

   private:
    // Disabled
    sca_out<T, sca_tdf::SCA_DT_CUT>( const sca_tdf::sca_out<T, sca_tdf::SCA_DT_CUT>& );
    sca_tdf::sca_out<T, sca_tdf::SCA_DT_CUT>& operator= (
      const sca_tdf::sca_out<T, sca_tdf::SCA_DT_CUT>& );
  };

} // namespace sca_tdf
```

5.1.9.3 Constraint on usage

A port of class **sca_tdf::sca_out**<T, **sca_tdf::SCA_DT_CUT**> shall only be a member of a module derived from class **sca_tdf::sca_module**; otherwise, it shall be an error.

5.1.9.4 Constructors

```
sca_out();

explicit sca_out( const char* );
```

The constructor for class **sca_tdf::sca_out<T, sca_tdf::SCA_DT_CUT>** 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_dt_cut**") to generate a unique string name that it shall then pass through to the constructor belonging to the base class **sca_core::sca_port**.

5.1.9.5 set_delay

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

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. The total delay of the port shall be the sum of the delay set by the member functions **set_delay** and **set_ct_delay**. It shall be an error if the member function is called outside the constructor of the parent module or called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.9.6 set_ct_delay

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

void set_ct_delay( double, sc_core::sc_time_unit );
```

The member function **set_ct_delay** shall define the continuous-time delay to be inserted before the first output sample. If the member function is not called, the continuous-time delay is set to **sc_core::SC_ZERO_TIME**. The total delay of the port shall be the sum of the delay set by the member functions **set_delay** and **set_ct_delay**. It shall be an error if the member function is called outside the constructor of the parent module or called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.9.7 set_rate

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

The member function **set_rate** shall define the number of samples that can be written during the execution of the member function **processing** of the current TDF module by using 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 context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.9.8 set_timestep

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

void set_timestep( double, sc_core::sc_time_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 5.3). It shall be an error if the member function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.9.9 set_max_timestep

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

void set_max_timestep( double, sc_core::sc_time_unit );
```

The member function **set_max_timestep** shall define the maximum timestep between two consecutive samples. If **set_max_timestep** is not called, an implementation shall set the maximum timestep to the value returned by function **sca_core::sca_max_time**. It shall be an error if the member function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.9.10 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.9.11 get_ct_delay

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

The member function **get_ct_delay** shall return the continuous-time delay of type **sca_core::sca_time** set at the port. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.9.12 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.9.13 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 with index *sample_id*. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

The relationship shown in Equation (5.4) shall hold:

$$P.\textbf{get\_time}(sample\_id) = M.\textbf{get\_time}() + \frac{M.\textbf{get\_timestep}() \cdot sample\_id}{P.\textbf{get\_rate}()} \tag{5.4}$$

where *P* is an instance of a port of class **sca_tdf::sca_out<T, sca_tdf::SCA_DT_CUT>** and *M* is the parent module derived from class **sca_tdf::sca_module** (see 5.3).

NOTE—The relation is valid within the time resolution bound, which is returned by the function **sc_core::sc_get_time_resolution** (see 5.3.1.2).

5.1.9.14 get_timestep

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

The member function **get_timestep** shall return the timestep between the preceding and current sample with index *sample_id*. If the preceding sample is not available, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.9.15 get_max_timestep

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

The member function **get_max_timestep** shall return the maximum timestep between two consecutive samples according to the execution semantics (see 5.3), ignoring the timesteps set for all TDF modules and TDF ports through the member function **set_timestep**. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.9.16 get_last_timestep

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

The member function **get_last_timestep** shall return the timestep between the two samples preceding the sample with index *sample_id*. If the timestep between these two preceding samples is equal to **sc_core::SC_ZERO_TIME**, the member function shall return the last non-zero timestep. If one or both of the preceding samples are not available or if all preceding timesteps are all equal to **sc_core::SC_ZERO_TIME**, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.9.17 kind

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

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

5.1.9.18 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 constructor of the parent module or in the member functions **initialize** or **reinitialize** of the current TDF 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 the port requires that the port has been assigned a delay using the member function **set_delay**, which shall be called in the member functions **set_attributes** or **change_attributes** of the TDF module.

5.1.9.19 set_initial_value

```
void set_initial_value( const T& );
```

The member function **set_initial_value** shall set the initial value of the TDF decoupling port to the value supplied as argument. It shall be an error if the member function is called outside the context of the member function **set_attributes** of the current TDF module.

5.1.9.20 read_delayed_value

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

The member function **read_delayed_value** shall return a reference to the value of a delayed sample that is available at the port. The argument *sample_id* denotes the index of the delayed sample being read. The samples shall be indexed from zero to *P*.**get_delay**()–1, where *P* denotes the port. A *sample_id* of zero shall refer to the first delayed sample in time. It shall be an error if *sample_id* is greater than or equal to the port delay.

The member function shall only be called in the member function **reinitialize** of the current TDF module. Otherwise, it shall be an error. Consecutive reads with the same *sample_id* during the same module activation shall return the same delayed value.

5.1.9.21 is_timestep_changed

```
bool is_timestep_changed( unsigned long sample_id = 0 ) const;
```

The member function **is_timestep_changed** shall return true if the timestep of the sample with index *sample_id* of the TDF port, returned by its member function **get_timestep**(*sample_id*), has changed with respect to the preceding sample. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.9.22 is_rate_changed

```
bool is_rate_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed between the last activation and before last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall

be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.9.23 is_delay_changed

```
bool is_delay_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed between the last activation and before last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.9.24 write

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

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

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

The member functions **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, sca_tdf::SCA_DT_CUT>& operator= ( const sca_tdf::sca_in<T>& );

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

The **operator=** shall read the first value from the input port of class **sca_tdf::sca_in** or **sca_tdf::sca_de::sca_in** and write it to the first value of the output port.

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

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

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

The member functions **write**, **operator=**, and **operator[]** shall only be called in the context of the member function **processing** 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.

5.1.10 sca_tdf::sca_de::sca_in, sca_tdf::sc_in

5.1.10.1 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 5.3.2.4).

The class **sca_tdf::sc_in** shall be defined as an alias for class **sca_tdf::sca_de::sca_in**.

5.1.10.2 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* );

      void set_delay( unsigned long );
      void set_rate( unsigned long );
      void set_timestep( const sca_core::sca_time& );
      void set_timestep( double, sc_core::sc_time_unit );
      void set_max_timestep( const sca_core::sca_time& );
      void set_max_timestep( double, sc_core::sc_time_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( unsigned long sample_id = 0 ) const;
      sca_core::sca_time get_max_timestep() const;
      sca_core::sca_time get_last_timestep( unsigned long sample_id = 0 ) const;

      virtual const char* kind() const;

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

      bool is_timestep_changed( unsigned long sample_id = 0 ) const;
      bool is_rate_changed() const;
      bool is_delay_changed() const;

      const T& read( unsigned long sample_id = 0 );
      operator const T& ();
      const T& operator[] ( unsigned long sample_id );

      const sc_event& default_event() const;
      const sc_event& value_changed_event() const;
      bool event() const;

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

      virtual 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> >& );

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

5.1.10.3 Constraint on usage

A port of class **sca_tdf::sca_de::sca_in** and **sca_tdf::sc_in** shall only be a member of a module derived from class **sca_tdf::sca_module**; otherwise, it shall be an error.

5.1.10.4 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 not defined by this standard. The implementation shall use this operator for writing trace values in time-domain simulation (see 9.1).

```
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 shall 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 or reading from ports of type **T**.

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

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

5.1.10.5 Constructors

```
sca_in();

explicit sca_in( const char* );
```

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.

```
sc_in() : sca_tdf::sca_de::sca_in<T>() {}

explicit sc_in( const char* name_ ) : sca_tdf::sca_de::sca_in<T>( name_ ) {}
```

The constructor for class **sca_tdf::sc_in** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sca_tdf::sca_de::sca_in** 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**.

5.1.10.6 set_delay

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

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 context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.10.7 set_rate

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

The member function **set_rate** shall define the number of samples that can be read during the execution of the member function **processing** of the current TDF module by using 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 context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.10.8 set_timestep

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

void set_timestep( double, sc_core::sc_time_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 5.3). It shall be an error if the function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.10.9 set_max_timestep

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

void set_max_timestep( double, sc_core::sc_time_unit );
```

The member function **set_max_timestep** shall define the maximum timestep between two consecutive samples. If **set_max_timestep** is not called, an implementation shall set the maximum timestep to the value returned by function **sca_core::sca_max_time**. It shall be an error if the member function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.10.10 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.10.11 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.10.12 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 with index *sample_id*. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

The relationship shown in Equation (5.5) shall hold:

$$P.\mathbf{get\_time}(sample\_id) = M.\mathbf{get\_time}() + \frac{M.\mathbf{get\_timestep}() \cdot sample\_id}{P.\mathbf{get\_rate}()} \tag{5.5}$$

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 5.3).

NOTE—The relation is valid within the time resolution bound, which is returned by the function **sc_core::sc_get_time_resolution** (see 5.3.1.2).

5.1.10.13 get_timestep

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

The member function **get_timestep** shall return the timestep between the preceding and current sample with index *sample_id*. If the preceding sample is not available, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.10.14 get_max_timestep

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

The member function **get_max_timestep** shall return the maximum timestep between two consecutive samples according to the execution semantics (see 5.3), ignoring the timesteps set for all TDF modules and TDF ports through the member function **set_timestep**. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.10.15 get_last_timestep

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

The member function **get_last_timestep** shall return the timestep between the two samples preceding the sample with index *sample_id*. If the timestep between these two preceding samples is equal to **sc_core::SC_ZERO_TIME**, the member function shall return the last non-zero timestep. If one or both of the preceding samples are not available or if all preceding timesteps are all equal to **sc_core::SC_ZERO_TIME**, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.10.16 kind

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

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

5.1.10.17 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 functions **initialize** or **reinitialize** of the current TDF 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 the port requires that the port has been assigned a delay using the member function **set_delay**, which shall be called in the member functions **set_attributes** or **change_attributes** of the TDF module.

5.1.10.18 read_delayed_value

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

The member function **read_delayed_value** shall return a reference to the value of a delayed sample that is available at the port. The argument *sample_id* denotes the index of the delayed sample being read. The samples shall be indexed from zero to *P*.**get_delay**()–1, where *P* denotes the port. A *sample_id* of zero shall refer to the first delayed sample in time. It shall be an error if *sample_id* is greater than or equal to the port delay.

The member function shall only be called in the member function **reinitialize** of the current TDF module. Otherwise, it shall be an error. Consecutive reads with the same *sample_id* during the same module activation shall return the same delayed value.

5.1.10.19 is_timestep_changed

```
bool is_timestep_changed( unsigned long sample_id = 0 ) const;
```

The member function **is_timestep_changed** shall return true if the timestep of the sample with index *sample_id* of the TDF port, returned by its member function **get_timestep**(*sample_id*), has changed with respect to the preceding sample. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.10.20 is_rate_changed

```
bool is_rate_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed between the last activation and before

last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.10.21 is_delay_changed

```
bool is_delay_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed between the last activation and before last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.10.22 read

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

operator const T& ();

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

The member functions **read**, **operator const T&**, and **operator[]** shall return a reference to the value of a particular sample that is available at the port. 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 functions **read**, **operator const T&**, and **operator[]** shall only be called in the context of the member functions **processing** and **ac_processing** 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 5.3).

5.1.10.23 default_event

```
const sc_event& default_event() const;
```

The member function **default_event** shall return a reference to the default event, which is returned by the member function **default_event** of the channel, to which the port is bound.

5.1.10.24 value_changed_event

```
const sc_event& value_changed_event() const;
```

The member function **value_changed_event** shall return a reference to the value-changed event, which is returned by the member function **value_changed_event** of the channel, to which the port is bound.

5.1.10.25 event

```
bool event() const;
```

The member function **event** shall return the value, which is returned by the member function **event** of the channel, to which the port is bound, at the current module time.

5.1.10.26 bind, operator()

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

virtual 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> >& );

virtual 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 functions **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.

5.1.11 sca_tdf::sca_de::sca_in<bool>, sca_tdf::sc_in<bool>

5.1.11.1 Description

The class **sca_tdf::sca_de::sca_in**<bool> shall define a specialized port class for the TDF MoC. It provides additional member functions appropriate for two-valued signals.

The class **sca_tdf::sc_in**<bool> shall be defined as an alias for class **sca_tdf::sca_de::sca_in**<bool>.

5.1.11.2 Class definition

```
namespace sca_tdf {

  namespace sca_de {

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

      void set_delay( unsigned long );
      void set_rate( unsigned long );
      void set_timestep( const sca_core::sca_time& );
      void set_timestep( double, sc_core::sc_time_unit );
      void set_max_timestep( const sca_core::sca_time& );
      void set_max_timestep( double, sc_core::sc_time_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( unsigned long sample_id = 0 ) const;
      sca_core::sca_time get_max_timestep() const;
      sca_core::sca_time get_last_timestep( unsigned long sample_id = 0 ) const;

      virtual const char* kind() const;

      void initialize( const bool value, unsigned long sample_id = 0 );
      const bool& read_delayed_value( unsigned long sample_id = 0 ) const;

      bool is_timestep_changed( unsigned long sample_id = 0 ) const;
```

```
      bool is_rate_changed() const;
      bool is_delay_changed() const;

      const bool& read( unsigned long sample_id = 0 );
      operator const bool& ();
      const bool& operator[] ( unsigned long sample_id );

      const sc_event& default_event() const;
      const sc_event& value_changed_event() const;
      const sc_event& posedge_event() const;
      const sc_event& negedge_event() const;

      bool event() const;
      bool posedge() const;
      bool negedge() const;

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

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

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

    private:
      // Disabled
      sca_in( const sca_tdf::sca_de::sca_in<bool>& );
      sca_tdf::sca_de::sca_in<bool>& operator= ( const sca_tdf::sca_de::sca_in<bool>& );
    };

  } // namespace sca_de

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

} // namespace sca_tdf
```

5.1.11.3 Constraint on usage

A port of class **sca_tdf::sca_de::sca_in**<bool> and **sca_tdf::sc_in**<bool> shall only be a member of a module derived from class **sca_tdf::sca_module**; otherwise, it shall be an error.

5.1.11.4 Constructors

```
sca_in();

explicit sca_in( const char* );
```

The constructor for class **sca_tdf::sca_de::sca_in**<bool> 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.

```
sc_in() : sca_tdf::sca_de::sca_in<bool>() {}

explicit sc_in( const char* name_ ) : sca_tdf::sca_de::sca_in<bool>( name_ ) {}
```

The constructor for class **sca_tdf::sc_in**<bool> shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sca_tdf::sca_de::sca_in**<bool> 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**.

5.1.11.5 set_delay

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

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 context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.11.6 set_rate

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

The member function **set_rate** shall define the number of samples that can be read during the execution of the member function **processing** of the current TDF module by using 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 context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.11.7 set_timestep

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

void set_timestep( double, sc_core::sc_time_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 5.3). It shall be an error if the function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.11.8 set_max_timestep

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

void set_max_timestep( double, sc_core::sc_time_unit );
```

The member function **set_max_timestep** shall define the maximum timestep between two consecutive samples. If **set_max_timestep** is not called, an implementation shall set the maximum timestep to the value returned by function **sca_core::sca_max_time**. It shall be an error if the member function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.11.9 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.11.10 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.11.11 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 with index *sample_id*. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

The relationship shown in Equation (5.6) shall hold:

$$P.\mathbf{get\_time}(sample\_id) = M.\mathbf{get\_time}() + \frac{M.\mathbf{get\_timestep}() \cdot sample\_id}{P.\mathbf{get\_rate}()} \tag{5.6}$$

where *P* is an instance of a port of class **sca_tdf::sca_de::sca_in**<bool> and *M* is the parent module derived from class **sca_tdf::sca_module** (see 5.3).

NOTE—The relation is valid within the time resolution bound, which is returned by the function **sc_core::sc_get_time_resolution** (see 5.3.1.2).

5.1.11.12 get_timestep

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

The member function **get_timestep** shall return the timestep between the preceding and current sample with index *sample_id*. If the preceding sample is not available, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.11.13 get_max_timestep

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

The member function **get_max_timestep** shall return the maximum timestep between two consecutive samples according to the execution semantics (see 5.3), ignoring the timesteps set for all TDF modules and TDF ports through the member function **set_timestep**. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.11.14 get_last_timestep

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

The member function **get_last_timestep** shall return the timestep between the two samples preceding the sample with index *sample_id*. If the timestep between these two preceding samples is equal to **sc_core::SC_ZERO_TIME**, the member function shall return the last non-zero timestep. If one or both of the preceding samples are not available or if all preceding timesteps are all equal to **sc_core::SC_ZERO_TIME**, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.11.15 kind

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

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

5.1.11.16 initialize

```
void initialize( const bool& 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 functions **initialize** or **reinitialize** of the current TDF 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 the port requires that the port has been assigned a delay using the member function **set_delay**, which shall be called in the member functions **set_attributes** or **change_attributes** of the TDF module.

5.1.11.17 read_delayed_value

```
const bool& read_delayed_value( unsigned long sample_id = 0 ) const;
```

The member function **read_delayed_value** shall return a reference to the value of a delayed sample that is available at the port. The argument *sample_id* denotes the index of the delayed sample being read. The samples shall be indexed from zero to *P*.**get_delay**()–1, where *P* denotes the port. A *sample_id* of zero shall refer to the first delayed sample in time. It shall be an error if *sample_id* is greater than or equal to the port delay.

The member function shall only be called in the member function **reinitialize** of the current TDF module. Otherwise, it shall be an error. Consecutive reads with the same *sample_id* during the same module activation shall return the same delayed value.

5.1.11.18 is_timestep_changed

```
bool is_timestep_changed( unsigned long sample_id = 0 ) const;
```

The member function **is_timestep_changed** shall return true if the timestep of the sample with index *sample_id* of the TDF port, returned by its member function **get_timestep**(*sample_id*), has changed with respect to the preceding sample. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.11.19 is_rate_changed

```
bool is_rate_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed between the last activation and before last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.11.20 is_delay_changed

```
bool is_delay_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed between the last activation and before last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.11.21 read

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

operator const bool& ();

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

The member functions **read**, **operator const bool&**, and **operator[]** shall return a reference to the value of a particular sample that is available at the port. 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 functions **read**, **operator const bool&**, and **operator[]** shall only be called in the context of the member functions **processing** and **ac_processing** 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**<bool>. The member function **read** of the interface proper of class **sc_core::sc_signal_in_if**<bool> shall be called in the evaluation phase at the first delta cycle of the associated time of the sample. (see 5.3).

5.1.11.22 default_event

```
const sc_event& default_event() const;
```

The member function **default_event** shall return a reference to the default event, which is returned by the member function **default_event** of the channel, to which the port is bound.

5.1.11.23 value_changed_event

```
const sc_event& value_changed_event() const;
```

The member function **value_changed_event** shall return a reference to the value-changed event, which is returned by the member function **value_changed_event** of the channel, to which the port is bound.

5.1.11.24 posedge_event

```
const sc_event& posedge_event() const;
```

The member function **posedge_event** shall return a reference to the event, which is returned by the member function **posedge_event** of the channel, to which the port is bound.

5.1.11.25 negedge_event

```
const sc_event& negedge_event() const;
```

The member function **negedge_event** shall return a reference to the event, which is returned by the member function **negedge_event** of the channel, to which the port is bound.

5.1.11.26 event

```
bool event() const;
```

The member function **event** shall return the value, which is returned by the member function **event** of the channel, to which the port is bound, at the current module time.

5.1.11.27 posedge

```
bool posedge() const;
```

The member function **posedge** shall return the value, which is returned by the member function **posedge** of the channel, to which the port is bound, at the current module time.

5.1.11.28 negedge

```
bool negedge() const;
```

The member function **negedge** shall return the value, which is returned by the member function **negedge** of the channel, to which the port is bound, at the current module time.

5.1.11.29 bind, operator()

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

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

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

The member functions **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**<bool> to the channel or port instance passed as an argument.

5.1.12 sca_tdf::sca_de::sca_in<sc_dt::sc_logic>, sca_tdf::sc_in<sc_dt::sc_logic>

5.1.12.1 Description

The class **sca_tdf::sca_de::sca_in**<**sc_dt::sc_logic**> shall define a specialized port class for the TDF MoC. It provides additional member functions appropriate for four-valued signals.

The class **sca_tdf::sc_in**<**sc_dt::sc_logic**> shall be defined as an alias for class **sca_tdf::sca_de::sca_in**<**sc_dt::sc_logic**>.

5.1.12.2 Class definition

```
namespace sca_tdf {

  namespace sca_de {

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

      void set_delay( unsigned long );
      void set_rate( unsigned long );
      void set_timestep( const sca_core::sca_time& );
      void set_timestep( double, sc_core::sc_time_unit );
      void set_max_timestep( const sca_core::sca_time& );
      void set_max_timestep( double, sc_core::sc_time_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( unsigned long sample_id = 0 ) const;
      sca_core::sca_time get_max_timestep() const;
      sca_core::sca_time get_last_timestep( unsigned long sample_id = 0 ) const;

      virtual const char* kind() const;

      void initialize( const sc_dt::sc_logic value, unsigned long sample_id = 0 );
      const sc_dt::sc_logic& read_delayed_value( unsigned long sample_id = 0 ) const;

      bool is_timestep_changed( unsigned long sample_id = 0 ) const;
      bool is_rate_changed() const;
      bool is_delay_changed() const;

      const sc_dt::sc_logic& read( unsigned long sample_id = 0 );
      operator const sc_dt::sc_logic& ();
      const sc_dt::sc_logic& operator[] ( unsigned long sample_id );

      const sc_event& default_event() const;
      const sc_event& value_changed_event() const;
      const sc_event& posedge_event() const;
      const sc_event& negedge_event() const;

      bool event() const;
      bool posedge() const;
      bool negedge() const;

      virtual void bind( sc_core::sc_signal_in_if<sc_dt::sc_logic>& );
      void operator() ( sc_core::sc_signal_in_if<sc_dt::sc_logic>& );

      virtual void bind( sc_core::sc_port<sc_core::sc_signal_in_if<sc_dt::sc_logic> >& );
      void operator() ( sc_core::sc_port<sc_core::sc_signal_in_if<sc_dt::sc_logic> >& );
```

```
      virtual void bind( sc_core::sc_port<sc_core::sc_signal_inout_if<sc_dt::sc_logic> >& );
      void operator() ( sc_core::sc_port<sc_core::sc_signal_inout_if<sc_dt::sc_logic> >& );

     private:
      // Disabled
      sca_in( const sca_tdf::sca_de::sca_in<sc_dt::sc_logic>& );
      sca_tdf::sca_de::sca_in<sc_dt::sc_logic>& operator= (
        const sca_tdf::sca_de::sca_in<sc_dt::sc_logic>& );
    };

  } // namespace sca_de

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

} // namespace sca_tdf
```

5.1.12.3 Constraint on usage

A port of class **sca_tdf::sca_de::sca_in<sc_dt::sc_logic>** and **sca_tdf::sc_in<sc_dt::sc_logic>** shall only be a member of a module derived from class **sca_tdf::sca_module**; otherwise, it shall be an error.

5.1.12.4 Constructors

```
sca_in();

explicit sca_in( const char* );
```

The constructor for class **sca_tdf::sca_de::sca_in<sc_dt::sc_logic>** 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.

```
sc_in() : sca_tdf::sca_de::sca_in<sc_dt::sc_logic>() {}

explicit sc_in( const char* name_ ) : sca_tdf::sca_de::sca_in<sc_dt::sc_logic>( name_ ) {}
```

The constructor for class **sca_tdf::sc_in<sc_dt::sc_logic>** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sca_tdf::sca_de::sca_in<sc_dt::sc_logic>** 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**.

5.1.12.5 set_delay

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

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 context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.12.6 set_rate

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

The member function **set_rate** shall define the number of samples that can be read during the execution of the member function **processing** of the current TDF module by using 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 context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.12.7 set_timestep

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

void set_timestep( double, sc_core::sc_time_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 5.3). It shall be an error if the function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.12.8 set_max_timestep

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

void set_max_timestep( double, sc_core::sc_time_unit );
```

The member function **set_max_timestep** shall define the maximum timestep between two consecutive samples. If **set_max_timestep** is not called, an implementation shall set the maximum timestep to the value returned by function **sca_core::sca_max_time**. It shall be an error if the member function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.12.9 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.12.10 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.12.11 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 with index *sample_id*. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

The relationship shown in Equation (5.7) shall hold:

$$P.\textbf{get\_time}(sample\_id) = M.\textbf{get\_time}() + \frac{M.\textbf{get\_timestep}() \cdot sample\_id}{P.\textbf{get\_rate}()} \tag{5.7}$$

where *P* is an instance of a port of class **sca_tdf::sca_de::sca_in**<**sc_dt::sc_logic**> and *M* is the parent module derived from class **sca_tdf::sca_module** (see 5.3).

NOTE—The relation is valid within the time resolution bound, which is returned by the function **sc_core::sc_get_time_resolution** (see 5.3.1.2).

5.1.12.12 get_timestep

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

The member function **get_timestep** shall return the timestep between the preceding and current sample with index *sample_id*. If the preceding sample is not available, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.12.13 get_max_timestep

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

The member function **get_max_timestep** shall return the maximum timestep between two consecutive samples according to the execution semantics (see 5.3), ignoring the timesteps set for all TDF modules and TDF ports through the member function **set_timestep**. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.12.14 get_last_timestep

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

The member function **get_last_timestep** shall return the timestep between the two samples preceding the sample with index *sample_id*. If the timestep between these two preceding samples is equal to **sc_core::SC_ZERO_TIME**, the member function shall return the last non-zero timestep. If one or both of the preceding samples are not available or if all preceding timesteps are all equal to **sc_core::SC_ZERO_TIME**, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.12.15 kind

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

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

5.1.12.16 initialize

```
void initialize( const sc_dt::sc_logic& 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 functions **initialize** or **reinitialize** of the current TDF 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 the port requires that the port has been assigned a delay using the member function **set_delay**, which shall be called in the member functions **set_attributes** or **change_attributes** of the TDF module.

5.1.12.17 read_delayed_value

```
const sc_dt::sc_logic& read_delayed_value( unsigned long sample_id = 0 ) const;
```

The member function **read_delayed_value** shall return a reference to the value of a delayed sample that is available at the port. The argument *sample_id* denotes the index of the delayed sample being read. The samples shall be indexed from zero to *P*.**get_delay**()–1, where *P* denotes the port. A *sample_id* of zero shall refer to the first delayed sample in time. It shall be an error if *sample_id* is greater than or equal to the port delay.

The member function shall only be called in the member function **reinitialize** of the current TDF module. Otherwise, it shall be an error. Consecutive reads with the same *sample_id* during the same module activation shall return the same delayed value.

5.1.12.18 is_timestep_changed

```
bool is_timestep_changed( unsigned long sample_id = 0 ) const;
```

The member function **is_timestep_changed** shall return true if the timestep of the sample with index *sample_id* of the TDF port, returned by its member function **get_timestep**(*sample_id*), has changed with respect to the preceding sample. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.12.19 is_rate_changed

```
bool is_rate_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed between the last activation and before last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.12.20 is_delay_changed

```
bool is_delay_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed between the last activation and before last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.12.21 read

```
const sc_dt::sc_logic& read( unsigned long sample_id = 0 );

operator const sc_dt::sc_logic& ();

const sc_dt::sc_logic& operator[] ( unsigned long sample_id );
```

The member functions **read**, **operator const sc_dt::sc_logic&** and **operator[]** shall return a reference to the value of a particular sample that is available at the port. 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 functions **read**, **operator const sc_dt::sc_logic&**, and **operator[]** shall only be called in the context of the member functions **processing** and **ac_processing** 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<sc_dt::sc_logic>**. The member function **read** of the interface proper of class **sc_core::sc_signal_in_if<sc_dt::sc_logic>** shall be called in the evaluation phase at the first delta cycle of the associated time of the sample. (see 5.3).

5.1.12.22 default_event

```
const sc_event& default_event() const;
```

The member function **default_event** shall return a reference to the default event, which is returned by the member function **default_event** of the channel, to which the port is bound.

5.1.12.23 value_changed_event

```
const sc_event& value_changed_event() const;
```

The member function **value_changed_event** shall return a reference to the value-changed event, which is returned by the member function **value_changed_event** of the channel, to which the port is bound.

5.1.12.24 posedge_event

```
const sc_event& posedge_event() const;
```

The member function **posedge_event** shall return a reference to the event, which is returned by the member function **posedge_event** of the channel, to which the port is bound.

5.1.12.25 negedge_event

```
const sc_event& negedge_event() const;
```

The member function **negedge_event** shall return a reference to the event, which is returned by the member function **negedge_event** of the channel, to which the port is bound.

5.1.12.26 event

```
bool event() const;
```

The member function **event** shall return the value, which is returned by the member function **event** of the channel, to which the port is bound, at the current module time.

5.1.12.27 posedge

```
bool posedge() const;
```

The member function **posedge** shall return the value, which is returned by the member function **posedge** of the channel, to which the port is bound, at the current module time.

5.1.12.28 negedge

```
bool negedge() const;
```

The member function **negedge** shall return the value, which is returned by the member function **negedge** of the channel, to which the port is bound, at the current module time.

5.1.12.29 bind, operator()

```
virtual void bind( sc_core::sc_signal_in_if<sc_dt::sc_logic>& );
void operator() ( sc_core::sc_signal_in_if<sc_dt::sc_logic>& );

virtual void bind( sc_core::sc_port<sc_core::sc_signal_in_if<sc_dt::sc_logic> >& );
void operator() ( sc_core::sc_port<sc_core::sc_signal_in_if<sc_dt::sc_logic> >& );

virtual void bind( sc_core::sc_port<sc_core::sc_signal_inout_if<sc_dt::sc_logic> >& );
void operator() ( sc_core::sc_port<sc_core::sc_signal_inout_if<sc_dt::sc_logic> >& );
```

The member functions **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<sc_dt::sc_logic>** to the channel or port instance passed as an argument.

5.1.13 sca_tdf::sca_de::sca_out, sca_tdf::sc_out

5.1.13.1 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 5.3.2.4).

The class **sca_tdf::sc_out** shall be defined as an alias for class **sca_tdf::sca_de::sca_out**.

5.1.13.2 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* );

      void set_delay( unsigned long );
      void set_rate( unsigned long );
      void set_timestep( const sca_core::sca_time& );
      void set_timestep( double, sc_core::sc_time_unit );
      void set_max_timestep( const sca_core::sca_time& );
      void set_max_timestep( double, sc_core::sc_time_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( unsigned long sample_id = 0 ) const;
      sca_core::sca_time get_max_timestep() const;
      sca_core::sca_time get_last_timestep( unsigned long sample_id = 0 ) const;

      virtual const char* kind() const;

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

      bool is_timestep_changed( unsigned long sample_id = 0 ) const;
      bool is_rate_changed() const;
      bool is_delay_changed() const;

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

5.1.13.4 Constraint on usage

A port of class **sca_tdf::sca_de::sca_out** and **sca_tdf::sc_out** shall only be a member of a module derived from class **sca_tdf::sca_module**; otherwise, it shall be an error.

5.1.13.3 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 not defined by this standard. The implementation shall use this operator for writing trace values in time-domain simulation (see 9.1).

```
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 shall 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 or reading from ports of type **T**.

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

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

5.1.13.5 Constructors

```
sca_out();

explicit sca_out( const char* );
```

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.

```
sc_out() : sca_tdf::sca_de::sca_out<T>() {}

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

The constructor for class **sca_tdf::sc_out** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sca_tdf::sca_de::sca_out** 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**.

5.1.13.6 set_delay

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

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 context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.13.7 set_rate

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

The member function **set_rate** shall define the number of samples that can be written during the execution of the member function **processing** of the current TDF module by using 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 context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.13.8 set_timestep

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

void set_timestep( double, sc_core::sc_time_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 5.3). It shall be an error if the member function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.13.9 set_max_timestep

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

void set_max_timestep( double, sc_core::sc_time_unit );
```

The member function **set_max_timestep** shall define the maximum timestep between two consecutive samples. If **set_max_timestep** is not called, an implementation shall set the maximum timestep to the value returned by function **sca_core::sca_max_time**. It shall be an error if the member function is called outside the context of the member functions **set_attributes** or **change_attributes** of the current TDF module.

5.1.13.10 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.13.11 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 if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.13.12 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 with index *sample_id*. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

The relationship shown in Equation (5.8) shall hold:

$$P.\mathbf{get\_time}(sample\_id) = M.\mathbf{get\_time}() + \frac{M.\mathbf{get\_timestep}() \cdot sample\_id}{P.\mathbf{get\_rate}()} \tag{5.8}$$

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 5.3).

NOTE—The relation is valid within the time resolution bound, which is returned by the function **sc_core::sc_get_time_resolution** (see 5.3.1.2).

5.1.13.13 get_timestep

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

The member function **get_timestep** shall return the timestep between the preceding and current sample with index *sample_id*. If the preceding sample is not available, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.13.14 get_max_timestep

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

The member function **get_max_timestep** shall return the maximum timestep between two consecutive samples according to the execution semantics (see 5.3), ignoring the timesteps set for all TDF modules and TDF ports through the member function **set_timestep**. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.13.15 get_last_timestep

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

The member function **get_last_timestep** shall return the timestep between the two samples preceding the sample with index *sample_id*. If the timestep between these two preceding samples is equal to **sc_core::SC_ZERO_TIME**, the member function shall return the last non-zero timestep. If one or both of the preceding samples are not available or if all preceding timesteps are all equal to **sc_core::SC_ZERO_TIME**, the member function shall return the propagated timestep (see 5.3.1.2). It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.13.16 kind

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

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

5.1.13.17 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 functions **initialize** or **reinitialize** of the current TDF 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 the port requires that the port has been assigned a delay using the member function **set_delay**, which shall be called in the member functions **set_attributes** or **change_attributes** of the TDF module.

5.1.13.18 initialize_de_signal

```
void initialize_de_signal( const T& );
```

The member function **initialize_de_signal** shall set the initial value of the signal, to which the port is bound, by calling member function **write** of that signal using the value passed as an argument to member function **initialize_de_signal**. The port need not have been bound at the point during elaboration when member function **initialize_de_signal** is called. In this case, the implementation shall defer the call to **write** until after the port has been bound and the identity of the signal is known. It shall be an error to call this member function after the elaboration phase has finished.

5.1.13.19 read_delayed_value

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

The member function **read_delayed_value** shall return a reference to the value of a delayed sample that is available at the port. The argument *sample_id* denotes the index of the delayed sample being read. The samples shall be indexed from zero to *P*.**get_delay**()–1, where *P* denotes the port. A *sample_id* of zero shall refer to the first delayed sample in time. It shall be an error if *sample_id* is greater than or equal to the port delay.

The member function shall only be called in the member function **reinitialize** of the current TDF module. Otherwise, it shall be an error. Consecutive reads with the same *sample_id* during the same module activation shall return the same delayed value.

5.1.13.20 is_timestep_changed

```
bool is_timestep_changed( unsigned long sample_id = 0 ) const;
```

The member function **is_timestep_changed** shall return true if the timestep of the sample with index *sample_id* of the TDF port, returned by its member function **get_timestep**(*sample_id*), has changed with respect to the preceding sample. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.13.21 is_rate_changed

```
bool is_rate_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_rate_changed** shall return true if the rate of the TDF port has changed between the last activation and before last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.13.22 is_delay_changed

```
bool is_delay_changed() const;
```

In the context of the callbacks **processing**, **ac_processing**, and **reinitialize**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed since the last activation of the callback **processing** of the current TDF module. In the context of the callback **change_attributes**, the member function **is_delay_changed** shall return true if the delay of the TDF port has changed between the last activation and before last activation of the callback **processing** of the current TDF module. Otherwise, it shall return false. It shall be an error if the member function is called outside the context of the member functions **initialize**, **reinitialize**, **processing**, **ac_processing**, or **change_attributes** of the current TDF module.

5.1.13.23 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 functions **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= ( sca_tdf::sca_de::sca_in<T>& );
```

The **operator=** shall read the first value from the input port of class **sca_tdf::sca_in** or **sca_tdf::sca_de::sca_in** and write it to the first value of 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 functions **write** and **operator=** shall write the value made available through the object of class **sca_core::***sca_assign_from_proxy*† to the output port.

The member functions **write**, **operator=**, and **operator[]** shall only be called in the context of the member function **processing** 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 5.3).

5.1.14 sca_tdf::sca_trace_variable

5.1.14.1 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**.

5.1.14.2 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* );

    virtual const char* kind() const;

    void write( const T& );
    const T& read();
    operator const T& ();
    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= ( sca_tdf::sca_de::sca_in<T>& port );
  };

} // namespace sca_tdf
```

5.1.14.3 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**.

5.1.14.4 Constructors

```
sca_trace_variable();

explicit sca_trace_variable( const char* );
```

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**.

5.1.14.5 kind

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

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

5.1.14.6 write

```
void write( const T& );

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

The member functions **write** and **operator=** shall write one sample to the trace variable. The member functions shall only be called in the context of the member function **processing** of the current module; otherwise, it shall be an error.

5.1.14.7 read

```
const T& read();

operator const T& ();
```

The member functions **read** and **operator const T&** shall return a reference to the trace variable. The member functions shall only be called in the context of the member functions **processing** and **ac_processing** of the current module; otherwise, it shall be an error.

5.2 Hierarchical 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 1666-2011 as well as the following specific rules as defined in this subclause. Otherwise, it shall be an error.

a) 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.

b) 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.

c) 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.

d) 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.

e) 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.

f) 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.

5.3 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**, **processing**, **change_attributes**, and **reinitialize** 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.

5.3.1 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 (see 3.1.4).

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.

5.3.1.1 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**.

5.3.1.2 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 propagated timestep of a module (T_m) derived from class **sca_tdf::sca_module** shall be consistent with the rate (R) and the propagated timestep of any port (T_p) derived from class **sca_tdf::sca_in**, **sca_tdf::sca_out**, **sca_tdf::sca_de::sca_in** or **sca_tdf::sca_de::sca_out** within that module, according to Equation (5.9):

$$T_m = T_p \cdot R \tag{5.9}$$

The maximum timesteps, set by member function **set_max_timestep**, shall be propagated to all TDF modules and TDF ports in the cluster according to Equation (5.9). The maximum timestep shall be resolved by taking the smallest propagated maximum timestep of the TDF modules in the cluster. In case none of the TDF modules in the cluster specifies the maximum timestep smaller than the time returned by function **sca_core::sca_max_time**, the largest propagated maximum timestep in the cluster shall be equal to the time returned by function **sca_core::sca_max_time**. If in a TDF cluster no timestep has been assigned using member function **set_timestep** of a TDF module or TDF port, the propagated maximum timestep shall be used as the propagated timestep. It shall be an error if the propagated maximum timestep is smaller than the propagated timestep in the cluster.

In case the TDF attributes are changed and the Equation (5.9) cannot be satisfied in consequence to this change, an implementation shall at least satisfy the Equation (5.9) for the TDF modules in the cluster, which contain the TDF ports with the smallest timestep.

The timestep of a module, returned by the member function **get_timestep**, shall be equal to the time between the last and current activation of the callback **processing**, except for the first module activation, where it shall return the propagated value.

The timestep shall be only updated immediately before the execution of the callbacks **initialize**, **processing**, and **reinitialize**. If the difference between two timesteps is smaller than or equal to the time resolution, as returned by the function **sc_core::sc_get_time_resolution**, the timesteps are considered to be indistinguishable. In case a timestep change is imposed, as a result of the use of the member functions **set_timestep** or **set_max_timestep**, the time at which the next cluster execution period starts shall be equal to the current module time plus the smallest propagated timestep in the cluster.

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; otherwise it shall be an error. It shall be an error if the propagated timestep is equal to the time returned by function **sca_core::sca_max_time**.

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 equal to the current module activation time.

NOTE—An application needs to assign at least one timestep, using member functions **set_timestep** or **set_max_timestep**, 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. If no timestep has been assigned, the propagated timestep becomes equal to the time returned by function **sca_core::sca_max_time**, which shall result in an error (see 4.2.6).

5.3.1.3 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 $r_{M_1}, r_{M_2}, \ldots, r_{M_N}$, which 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:

a) 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**, Equation (5.10) shall hold:

$$r_{M_i} \cdot P_i.\textbf{get\_rate}() = r_{M_j} \cdot P_j.\textbf{get\_rate}() \tag{5.10}$$

b) For each cluster, there exists an order of activation of the TDF modules that fulfills the activation conditions as defined in 5.3.2.2, such that each TDF module M_i shall be activated exactly r_{M_i} times.

c) For each cluster, there exists an activation order of modules where the time stamps of samples read from ports of class **sca_tdf::sca_de::sca_in** at a particular module activation are smaller than or equal to the time stamps of samples written to ports of class **sca_tdf::sca_de::sca_out** at later scheduled module activations.

5.3.2 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.

5.3.2.1 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**, immediately before the first call to the first scheduled member function **processing**. The current module time, returned by member function **get_time** in the context of the member function **initialize**, shall be equal to the time of the first module activation.

The initial sample values at ports with an associated delay greater than zero are defined by the execution of the port member function **initialize**, which shall be called in the TDF module callback **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 to the output port while executing the member function **processing** (see 5.1.7.15, 5.1.8.18, and 5.1.9.18).

5.3.2.2 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's 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 5.1.5.16 and 5.1.10.17).

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.

5.3.2.3 Attribute changes and reinitialization

The TDF attributes can be changed during simulation in the context of the member function **change_attributes** of the modules derived from class **sca_tdf::sca_module**. The member functions **change_attributes** of each TDF module, which belong to the same cluster, shall be executed, in no particular order, after each cluster execution period. The cluster execution period shall be the shortest possible periodic cycle of module activations to fulfill the execution semantics (see 5.3.2.2). The current module time, returned by member function **get_time** in the context of the member function **change_attributes**, shall be equal to the highest annotated time to any of the samples during the cluster execution period.

After execution of all member functions **change_attributes**, an implementation shall check the consistency of timesteps and rates and perform timestep calculation and propagation (see 5.3.1.2 and 5.3.1.3). This is followed by the execution of the member functions **reinitialize**. The current module time, returned by member function **get_time** in the context of member function **reinitialize**, shall be equal to the time of the next module activation.

If the delay of a port is changed using the member function **set_delay** in the member function **change_attributes**, the following rules shall be applied:

— If the delay of a port is decreased, the number of delayed samples shall be reduced to the new port delay, starting with removing the oldest samples.

— If the delay of a port is increased, the number of delayed samples shall be increased, where the new delay samples shall be added before the available samples. By default, the initial delay values are defined by the default constructor of the corresponding data type. The member function **initialize** may be called in the callback **reinitialize** to initialize these new delay values.

5.3.2.4 Synchronization with the 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**.

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 shall be available (see 5.1.10.22).

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 5.1.13.23).

5.4 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:

a) The timestep returned by the member function **get_timestep** of the parent module of class **sca_tdf::sca_module**.

b) The timestep returned by the member function **get_timestep** of an instance of class **sca_tdf::sca_in** or **sca_tdf::sca_de::sca_in**, which is passed as an argument to the linear dynamic equations.

c) The timestep passed as an argument to the linear dynamic equations.

If a timestep value is defined for the equations, the timestep value shall be smaller than 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 timestep shall be extended by the difference between these two times.

If the time of the first discrete time point of the current calculation is equal or smaller than the time of the last discrete time point of the previous calculation, the state of the equation system shall be restored to the state at the last time point of the calculation before the last calculation.

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 **start_of_simulation**. 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.

5.4.1 sca_tdf::*sca_ct_proxy*†

5.4.1.1 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 5.4.3.9, 5.4.4.9).

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

  };

} // namespace sca_tdf
```

5.4.1.3 Constraint on usage

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

5.4.1.4 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.

5.4.1.5 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. At zero time (t=0), the values written to the vector shall be equal to the initial calculated value at t=0.

5.4.1.6 to_port

```
void to_port( sca_tdf::sca_out_base<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 is written to the corresponding samples of the port. It shall be an error if the time associated with the last output sample is larger than the time associated with the input sample plus the continuous-time delay.

5.4.1.7 assign_to

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

void assign_to( sca_tdf::sca_out_base<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**.

5.4.2 sca_tdf::*sca_ct_vector_proxy*†

5.4.2.1 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 5.4.5.8).

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

} // namespace sca_tdf
```

5.4.2.3 Constraint on usage

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

5.4.2.4 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.

5.4.2.5 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. At zero time (t=0), the values written to the matrix shall be equal to the initial calculated value at t=0.

5.4.2.6 to_port

```
void to_port( sca_tdf::sca_out_base< 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 is written to the corresponding samples of the port. It shall be an error if the time associated with the last output sample is larger than the time associated with the input sample plus the continuous-time delay.

5.4.2.7 assign_to

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

void assign_to( sca_tdf::sca_out_base<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**.

5.4.3 sca_tdf::sca_ltf_nd

5.4.3.1 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 shown in Equation (5.11):

$$H(s) = k \cdot \frac{\sum_{i=0}^{M-1} num_i \cdot s^i}{\sum_{i=0}^{N-1} den_i \cdot s^i} \cdot e^{(-s \cdot delay)} \tag{5.11}$$

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 continuous-time delay applied to the values available at the input.

5.4.3.2 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, sc_core::sc_time_unit );

    double estimate_next_value() const;

    void enable_iterations();

    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,
                                       const sca_core::sca_time& tstep = sca_core::sca_max_time() );

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

    sca_tdf::sca_ct_proxy†& calculate( const sca_util::sca_vector<double>& num,
                                       const sca_util::sca_vector<double>& den,
```

```
                                    const sca_core::sca_time& delay,
                                          sca_util::sca_vector<double>& state,
                                    const sca_util::sca_vector<double>& input,
                                          double                        k = 1.0,
                                    const sca_core::sca_time& tstep = sca_core::sca_max_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,
                                    const sca_core::sca_time& delay,
                                          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,
                                    const sca_core::sca_time& delay,
                                          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,
                                    const sca_core::sca_time& tstep = sca_core::sca_max_time() );

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

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

sca_tdf::sca_ct_proxy<T>& 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,
                                  const sca_core::sca_time& tstep = sca_core::sca_max_time() );

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

sca_tdf::sca_ct_proxy<T>& 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,
                                  const sca_core::sca_time& tstep = sca_core::sca_max_time() );

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

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

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

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

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

sca_tdf::sca_ct_proxy<T>& 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,
                                    const sca_core::sca_time& tstep = sca_core::sca_max_time() );

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

} // namespace sca_tdf
```

5.4.3.3 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**.

5.4.3.4 Constraint on usage

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

5.4.3.5 kind

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

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

5.4.3.6 set_max_delay

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

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

The member function **set_max_delay** shall define the maximum allowable continuous-time 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 context of the member function **set_attributes** of the current TDF module (see 5.1.1.5).

5.4.3.7 estimate_next_value

```
double estimate_next_value() const;
```

The member function **estimate_next_value** shall return an estimation of the value of type double one timestep ahead. This timestep shall be equal to the last computed non-zero timestep. The estimation shall use the same filter coefficients as provided for the last calculation of the equation system. The accuracy of the estimated next value is implementation-defined.

NOTE 1—An implementation may give a warning in case the provided filter coefficients could lead to an inaccurate estimation of the next value.

NOTE 2—If the number of numerator coefficients is equal or larger than the number of denumerator coefficients, an implementation may give an inaccurate estimation of the next value.

5.4.3.8 enable_iterations

```
void enable_iterations();
```

The member function **enable_iterations** shall enable zero timestep recalculations by means of setting the timestep argument *tstep* of the member function **calculate** and **operator()** to **sc_core::SC_ZERO_TIME**. It shall be called during elaboration; otherwise it shall be an error.

5.4.3.9 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 computation of the solution of the equation system shall be started at the time of the first module activation.

The arguments include the gain of the transfer function *k*, the vectors of the numerator coefficients *num* and denominator coefficients *den*, the continuous-time 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 continuous-time 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 continuous-time 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. In case where states are stored internally and the member function **get_timestep** returns **sc_core::SC_ZERO_TIME** for

the first calculation of the equation system of the current module activation, the state vector is restored to the values before the last module activation. If the size of the state vector is zero, its size shall be defined by the member functions **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 returned by function **sca_core::sca_max_time**, the member functions **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 than 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 the timestep *tstep* is set to the value **sc_core::SC_ZERO_TIME** and the internal time of the equation system has progressed to the same point in time equal to the current module activation returned by member function **get_time**, the time and the state of the equation system shall be restored to the time and state before the last calculation, and shall use the new input values. It shall be an error if the timestep *tstep* is equal to **sc_core::SC_ZERO_TIME** and the member function **enable_iterations** has not been called before the callback **start_of_simulation**.

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.

5.4.4 sca_tdf::sca_ltf_zp

5.4.4.1 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 shown in Equation (5.12):

$$H(s) = k \cdot \frac{\prod_{i=0}^{M-1}(s - zeros_i)}{\prod_{i=0}^{N-1}(s - poles_i)} \cdot e^{(-s \cdot delay)} \tag{5.12}$$

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 continuous-time delay applied to the values available at the input.

5.4.4.2 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, sc_core::sc_time_unit );

  double estimate_next_value() const;

  void enable_iterations();

  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,
                          const sca_core::sca_time& tstep = sca_core::sca_max_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_core::sca_time& delay,
                                sca_util::sca_vector<double>& state,
                                double                        input,
                                double                        k = 1.0,
                          const sca_core::sca_time& tstep = sca_core::sca_max_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,
                          const sca_core::sca_time& tstep = sca_core::sca_max_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_core::sca_time& delay,
                                sca_util::sca_vector<double>& state,
                          const sca_util::sca_vector<double>& input,
                                double                        k = 1.0,
                          const sca_core::sca_time& tstep = sca_core::sca_max_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,
                          const sca_core::sca_time& delay,
                                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,
                          const sca_core::sca_time& delay,
```

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

sca_tdf::sca_ct_proxy<T>& 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,
                        const sca_core::sca_time& tstep = sca_core::sca_max_time() );

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

sca_tdf::sca_ct_proxy<T>& 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,
                        const sca_core::sca_time& tstep = sca_core::sca_max_time() );

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

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

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

sca_tdf::sca_ct_proxy<T>& 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,
                        const sca_core::sca_time& tstep = sca_core::sca_max_time() );

sca_tdf::sca_ct_proxy<T>& operator() (
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                        const sca_core::sca_time& delay,
                              sca_util::sca_vector<double>& state,
                              double                        input,
                              double                        k = 1.0,
```

```
                        const sca_core::sca_time& tstep = sca_core::sca_max_time() );

sca_tdf::sca_ct_proxy<T>& 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,
                        const sca_core::sca_time& tstep = sca_core::sca_max_time() );

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

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

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

sca_tdf::sca_ct_proxy<T>& 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,
                        const sca_core::sca_time& tstep = sca_core::sca_max_time() );

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

sca_tdf::sca_ct_proxy<T>& 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,
                        const sca_core::sca_time& tstep = sca_core::sca_max_time() );

sca_tdf::sca_ct_proxy<T>& operator() (
                        const sca_util::sca_vector<sca_util::sca_complex>& zeros,
                        const sca_util::sca_vector<sca_util::sca_complex>& poles,
                        const sca_core::sca_time& delay,
```

```
                            const sca_util::sca_vector<double>& input,
                                  double                        k = 1.0,
                            const sca_core::sca_time& tstep = sca_core::sca_max_time() );

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

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

} // namespace sca_tdf
```

5.4.4.3 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**.

5.4.4.4 Constraint on usage

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

5.4.4.5 kind

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

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

5.4.4.6 set_max_delay

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

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

The member function **set_max_delay** shall define the maximum allowable continuous-time 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 context of the member function **set_attributes** of the current TDF module (see 5.1.1.5).

5.4.4.7 estimate_next_value

```
double estimate_next_value() const;
```

The member function **estimate_next_value** shall return an estimation of the value of type double one timestep ahead. This timestep shall be equal to the last computed non-zero timestep. The estimation shall use the same filter coefficients as provided for the last calculation of the equation system. The accuracy of the estimated next value is implementation-defined.

NOTE 1—An implementation may give a warning in case the provided filter coefficients could lead to an inaccurate estimation of the next value.

NOTE 2—If the number of zeros is equal or larger than the number of poles, an implementation may give an inaccurate estimation of the next value.

5.4.4.8 enable_iterations

```
void enable_iterations();
```

The member function **enable_iterations** shall enable zero timestep recalculations by means of setting the timestep argument *tstep* of the member function **calculate** and **operator()** to **sc_core::SC_ZERO_TIME**. It shall be called during elaboration; otherwise it shall be an error.

5.4.4.9 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 computation of the solution of the equation system shall be started at the time of the first module activation.

The arguments include the gain of the transfer function *k*, the vectors of the zero coefficients *zeros* and pole coefficients *poles*, the continuous-time 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 shown in Equation (5.13):

$$1.0 + 0.0\,j \tag{5.13}$$

The argument *delay* specifies the continuous-time 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 continuous-time 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. In case where states are stored internally and the member function **get_timestep** returns **sc_core::SC_ZERO_TIME** for the first calculation of the equation system of the current module activation, the state vector is restored to the values before the last module activation. If the size of the state vector is zero, its size shall be defined by the member functions **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 returned by function **sca_core::sca_max_time**, the member functions **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 than 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 the timestep *tstep* is set to the value **sc_core::SC_ZERO_TIME** and the internal time of the equation system has progressed to the same point in time equal to the current module activation returned by member function **get_time**, the time and the state of the equation system shall be restored to the time and state before the last calculation, and shall use the new input values. It shall be an error if the timestep *tstep* is equal to **sc_core::SC_ZERO_TIME** and the member function **enable_iterations** has not been called before the callback **start_of_simulation**.

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.

5.4.5 sca_tdf::sca_ss

5.4.5.1 Description

The class **sca_tdf::sca_ss** shall implement a system, which behavior is defined by the state-space equations Equation (5.14) and Equation (5.15):

$$\frac{ds(t)}{dt} = \mathbf{A}\cdot s(t) + \mathbf{B}\cdot x(t - delay) \tag{5.14}$$

$$y(t) = \mathbf{C}\cdot s(t) + \mathbf{D}\cdot x(t - delay) \tag{5.15}$$

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 continuous-time 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.

5.4.5.2 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, sc_core::sc_time_unit );

    sca_util::sca_vector<double> estimate_next_value() const;

    void enable_iterations();


    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)
                                  const sca_core::sca_time& tstep = sca_core::sca_max_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
                                  const sca_core::sca_time& delay,
                                        sca_util::sca_vector<double>& s, // state vector s(t)
                                  const sca_util::sca_vector<double>& x, // input vector x(t)
                                  const sca_core::sca_time& tstep = sca_core::sca_max_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,
                                        const sca_core::sca_time& tstep = sca_core::sca_max_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_core::sca_time& delay,
                                              sca_util::sca_vector<double>& s,
                                        const sca_util::sca_matrix<double>& x,
                                        const sca_core::sca_time& tstep = sca_core::sca_max_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,
                                const sca_core::sca_time& delay,
                                      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,
                                const sca_core::sca_time& delay,
                                      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,
                                const sca_core::sca_time& tstep = sca_core::sca_max_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_core::sca_time& delay,
                                const sca_util::sca_vector<double>& x,
                                const sca_core::sca_time& tstep = sca_core::sca_max_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,
                                const sca_core::sca_time& tstep = sca_core::sca_max_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_core::sca_time& delay,
                                const sca_util::sca_matrix<double>& x,
                                const sca_core::sca_time& tstep = sca_core::sca_max_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,
                                const sca_core::sca_time& delay,
                                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,
                                const sca_core::sca_time& delay,
                                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,
                                const sca_core::sca_time& tstep = sca_core::sca_max_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_core::sca_time& delay,
                                      sca_util::sca_vector<double>& s,
                                const sca_util::sca_vector<double>& x,
                                const sca_core::sca_time& tstep = sca_core::sca_max_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,
                                const sca_core::sca_time& tstep = sca_core::sca_max_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_core::sca_time& delay,
                                      sca_util::sca_vector<double>& s,
                                const sca_util::sca_matrix<double>& x,
                                const sca_core::sca_time& tstep = sca_core::sca_max_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,
                                const sca_core::sca_time& delay,
                                      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,
                                    const sca_core::sca_time& delay,
                                          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,
                                    const sca_core::sca_time& tstep = sca_core::sca_max_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_core::sca_time& delay,
                                    const sca_util::sca_vector<double>& x,
                                    const sca_core::sca_time& tstep = sca_core::sca_max_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,
                                    const sca_core::sca_time& tstep = sca_core::sca_max_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_core::sca_time& delay,
                                    const sca_util::sca_matrix<double>& x,
                                    const sca_core::sca_time& tstep = sca_core::sca_max_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,
                                    const sca_core::sca_time& delay,
                                    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,
                                    const sca_core::sca_time& delay,
                                    const sca_tdf::sca_de::sca_in< sca_util::sca_vector<double> >& x );
};
```

```
} // namespace sca_tdf
```

5.4.5.3 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**.

5.4.5.4 kind

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

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

5.4.5.5 set_max_delay

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

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

The member function **set_max_delay** shall define the maximum allowable continuous-time 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 context of the member function **set_attributes** of the current TDF module (see 5.1.1.5).

5.4.5.6 estimate_next_value

```
sca_util::sca_vector<double> estimate_next_value() const;
```

The member function **estimate_next_value** shall return an estimation of the value of type **sca_util::sca_vector**<double> one timestep ahead. This timestep shall be equal to the last computed non-zero timestep. The estimation shall use the same filter coefficients as provided for the last calculation of the equation system. The accuracy of the estimated next value is implementation-defined.

NOTE 1—An implementation may give a warning in case the provided filter coefficients could lead to an inaccurate estimation of the next value.

NOTE 2—If any of the values in matrix **D** of the state-space equation system is non-zero, an implementation may give an inaccurate estimation of the next value.

5.4.5.7 enable_iterations

```
void enable_iterations();
```

The member function **enable_iterations** shall enable zero timestep recalculations by means of setting the timestep argument *tstep* of the member function **calculate** and **operator()** to **sc_core::SC_ZERO_TIME**. It shall be called during elaboration; otherwise it shall be an error.

5.4.5.8 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 computation of the solution of the equation system shall be started at the time of the first module activation.

The arguments include the matrices *a*, *b*, *c*, and *d*, the continuous-time 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:

a) Argument *a* shall be a square matrix of the size of state vector *s*.

b) 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*.

c) 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*.

d) 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 continuous-time 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 continuous-time 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. In case where states are stored internally and the member function **get_timestep** returns **sc_core::SC_ZERO_TIME** for the first calculation of the equation system of the current module activation, the state vector is restored to the values before the last module activation. If the size of the state vector is zero, its size shall be defined by the member functions **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 returned by function **sca_core::sca_max_time**, the member functions **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 than 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 the timestep *tstep* is set to the value **sc_core::SC_ZERO_TIME** and the internal time of the equation system has progressed to the same point in time equal to the current module activation returned by member function **get_time**, the time and the state of the equation system shall be restored to the time and state before the last calculation, and shall use the new input values. It shall be an error if the timestep *tstep* is equal to **sc_core::SC_ZERO_TIME** and the member function **enable_iterations** has not been called before the callback **start_of_simulation**.

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.

6. 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 clause.

6.1 Class definitions

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

6.1.1 sca_lsf::sca_module

6.1.1.1 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.

6.1.1.2 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
```

6.1.2 sca_lsf::sca_signal_if

6.1.2.1 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.

6.1.2.2 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
```

6.1.3 sca_lsf::sca_signal

6.1.3.1 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.

6.1.3.2 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
```

6.1.3.3 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**.

6.1.3.4 kind

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

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

6.1.4 sca_lsf::sca_in

6.1.4.1 Description

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

6.1.4.2 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
```

6.1.4.3 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**.

6.1.4.4 kind

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

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

6.1.5 sca_lsf::sca_out

6.1.5.1 Description

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

6.1.5.2 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
```

6.1.5.3 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**.

6.1.5.4 kind

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

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

6.1.6 sca_lsf::sca_add

6.1.6.1 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 Equation (6.1) to the equation system:

$$y(t) = k_1 \cdot x_1(t) + k_2 \cdot x_2(t) \tag{6.1}$$

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.

6.1.6.2 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
```

6.1.6.3 kind

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

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

6.1.7 sca_lsf::sca_sub

6.1.7.1 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 Equation (6.2) to the equation system:

$$y(t) = k_1 \cdot x_1(t) - k_2 \cdot x_2(t) \tag{6.2}$$

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.

6.1.7.2 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
```

6.1.7.3 kind

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

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

6.1.8 sca_lsf::sca_gain

6.1.8.1 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 Equation (6.3) to the equation system:

$$y(t) = k \cdot x(t) \tag{6.3}$$

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

6.1.8.2 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
```

6.1.8.3 kind

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

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

6.1.9 sca_lsf::sca_dot

6.1.9.1 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 Equation (6.4) to the equation system:

$$y(t) = k \cdot \frac{dx(t)}{dt} \tag{6.4}$$

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

6.1.9.2 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
```

6.1.9.3 kind

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

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

6.1.10 sca_lsf::sca_integ

6.1.10.1 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 Equation (6.5) to the equation system:

$$y(t) = k \cdot \int_{t_{\text{start}}}^{t} x(t)dt + y_0 \tag{6.5}$$

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 shall be done from the first calculation time point t_{start} to the current time t.

If y_0 is set to **sca_util::SCA_UNDEFINED**, the primitive shall contribute the equation $y = k \cdot x$ for the first calculation instead of Equation (6.5). In this case, y_0 is set to the resulting y value of the first calculation.

6.1.10.2 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
```

6.1.10.3 kind

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

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

6.1.11 sca_lsf::sca_delay

6.1.11.1 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 Equation (6.6) to the equation system:

$$y(t) = \begin{cases} y_0 & t \le delay \\ k \cdot x(t - delay) & t > delay \end{cases} \tag{6.6}$$

where t is the time in seconds, *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.

6.1.11.2 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,
                        const 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
```

6.1.11.3 Constraint of usage

The delay shall be greater than or equal to zero.

6.1.11.4 kind

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

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

6.1.12 sca_lsf::sca_source

6.1.12.1 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 Equation (6.7) to the equation system:

$$y(t) = \begin{cases} init\_value & t < delay \\ offset + amplitude \cdot \sin(2\pi \cdot frequency \cdot (t - delay) + phase) & t \geq delay \end{cases} \tag{6.7}$$

where t is the time in seconds, *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, and $y(t)$ is the LSF output signal.

In small-signal frequency-domain simulation, the primitive shall contribute Equation (6.8) to the equation system:

$$y(f) = ac\_amplitude \cdot \{\cos(ac\_phase) + j \cdot \sin(ac\_phase)\} \tag{6.8}$$

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 Equation (6.9) to the equation system:

$$y(f) = ac\_noise\_amplitude \tag{6.9}$$

where f is the simulation frequency, and *ac_noise_amplitude* is the small-signal noise amplitude.

6.1.12.2 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,
                         const 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
```

6.1.12.3 kind

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

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

6.1.13 sca_lsf::sca_ltf_nd

6.1.13.1 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 5.4.3). The primitive shall contribute Equation (6.10) to the equation system:

$$\begin{aligned} & den_{N-1}\frac{d^{N-1}y(t)}{dt} + den_{N-2}\frac{d^{N-2}y(t)}{dt} + \cdots + den_1\frac{dy(t)}{dt} + den_0 \cdot y(t) \\ & = k \cdot \left(num_{M-1}\frac{d^{M-1}x(t-delay)}{dt} + num_{M-2}\frac{d^{M-2}x(t-delay)}{dt} \right. \\ & \left. + \cdots + num_1\frac{dx(t-delay)}{dt} + num_0 \cdot x(t-delay) \right) \end{aligned} \tag{6.10}$$

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 continuous-time delay in seconds, applied to the values available at the input, and $y(t)$ is the LSF output signal.

6.1.13.2 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", sc_core::SC_ZERO_TIME ), 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_,
                const 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
```

6.1.13.3 Constraint on usage

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

6.1.13.4 kind

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

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

6.1.14 sca_lsf::sca_ltf_zp

6.1.14.1 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 5.4.4). The primitive shall contribute Equation (6.11) to the equation system:

$$\begin{aligned}&\left(\frac{d}{dt}-poles_{N-1}\right)\left(\frac{d}{dt}-poles_{N-2}\right)\cdots\left(\frac{d}{dt}-poles_{1}\right)\left(\frac{d}{dt}-poles_{0}\right)y(t)\\&=k\cdot\left\{\left(\frac{d}{dt}-zeros_{M-1}\right)\left(\frac{d}{dt}-zeros_{M-2}\right)\cdots\left(\frac{d}{dt}-zeros_{1}\right)\left(\frac{d}{dt}-zeros_{0}\right)x(t-delay)\right\}\end{aligned} \tag{6.11}$$

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 continuous-time delay in seconds applied to the values available at the input, and $y(t)$ is the LSF output signal.

6.1.14.2 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", sc_core::SC_ZERO_TIME ), 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_,
                const 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
```

6.1.14.3 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.

6.1.14.4 kind

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

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

6.1.15 sca_lsf::sca_ss

6.1.15.1 Description

The class **sca_lsf::sca_ss** shall implement a primitive module for the LSF MoC that realizes a system, which behavior is defined by single-input single-output state-space equations (see 5.4.5). The primitive shall contribute Equation (6.12) and Equation (6.13) to the equation system:

$$\frac{ds(t)}{dt} = \mathbf{A}\cdot s(t) + \mathbf{B}\cdot x(t - delay) \tag{6.12}$$

$$y(t) = \mathbf{C}\cdot s(t) + \mathbf{D}\cdot x(t - delay) \tag{6.13}$$

where $s(t)$ is the state vector, $x(t)$ is the LSF input signal, *delay* is the continuous-time 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.

6.1.15.2 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>(),
            const 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
```

6.1.15.3 Constraint on usage

It shall be an error if one of the following conditions is not met:

— Argument *a* shall be a square matrix of the size of state vector *s*.

— Argument *b* shall be a matrix with one column and the size of state vector *s* rows.

— Argument *c* shall be a matrix with one row and the size of state vector *s* columns.

— 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**.

6.1.15.4 kind

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

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

6.1.16 sca_lsf::sca_tdf::sca_gain, sca_lsf::sca_tdf_gain

6.1.16.1 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 Equation (6.14) to the equation system:

$$y(t) = scale \cdot inp \cdot x(t) \tag{6.14}$$

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.

The class **sca_lsf::sca_tdf_gain** shall be defined as an alias for class **sca_lsf::sca_tdf::sca_gain**.

6.1.16.2 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
```

6.1.16.3 kind

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

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

6.1.17 sca_lsf::sca_tdf::sca_source, sca_lsf::sca_tdf_source

6.1.17.1 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 Equation (6.15) to the equation system:

$$y(t) = scale \cdot inp \tag{6.15}$$

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.

The class **sca_lsf::sca_tdf_source** shall be defined as an alias for class **sca_lsf::sca_tdf::sca_source**.

6.1.17.2 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
```

6.1.17.3 kind

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

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

6.1.18 sca_lsf::sca_tdf::sca_sink, sca_lsf::sca_tdf_sink

6.1.18.1 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*.

The class **sca_lsf::sca_tdf_sink** shall be defined as an alias for class **sca_lsf::sca_tdf::sca_sink**.

6.1.18.2 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
```

6.1.18.3 kind

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

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

6.1.19 sca_lsf::sca_tdf::sca_mux, sca_lsf::sca_tdf_mux

6.1.19.1 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 Equation (6.16) to the equation system:

$$y(t) = \begin{cases} x_1(t) & ctrl = false \\ x_2(t) & ctrl = true \end{cases} \tag{6.16}$$

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.

The class **sca_lsf::sca_tdf_mux** shall be defined as an alias for class **sca_lsf::sca_tdf::sca_mux**.

6.1.19.2 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
```

6.1.19.3 kind

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

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

6.1.20 sca_lsf::sca_tdf::sca_demux, sca_lsf::sca_tdf_demux

6.1.20.1 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 Equation (6.17) and Equation (6.18) to the equation system:

$$y_1(t) = \begin{cases} x(t) & ctrl = false \\ 0 & ctrl = true \end{cases} \tag{6.17}$$

$$y_2(t) = \begin{cases} 0 & ctrl = false \\ x(t) & ctrl = true \end{cases} \tag{6.18}$$

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.

The class **sca_lsf::sca_tdf_demux** shall be defined as an alias for class **sca_lsf::sca_tdf::sca_demux**.

6.1.20.2 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
```

6.1.20.3 kind

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

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

6.1.21 sca_lsf::sca_de::sca_gain, sca_lsf::sca_de_gain

6.1.21.1 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 Equation (6.19) to the equation system:

$$y(t) = scale \cdot inp \cdot x(t) \tag{6.19}$$

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.

The class **sca_lsf::sca_de_gain** shall be defined as an alias for class **sca_lsf::sca_de::sca_gain**.

6.1.21.2 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
```

6.1.21.3 kind

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

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

6.1.22 sca_lsf::sca_de::sca_source, sca_lsf::sca_de_source

6.1.22.1 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 Equation (6.20) to the equation system:

$$y(t) = scale \cdot inp \tag{6.20}$$

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.

The class **sca_lsf::sca_de_source** shall be defined as an alias for class **sca_lsf::sca_de::sca_source**.

6.1.22.2 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
```

6.1.22.3 kind

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

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

6.1.23 sca_lsf::sca_de::sca_sink, sca_lsf::sca_de_sink

6.1.23.1 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*.

The class **sca_lsf::sca_de_sink** shall be defined as an alias for class **sca_lsf::sca_de::sca_sink**.

6.1.23.2 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
```

6.1.23.3 kind

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

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

6.1.24 sca_lsf::sca_de::sca_mux, sca_lsf::sca_de_mux

6.1.24.1 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 Equation (6.21) to the equation system:

$$y(t) = \begin{cases} x_1(t) & ctrl = false \\ x_2(t) & ctrl = true \end{cases} \tag{6.21}$$

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.

The class **sca_lsf::sca_de_mux** shall be defined as an alias for class **sca_lsf::sca_de::sca_mux**.

6.1.24.2 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
```

6.1.24.3 kind

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

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

6.1.25 sca_lsf::sca_de::sca_demux, sca_lsf::sca_de_demux

6.1.25.1 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 Equation (6.22) and Equation (6.23) to the equation system:

$$y_1(t) = \begin{cases} x(t) & ctrl = false \\ 0 & ctrl = true \end{cases} \tag{6.22}$$

$$y_2(t) = \begin{cases} 0 & ctrl = false \\ x(t) & ctrl = true \end{cases} \tag{6.23}$$

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.

The class **sca_lsf::sca_de_demux** shall be defined as an alias for class **sca_lsf::sca_de::sca_demux**.

6.1.25.2 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
```

6.1.25.3 kind

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

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

6.2 Hierarchical 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 1666-2011 as well as the following specific rules:

a) 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.

b) 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.

c) 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.

d) 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.

6.3 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.

6.3.1 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 4.2.7).

6.3.1.1 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 functions **set_timestep** or **set_max_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<T>**, if any.

It shall be an error if a timestep value is not assigned to at least one LSF module. The assigned and propagated timestep values shall be consistent throughout the LSF cluster; otherwise, it shall be an error. It shall be an error if the propagated timestep is equal to the time returned by function **sca_core::sca_max_time**.

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 marked to accept attribute changes, 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<T>** of the predefined LSF primitive modules. In this case, the LSF cluster is included in the timestep calculation of the TDF cluster and needs to comply with the same rules (see 5.3.1.2).

6.3.1.2 Equation system setup and solvability check

For each LSF cluster, an equation system shall be set up by combining:

— the contributing equations of each of the predefined LSF primitive modules in the cluster.

— 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.

6.3.2 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.

6.3.2.1 Initialization

For each LSF cluster:

— all LSF signals and states shall be set to zero.

— for all LSF signals consistent initial conditions shall be calculated in agreement with the initial conditions set by the predefined primitives.

6.3.2.2 Time-domain simulation

The solver shall at least provide results at the calculated timestep distances. If the current calculation timestep is **sc_core::SC_ZERO_TIME**, the time and the state of the equation system shall be restored to the time and state before the last calculation and the calculation shall be repeated on the new input values.

6.3.2.3 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.

6.3.2.4 Synchronization with the 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.

7. 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 clause.

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 at least the primitives having two terminals as defined in this clause. 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.

All ELN primitive modules, which support current tracing, shall be derived from class **sca_util::***sca_traceable_object*[†].

7.1 Class definitions

All names used in the ELN class definitions shall be placed in the namespace **sca_eln**.

7.1.1 sca_eln::sca_module

7.1.1.1 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.

7.1.1.2 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
```

7.1.2 sca_eln::sca_node_if

7.1.2.1 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.

7.1.2.2 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
```

7.1.3 sca_eln::sca_terminal

7.1.3.1 Description

The class **sca_eln::sca_terminal** shall define a port class for the ELN MoC.

7.1.3.2 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
```

7.1.3.3 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_eln_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**.

7.1.3.4 kind

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

The member function **kind** shall return the string “**sca_eln::sca_terminal**”.

7.1.4 sca_eln::sca_node

7.1.4.1 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.

7.1.4.2 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
```

7.1.4.3 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_eln_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**.

7.1.4.4 kind

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

The member function **kind** shall return the string "**sca_eln::sca_node**".

7.1.5 sca_eln::sca_node_ref

7.1.5.1 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 holds a voltage of zero volt. An application shall not access the associated interface directly.

7.1.5.2 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
```

7.1.5.3 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_eln_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**.

7.1.5.4 kind

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

The member function **kind** shall return the string "**sca_eln::sca_node_ref**".

7.1.6 sca_eln::sca_r

7.1.6.1 Description

The class **sca_eln::sca_r** shall implement a primitive module for the ELN MoC that represents a resistor. The primitive shall contribute Equation (7.1) to the equation system:

$$v_{p,n}(t) = i_{p,n}(t) \cdot value \tag{7.1}$$

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.

7.1.6.2 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
```

7.1.6.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_r**".

7.1.7 sca_eln::sca_c

7.1.7.1 Description

The class **sca_eln::sca_c** shall implement a primitive module for the ELN MoC that represents a capacitor. The primitive shall contribute Equation (7.2) to the equation system:

$$i_{p,n}(t) = \frac{d\left(value \cdot v_{p,n}(t) + q_0\right)}{dt} \tag{7.2}$$

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.

If the initial charge q_0 is set to **sca_util::SCA_UNDEFINED**, the primitive shall contribute no equation to the equation system for the first calculation. In this case, the initial charge q_0 shall be calculated as follows: $q_0 = value \cdot v_{p,n0}$, where $v_{p,n0}$ is the voltage across the capacitor after the first calculation.

7.1.7.2 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
```

7.1.7.3 Constraint of usage

The argument *value* shall not be numerically zero.

7.1.7.4 kind

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

The member function **kind** shall return the string "**sca_eln::sca_c**".

7.1.8 sca_eln::sca_l

7.1.8.1 Description

The class **sca_eln::sca_l** shall implement a primitive module for the ELN MoC that represents an inductor. The primitive shall contribute Equation (7.3) to the equation system:

$$v_{p,n}(t) = \frac{d\left(value \cdot i_{p,n}(t) + psi_0\right)}{dt} \tag{7.3}$$

where *value* is the inductance in henry, psi_0 is the initial linked 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*.

If the initial linked flux psi_0 is set to **sca_util::SCA_UNDEFINED**, the primitive shall contribute to the equation system the equation $v_{p,n} = 0$ for the first calculation instead of Equation (7.3). In this case, the initial linked flux psi_0 shall be calculated as follows: $psi_0 = value \cdot i_{p,n0}$, where $i_{p,n0}$ is the current flowing through the inductor after the first calculation.

7.1.8.2 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> psi0;

    virtual const char* kind() const;

    explicit sca_l( sc_core::sc_module_name, double value_ = 1.0, double psi0_ = 0.0 )
      : p( "p" ), n( "n" ), value( "value", value_ ), psi0( "psi0", psi0_ )
    { implementation-defined }
  };

} // namespace sca_eln
```

7.1.8.3 Constraint of usage

The argument *value* shall not be numerically zero.

7.1.8.4 kind

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

The member function **kind** shall return the string "**sca_eln::sca_l**".

7.1.9 sca_eln::sca_vcvs

7.1.9.1 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 Equation (7.4) to the equation system:

$$v_{np,nn}(t) = value \cdot v_{ncp,ncn}(t) \tag{7.4}$$

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*.

7.1.9.2 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
```

7.1.9.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_vcvs**".

7.1.10 sca_eln::sca_vccs

7.1.10.1 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 Equation (7.5) to the equation system:

$$i_{np,nn}(t) = value \cdot v_{ncp,ncn}(t) \tag{7.5}$$

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*.

7.1.10.2 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
```

7.1.10.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_vccs**".

7.1.11 sca_eln::sca_ccvs

7.1.11.1 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 Equation (7.6) and Equation (7.7) to the equation system:

$$v_{np,nn}(t) = value \cdot i_{ncp,ncn}(t) \tag{7.6}$$

$$v_{ncp,ncn}(t) = 0 \tag{7.7}$$

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*.

7.1.11.2 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
```

7.1.11.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_ccvs**".

7.1.12 sca_eln::sca_cccs

7.1.12.1 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 Equation (7.8) and Equation (7.9) to the equation system:

$$i_{np,nn}(t) = value \cdot i_{ncp,ncn}(t) \tag{7.8}$$

$$v_{ncp,ncn}(t) = 0 \tag{7.9}$$

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*.

7.1.12.2 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
```

7.1.12.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_cccs**".

7.1.13 sca_eln::sca_nullor

7.1.13.1 Description

The class **sca_eln::sca_nullor** shall implement a primitive module for the ELN MoC that represents a nullor. The primitive shall contribute Equation (7.10) and Equation (7.11) to the equation system:

$$v_{nip,nin}(t) = 0 \tag{7.10}$$

$$i_{nip,nin}(t) = 0 \tag{7.11}$$

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).

7.1.13.2 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
```

7.1.13.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_nullor**".

7.1.14 sca_eln::sca_gyrator

7.1.14.1 Description

The class **sca_eln::sca_gyrator** shall implement a primitive module for the ELN MoC that represents a gyrator. The primitive shall contribute Equation (7.12) and Equation (7.13) to the equation system:

$$i_{p_1,n_1}(t) = g_2 \cdot v_{p_2,n_2}(t) \tag{7.12}$$

$$i_{p_2,n_2}(t) = -g_1 \cdot v_{p_1,n_1}(t) \tag{7.13}$$

where g_1 and g_2 are the gyration conductances in siemens, $v_{p_2,n_2}(t)$ 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.

7.1.14.2 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
```

7.1.14.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_gyrator**".

7.1.15 sca_eln::sca_ideal_transformer

7.1.15.1 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 Equation (7.14) and Equation (7.15) to the equation system:

$$v_{p_1,n_1}(t) = ratio \cdot v_{p_2,n_2}(t) \tag{7.14}$$

$$i_{p_2,n_2}(t) = ratio \cdot i_{p_1,n_1}(t) \tag{7.15}$$

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.

7.1.15.2 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
```

7.1.15.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_ideal_transformer**".

7.1.16 sca_eln::sca_transmission_line

7.1.16.1 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 Equation (7.16) and Equation (7.17) to the equation system:

$$v_{a_1,b_1}(t) = \begin{cases} z_0 \cdot i_{a_1,b_1}(t) & t < delay \\ e^{-delta_0 delay}\left(v_{a_2,b_2}(t-delay) + z_0 \cdot i_{a_2,b_2}(t-delay)\right) + z_0 \cdot i_{a_1,b_1}(t) & t \geq delay \end{cases} \quad (7.16)$$

$$v_{a_2,b_2}(t) = \begin{cases} z_0 \cdot i_{a_2,b_2}(t) & t < delay \\ e^{-delta_0 delay}\left(v_{a_1,b_1}(t-delay) + z_0 \cdot i_{a_1,b_1}(t-delay)\right) + z_0 \cdot i_{a_2,b_2}(t) & t \geq delay \end{cases} \quad (7.17)$$

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.

7.1.16.2 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_core::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,
                                    const sca_core::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
```

7.1.16.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_transmission_line**".

7.1.17 sca_eln::sca_vsource

7.1.17.1 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 Equation (7.18) to the equation system:

$$v_{p,n}(t) = \begin{cases} init\_value & t < delay \\ offset + amplitude \cdot \sin(2\pi \cdot frequency \cdot (t - delay) + phase) & t \geq delay \end{cases} \quad (7.18)$$

where t is the time in seconds, *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, and $v_{p,n}(t)$ is the output voltage across terminals p and n.

In small-signal frequency-domain simulation, the primitive shall contribute Equation (7.19) to the equation system:

$$v_{p,n}(f) = ac\_amplitude \cdot \{\cos(ac\_phase) + j \cdot \sin(ac\_phase)\} \quad (7.19)$$

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 Equation (7.20) to the equation system:

$$v_{p,n}(f) = ac\_noise\_amplitude \quad (7.20)$$

where f is the simulation frequency in hertz, and *ac_noise_amplitude* is the small-signal noise amplitude in volt.

7.1.17.2 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,
                          const 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
```

7.1.17.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_vsource**".

7.1.18 sca_eln::sca_isource

7.1.18.1 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 Equation (7.21) to the equation system:

$$i_{p,n}(t) = \begin{cases} init\_value & t < delay \\ offset + amplitude \cdot \sin(2\pi \cdot frequency \cdot (t - delay) + phase) & t \geq delay \end{cases} \quad (7.21)$$

where t is the time in seconds, *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, and $i_{p,n}(t)$ is the output current through the primitive from terminal p to terminal n.

In small-signal frequency-domain simulation, the primitive shall contribute Equation (7.22) to the equation system:

$$i_{p,n}(f) = ac\_amplitude \cdot \{\cos(ac\_phase) + j \cdot \sin(ac\_phase)\} \quad (7.22)$$

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 Equation (7.23) to the equation system:

$$i_{p,n}(f) = ac\_noise\_amplitude \quad (7.23)$$

where f is the simulation frequency, and *ac_noise_amplitude* is the small-signal noise amplitude in ampere.

7.1.18.2 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,
                          const 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
```

7.1.18.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_isource**".

7.1.19 sca_eln::sca_tdf::sca_r, sca_eln::sca_tdf_r

7.1.19.1 Description

The class **sca_eln::sca_tdf::sca_r** shall implement a primitive module for the ELN MoC that represents a resistor, which resistance is controlled by a TDF input signal. The primitive shall contribute Equation (7.24) to the equation system:

$$v_{p,n}(t) = scale \cdot inp \cdot i_{p,n}(t) \tag{7.24}$$

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.

The class **sca_eln::sca_tdf_r** shall be defined as an alias for class **sca_eln::sca_tdf::sca_r**.

7.1.19.2 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
```

7.1.19.3 kind

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

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

7.1.20 sca_eln::sca_tdf::sca_c, sca_eln::sca_tdf_c

7.1.20.1 Description

The class **sca_eln::sca_tdf::sca_c** shall implement a primitive module for the ELN MoC that represents a capacitor, which capacitance is controlled by a TDF input signal. The primitive shall contribute Equation (7.25) to the equation system:

$$i_{p,n}(t) = \frac{d\left(scale \cdot inp \cdot v_{p,n}(t) + q_0\right)}{dt} \tag{7.25}$$

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.

If the initial charge q_0 is set to **sca_util::SCA_UNDEFINED**, the primitive shall contribute no equation to the equation system for the first calculation. In this case, the initial charge q_0 shall be calculated as follows: $q_0 = scale \cdot inp \cdot v_{p,n0}$, where $v_{p,n0}$ is the voltage across the capacitor after the first calculation.

The class **sca_eln::sca_tdf_c** shall be defined as an alias for class **sca_eln::sca_tdf::sca_c**.

7.1.20.2 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
```

7.1.20.3 Constraint of usage

The TDF input signal *inp* shall not be zero.

7.1.20.4 kind

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

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

7.1.21 sca_eln::sca_tdf::sca_l, sca_eln::sca_tdf_l

7.1.21.1 Description

The class **sca_eln::sca_tdf::sca_l** shall implement a primitive module for the ELN MoC that represents an inductor, which inductance is controlled by a TDF input signal. The primitive shall contribute Equation (7.26) to the equation system:

$$v_{p,n}(t) = \frac{d\left(scale \cdot inp \cdot i_{p,n}(t) + psi_0\right)}{dt} \tag{7.26}$$

where *scale* is the constant scale coefficient, *inp* is the TDF input signal, psi_0 is the initial linked 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.

If the initial linked flux psi_0 is set to **sca_util::SCA_UNDEFINED**, the primitive shall contribute to the equation system the equation $v_{p,n} = 0$ for the first calculation instead of Equation (7.26). In this case, the initial linked flux psi_0 shall be calculated as follows: $psi_0 = scale \cdot inp \cdot i_{p,n0}$, where $i_{p,n0}$ is the current flowing through the inductor after the first calculation.

The class **sca_eln::sca_tdf_l** shall be defined as an alias for class **sca_eln::sca_tdf::sca_l**.

7.1.21.2 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> psi0;

      virtual const char* kind() const;

      explicit sca_l( sc_core::sc_module_name, double scale_ = 1.0, double psi0_ = 0.0)
        : p( "p" ), n( "n" ), inp( "inp" ), scale( "scale", scale_ ), psi0( "psi0", psi0_ )
      { implementation-defined }
    };

  } // namespace sca_tdf

  typedef sca_eln::sca_tdf::sca_l sca_tdf_l;

} //namespace sca_eln
```

7.1.21.3 Constraint of usage

The TDF input signal *inp* shall not be zero.

7.1.21.4 kind

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

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

7.1.22 sca_eln::sca_tdf::sca_rswitch, sca_eln::sca_tdf_rswitch

7.1.22.1 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 Equation (7.27) to the equation system:

$$v_{p,n}(t) = \begin{cases} r_{on} \cdot i_{p,n}(t) & ctrl \neq off\_state \\ r_{off} \cdot i_{p,n}(t) & ctrl = off\_state \end{cases} \tag{7.27}$$

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*.

The class **sca_eln::sca_tdf_rswitch** shall be defined as an alias for class **sca_eln::sca_tdf::sca_rswitch**.

7.1.22.2 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
```

7.1.22.3 kind

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

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

7.1.23 sca_eln::sca_tdf::sca_vsource, sca_eln::sca_tdf_vsource

7.1.23.1 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 Equation (7.28) to the equation system:

$$v_{p,n}(t) = scale \cdot inp \tag{7.28}$$

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.

The class **sca_eln::sca_tdf_vsource** shall be defined as an alias for class **sca_eln::sca_tdf::sca_vsource**.

7.1.23.2 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
```

7.1.23.3 kind

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

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

7.1.24 sca_eln::sca_tdf::sca_isource, sca_eln::sca_tdf_isource

7.1.24.1 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 Equation (7.29) to the equation system:

$$i_{p,n}(t) = scale \cdot inp \tag{7.29}$$

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.

The class **sca_eln::sca_tdf_isource** shall be defined as an alias for class **sca_eln::sca_tdf::sca_isource**.

7.1.24.2 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
```

7.1.24.3 kind

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

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

7.1.25 sca_eln::sca_tdf::sca_vsink, sca_eln::sca_tdf_vsink

7.1.25.1 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*.

The class **sca_eln::sca_tdf_vsink** shall be defined as an alias for class **sca_eln::sca_tdf::sca_vsink**.

7.1.25.2 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
```

7.1.25.3 kind

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

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

7.1.26 sca_eln::sca_tdf::sca_isink, sca_eln::sca_tdf_isink

7.1.26.1 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 Equation (7.30) to the equation system:

$$v_{p,n}(t) = 0 \tag{7.30}$$

where $v_{p,n}(t)$ is the voltage across terminals *p* and *n*.

The class **sca_eln::sca_tdf_isink** shall be defined as an alias for class **sca_eln::sca_tdf::sca_isink**.

7.1.26.2 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
```

7.1.26.3 kind

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

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

7.1.27 sca_eln::sca_de::sca_r, sca_eln::sca_de_r

7.1.27.1 Description

The class **sca_eln::sca_de::sca_r** shall implement a primitive module for the ELN MoC that represents a resistor, which resistance is controlled by a discrete-event input signal. The primitive shall contribute Equation (7.31) to the equation system:

$$v_{p,n}(t) = scale \cdot inp \cdot i_{p,n}(t) \tag{7.31}$$

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.

The class **sca_eln::sca_de_r** shall be defined as an alias for class **sca_eln::sca_de::sca_r**.

7.1.27.2 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
```

7.1.27.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_de::sca_r**".

7.1.28 sca_eln::sca_de::sca_c, sca_eln::sca_de_c

7.1.28.1 Description

The class **sca_eln::sca_de::sca_c** shall implement a primitive module for the ELN MoC that represents a capacitor, which capacitance is controlled by a discrete-event input signal. The primitive shall contribute Equation (7.32) to the equation system:

$$i_{p,n}(t) = \frac{d\left(scale \cdot inp \cdot v_{p,n}(t) + q_0\right)}{dt} \tag{7.32}$$

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.

If the initial charge q_0 is set to **sca_util::SCA_UNDEFINED**, the primitive shall contribute no equation to the equation system for the first calculation. In this case, the initial charge q_0 shall be calculated as follows: $q_0 = scale \cdot inp \cdot v_{p,n0}$, where $v_{p,n0}$ is the voltage across the capacitor after the first calculation.

The class **sca_eln::sca_de_c** shall be defined as an alias for class **sca_eln::sca_de::sca_c**.

7.1.28.2 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
```

7.1.28.3 Constraint of usage

The discrete-event input signal *inp* shall not be zero.

7.1.28.4 kind

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

The member function **kind** shall return the string "**sca_eln::sca_de::sca_c**".

7.1.29 sca_eln::sca_de::sca_l, sca_eln::sca_de_l

7.1.29.1 Description

The class **sca_eln::sca_de::sca_l** shall implement a primitive module for the ELN MoC that represents an inductor, which inductance is controlled by a discrete-event input signal. The primitive shall contribute Equation (7.33) to the equation system:

$$v_{p,n}(t) = \frac{d\left(scale \cdot inp \cdot i_{p,n}(t) + psi_0\right)}{dt} \tag{7.33}$$

where *scale* is the constant scale coefficient, *inp* is the discrete-event input signal, psi_0 is the initial linked 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.

If the initial linked flux psi_0 is set to **sca_util::SCA_UNDEFINED**, the primitive shall contribute to the equation system the equation $v_{p,n} = 0$ for the first calculation instead of Equation (7.33). In this case, the initial linked flux psi_0 shall be calculated as follows: $psi_0 = scale \cdot inp \cdot i_{p,n0}$, where $i_{p,n0}$ is the current flowing through the inductor after the first calculation.

The class **sca_eln::sca_de_l** shall be defined as an alias for class **sca_eln::sca_de::sca_l**.

7.1.29.2 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> psi0;

      virtual const char* kind() const;

      explicit sca_l( sc_core::sc_module_name, double scale_ = 1.0, double psi0_ = 0.0 )
        : p( "p" ), n( "n" ), inp( "inp" ), scale( "scale", scale_ ), psi0( "psi0", psi0_ )
      { implementation-defined }
    };

  } // namespace sca_de

  typedef sca_eln::sca_de::sca_l sca_de_l;

} // namespace sca_eln
```

7.1.29.3 Constraint of usage

The discrete-event input signal *inp* shall not be zero.

7.1.29.4 kind

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

The member function **kind** shall return the string "**sca_eln::sca_de::sca_l**".

7.1.30 sca_eln::sca_de::sca_rswitch, sca_eln::sca_de_rswitch

7.1.30.1 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 Equation (7.34) to the equation system:

$$v_{p,n}(t) = \begin{cases} r_{on} \cdot i_{p,n}(t) & ctrl \neq off\_state \\ r_{off} \cdot i_{p,n}(t) & ctrl = off\_state \end{cases} \tag{7.34}$$

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*.

The class **sca_eln::sca_de_rswitch** shall be defined as an alias for class **sca_eln::sca_de::sca_rswitch**.

7.1.30.2 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
```

7.1.30.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_de::sca_rswitch**".

7.1.31 sca_eln::sca_de::sca_vsource, sca_eln::sca_de_vsource

7.1.31.1 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 Equation (7.35) to the equation system:

$$v_{p,n}(t) = scale \cdot inp \tag{7.35}$$

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.

The class **sca_eln::sca_de_vsource** shall be defined as an alias for class **sca_eln::sca_de::sca_vsource**.

7.1.31.2 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
```

7.1.31.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_de::sca_vsource**".

7.1.32 sca_eln::sca_de::sca_isource, sca_eln::sca_de_isource

7.1.32.1 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 Equation (7.36) to the equation system:

$$i_{p,n}(t) = scale \cdot inp \tag{7.36}$$

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.

The class **sca_eln::sca_de_isource** shall be defined as an alias for class **sca_eln::sca_de::sca_isource**.

7.1.32.2 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
```

7.1.32.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_de::sca_isource**".

7.1.33 sca_eln::sca_de::sca_vsink, sca_eln::sca_de_vsink

7.1.33.1 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*.

The class **sca_eln::sca_de_vsink** shall be defined as an alias for class **sca_eln::sca_de::sca_vsink**.

7.1.33.2 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
```

7.1.33.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_de::sca_vsink**".

7.1.34 sca_eln::sca_de::sca_isink, sca_eln::sca_de_isink

7.1.34.1 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 Equation (7.37) to the equation system:

$$v_{p,n}(t) = 0 \tag{7.37}$$

where $v_{p,n}(t)$ is the voltage across terminals *p* and *n*.

The class **sca_eln::sca_de_isink** shall be defined as an alias for class **sca_eln::sca_de::sca_isink**.

7.1.34.2 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
```

7.1.34.3 kind

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

The member function **kind** shall return the string "**sca_eln::sca_de::sca_isink**".

7.2 Hierarchical 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 1666-2011 as well as the following specific rules:

a) 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.

b) 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.

c) 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.

d) 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.

7.3 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.

7.3.1 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 4.2.7).

NOTE—Connections by channels of class **sca_eln::sca_node_ref** are ignored for building ELN clusters.

7.3.1.1 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 functions **set_timestep** or **set_max_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<T>**, if any.

It shall be an error if a timestep value is not assigned to at least one ELN module. The assigned and propagated timestep values shall be consistent throughout the ELN cluster; otherwise, it shall be an error. It shall be an error if the propagated timestep is equal to the time returned by function **sca_core::sca_max_time**.

NOTE—An ELN cluster can be considered as one TDF module marked to accept attribute changes, 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<T>** of the predefined ELN primitive modules. The ELN cluster is included in the timestep calculation of the TDF cluster and need to comply with the same rules (see 5.3.1.2).

7.3.1.2 Equation system setup and solvability check

For each ELN cluster, an equation system shall be set up by combining:

- the contributing equations of each of the predefined ELN primitive modules in the cluster.
- the equations implied by 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.

7.3.2 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.

7.3.2.1 Initialization

The ELN initialization phase calculates consistent initial conditions for the equation systems.

7.3.2.2 Time-domain simulation

The solver shall provide results at least at the calculated timestep distances. If the current calculation timestep is **sc_core::SC_ZERO_TIME**, the time and the state of the equation system shall be restored to the time and state before the last calculation and the calculation shall be repeated on the new input values.

7.3.2.3 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.

7.3.2.4 Synchronization with the 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.

8. Predefined analyses

8.1 Time-domain analysis

The time-domain analysis shall be applicable to all descriptions supported by the predefined models of computation as defined in Clause 5, Clause 6 and Clause 7. 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 1666-2011.

8.1.1 Elaboration and simulation

The execution of a time-domain analysis consists of elaboration followed by simulation (see 5.3, 6.3, 7.3). The elaboration and simulation shall use the same semantics as defined in IEEE Std 1666-2011.

8.1.2 Running elaboration and simulation

An implementation shall provide either or both of the following two mechanisms for running elaboration and simulation:

— Under application control using functions **sc_main** and **sc_core::sc_start**.

— Under control of the kernel.

An application may pause and resume the simulation using the function **sc_core::sc_pause** followed by the function **sc_core::sc_start**. The time of each individual TDF, LSF, and ELN module after pausing is implementation-defined. The time domain simulation is resumed from these individual module times.

An application may stop the simulation using the function **sc_core::sc_stop**. If the stop mode, defined by function **sc_set_stop_mode**, is set to **sc_core::SC_STOP_IMMEDIATE**, the time domain simulation shall stop immediately. If the stop mode is set to **sc_core::SC_STOP_FINISH_DELTA**, the time of each individual TDF, LSF, and ELN module after stopping is implementation-defined.

NOTE—TDF, LSF, and ELN modules can be instantiated in the **sc_main** context.

8.2 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 5.3, 6.3, and 7.3. 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:

a) Small-signal frequency-domain analysis shall solve for each frequency point the linear complex equation system including all small-signal frequency-domain source contributions.

b) 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**.

8.2.1 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 8.2.2). While performing the analysis, the state of the time-domain simulation shall not be changed.

8.2.1.1 Elaboration

The small-signal frequency-domain elaboration shall be performed if one of the dedicated start functions is executed (see 8.2.2). 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 set up one complex linear frequency-dependent equation system by composing the equation system contributions of TDF, LSF, and ELN descriptions.

8.2.1.2 Simulation

The linear complex equation system for the chosen analysis kind, shall be solved for each frequency point according to the kind of analysis.

8.2.2 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 8.1, if not yet performed.

8.2.2.1 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 logarithmically distributed and if *scale* is **sca_ac_analysis::SCA_LIN**, the remaining points shall be linearly distributed.

The second function shall calculate the small-signal frequency-domain behavior at the frequency points given by the vector *frequencies*.

8.2.2.2 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*.

8.2.3 Small-signal frequency-domain analysis of TDF descriptions

The small-signal frequency-domain and noise representation of a TDF description shall contribute the complex equation system shown in Equation (8.1):

$$A(f)\cdot x(f)+b(f)+b_{noise}(f)+c(f,x(f))=0 \tag{8.1}$$

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 5.1.1.10) or the corresponding registered member function (see 5.1.1.12). The contributions shall describe linear complex functions between ports of class **sca_tdf::sca_in** and ports of class **sca_tdf::sca_out_base**.

$x(f)$ is a complex vector representing the small-signal frequency-domain values of the ports of class **sca_tdf::sca_out_base**.

$b(f)$ and $b_{noise}(f)$ are complex frequency dependent vectors, which represent the contributions to the ports of class **sca_tdf::sca_out_base** 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_base**. 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_base**. In this case the contribution $b(f)$ shall be set to zero.

$c(f, x(f))$ 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_base**.

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 function **sca_ac_analysis::sca_ac** (see 8.2.3.1), 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_base**, 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.

8.2.3.1 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_base<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_base** 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.

8.2.3.2 sca_ac_analysis::sca_ac_noise

```
namespace sca_ac_analysis {

  template<class T>
  sca_util::sca_complex& sca_ac_noise( const sca_tdf::sca_out_base<T>& );

} // namespace sca_ac_analysis
```

The function **sca_ac_analysis::sca_ac_noise** applied to port of class **sca_tdf::sca_out_base** 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.

8.2.3.3 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; otherwise, it shall return false.

8.2.3.4 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; otherwise, it shall return false.

8.2.3.5 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.

8.2.3.6 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.

8.2.3.7 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.

8.2.3.8 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.

8.2.3.9 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.

8.2.3.10 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,
                                       const 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 8.2.3.10) 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.

8.2.3.11 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,
                                       const 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 8.2.3.11) 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.

8.2.3.12 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,
                                        const 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,
                                        const 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 8.2.3.12) .

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.

8.2.4 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 6.3.1.2) 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.

8.2.5 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 7.3.1.2) 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.

9. Utility definitions

9.1 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**.

9.1.1 Class definitions

All names used in the class definitions and function declarations for tracing shall be placed in the namespace **sca_util**.

9.1.1.1 sca_util::sca_trace_mode_base

9.1.1.1.1 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 9.1.1.2.5).

An implementation shall at least support the trace mode manipulators as defined in this subclause.

9.1.1.1.2 Class definition

```
namespace sca_util {

  class sca_trace_mode_base
  {
   public:
    virtual ~sca_trace_mode_base() = 0;
  };

  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
```

9.1.1.1.3 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 Equation (9.1):

$$DB = 20 \cdot \log_{10}(magnitude) \tag{9.1}$$

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. It shall be an error, if *n* is equal to zero.