



IEC 62529

Edition 1.0 2007-11

INTERNATIONAL STANDARD

IEEE 1641™

Standard for Signal and Test Definition



Documents

Document Sharing Hub



THIS PUBLICATION IS COPYRIGHT PROTECTED

Copyright © 2007 IEEE

All rights reserved. IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by the Institute of Electrical and Electronics Engineers, Inc.

Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the IEC Central Office.

Any questions about IEEE copyright should be addressed to the IEEE. Enquiries about obtaining additional rights to this publication and other information requests should be addressed to the IEC or your local IEC member National Committee.

IEC Central Office
3, rue de Varembe
CH-1211 Geneva 20
Switzerland
Email: inmail@iec.ch
Web: www.iec.ch

The Institute of Electrical and Electronics Engineers, Inc
3 Park Avenue
US-New York, NY10016-5997
USA
Email: stds-info@ieee.org
Web: 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 corrigenda or an amendment might have been published.

- Catalogue of IEC publications: www.iec.ch/searchpub

The IEC on-line Catalogue enables you to search by a variety of criteria (reference number, text, technical committee,...). It also gives information on projects, withdrawn and replaced publications.

- IEC Just Published: www.iec.ch/online_news/justpub

Stay up to date on all new IEC publications. Just Published details twice a month all new publications released. Available on-line and also by email.

- Electropedia: www.electropedia.org

The world's leading online dictionary of electronic and electrical terms containing more than 20 000 terms and definitions in English and French, with equivalent terms in additional languages. Also known as the International Electrotechnical Vocabulary online.

- Customer Service Centre: www.iec.ch/webstore/custserv

If you wish to give us your feedback on this publication or need further assistance, please visit the Customer Service Centre FAQ or contact us:

Email: csc@iec.ch

Tel.: +41 22 919 02 11

Fax: +41 22 919 03 00



IEC 62529

Edition 1.0 2007-11

INTERNATIONAL STANDARD

IEEE 1641™

Standard for signal and test definition

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

PRICE CODE **XP**

ICS 25.040

ISBN 2-8318-9482-4

CONTENTS

FOREWORD.....	10
IEEE introduction.....	13
1. Overview.....	14
1.1 Scope.....	14
1.2 Purpose.....	15
2. Definitions, abbreviations, and acronyms.....	15
2.1 Definitions	15
2.2 Abbreviations and acronyms	17
3. Structure of this standard	18
3.1 Layers.....	18
4. Signal modeling language (SML) layer.....	20
5. Basic signal component (BSC) layer	20
5.1 BSC layer base classes.....	20
5.2 BSCs	21
5.3 SignalFunction template	22
6. Test signal framework (TSF) layer.....	22
6.1 TSF classes	22
6.2 TSF signals	23
7. Test procedure language (TPL) layer	25
7.1 Goals of the TPL.....	25
7.2 Elements of the TPL	25
7.3 Use of the TPL.....	25
Annex A (normative) Signal modeling language (SML)	26
A.1 Use of the SML.....	26
A.2 Introduction.....	26
A.3 Physical types	27
A.4 Signal definitions	30
A.5 Pure signals	31
A.5.1 Nonperiodic signals.....	31
A.5.2 Periodic signals	32
A.6 Pure signal-combining mechanisms	33
A.6.1 Piecewise continuous signals (PCSs).....	33
A.6.2 Sum.....	36
A.6.3 Product	36
A.7 Pure function transformations.....	36
A.7.1 Fourier transform.....	37
A.8 Measuring, limiting, and sampling signals	37
A.8.1 Confining parameters to a limit.....	38
A.8.2 Sampling signals	38
A.9 Digital signals	38

A.9.1	Defining Digital.....	39
A.9.2	Defining DigitalSignal	39
A.9.3	Conversion routines.....	40
A.9.4	Patterns	41
A.10	Basic component SML.....	42
A.10.1	Source ::SignalFunction	42
A.10.2	Conditioner ::SignalFunction	44
A.10.3	EventFunction ::SignalFunction.....	47
A.10.4	Sensor ::SignalFunction	50
A.10.5	Digital ::SignalFunction	50
A.10.6	Connection ::SignalFunction.....	51
Annex B (normative)	Basic signal component (BSC) layer.....	53
B.1	BSC layer base classes.....	53
B.2	BSC subclasses	53
B.3	Description of a BSC	58
B.3.1	Diagrammatic representation of a BSC.....	58
B.3.2	BSC interfaces.....	59
B.3.3	Types of BSCs.....	60
B.3.4	BSC attribute default values.....	61
B.3.5	BSC subclass descriptions.....	61
B.4	Physical class	63
B.4.1	Permissible physical types and their units.....	65
B.4.2	Unit prefixes	69
B.5	PulseDefns class	70
B.5.1	PulseDefn class	71
B.6	SignalFunction class	71
B.6.1	Source ::SignalFunction	72
B.6.2	Conditioner ::SignalFunction	80
B.6.3	EventFunction ::SignalFunction.....	94
B.6.4	Sensor ::SignalFunction	100
B.6.5	Digital ::SignalFunction	106
B.6.6	Connection ::SignalFunction.....	108
Annex C (normative)	Dynamic signal descriptions.....	112
C.1	Introduction.....	112
C.2	Basic classes	113
C.2.1	ResourceManager.....	113
C.2.2	Signal.....	114
C.2.3	BSCs.....	116
C.3	Dynamic signal goals and use cases	118
Annex D (normative)	IDL basic components.....	119
D.1	Introduction.....	119
D.2	IDL BSC library.....	119
Annex E (informative)	Test signal framework (TSF) for ATLAS	161
E.1	Introduction.....	161
E.2	AC_SIGNAL<type: Current Power Voltage>	161
E.2.1	Definition	161
E.2.2	Interface properties.....	162

E.2.3	Notes.....	162
E.2.4	Model description.....	162
E.2.5	Rules.....	162
E.2.6	Example.....	163
E.3	AM_SIGNAL	163
E.3.1	Definition	163
E.3.2	Interface properties.....	164
E.3.3	Notes.....	164
E.3.4	Model description.....	164
E.3.5	Rules.....	165
E.3.6	Example.....	165
E.4	DC_SIGNAL<type: Voltage Current Power>	165
E.4.1	Definition	165
E.4.2	Interface properties.....	166
E.4.3	Notes.....	166
E.4.4	Model description.....	166
E.4.5	Rules.....	167
E.4.6	Example.....	167
E.5	DIGITAL_PARALLEL	168
E.5.1	Definition	168
E.5.2	Interface properties.....	168
E.5.3	Notes.....	168
E.5.4	Model description.....	169
E.5.5	Rules.....	169
E.5.6	Example.....	169
E.6	DIGITAL_SERIAL	170
E.6.1	Definition	170
E.6.2	Interface properties.....	171
E.6.3	Notes.....	171
E.6.4	Model description.....	171
E.6.5	Rules.....	171
E.6.6	Example.....	172
E.7	DME_INTERROGATION	172
E.7.1	Definition	172
E.7.2	Interface properties.....	173
E.7.3	Notes.....	173
E.7.4	Model description.....	174
E.7.5	Rules.....	164
E.7.6	Example.....	174
E.8	DME_RESPONSE	175
E.8.1	Definition	175
E.8.2	Interface properties.....	176
E.8.3	Notes.....	177
E.8.4	Model description.....	177
E.8.5	Rules.....	179
E.8.6	Example.....	179
E.9	FM_SIGNAL<type: Voltage Power Current>	180
E.9.1	Definition	180
E.9.2	Interface properties.....	180
E.9.3	Notes.....	180
E.9.4	Model description.....	180
E.9.5	Rules.....	181
E.9.6	Example.....	181
E.10	ILS_GLIDE_SLOPE<type: Voltage Power>	182

E.10.1	Definition	182
E.10.2	Interface properties.....	183
E.10.3	Notes.....	184
E.10.4	Model description.....	184
E.10.5	Rules.....	185
E.10.6	Example.....	185
E.11	ILS_LOCALIZER<type: Power Voltage>	186
E.11.1	Definition	186
E.11.2	Interface properties.....	186
E.11.3	Notes.....	187
E.11.4	Model description.....	187
E.11.5	Rules.....	188
E.11.6	Example.....	188
E.12	ILS_MARKER	189
E.12.1	Definition	189
E.12.2	Interface properties.....	190
E.12.3	Notes.....	190
E.12.4	Model description.....	190
E.12.5	Rules.....	191
E.12.6	Example.....	191
E.13	PM_SIGNAL	191
E.13.1	Definition	191
E.13.2	Interface properties.....	192
E.13.3	Notes.....	192
E.13.4	Model description.....	192
E.13.5	Rules.....	193
E.13.6	Example.....	193
E.14	PULSED_AC_SIGNAL<type: Current Power Voltage>	194
E.14.1	Definition	194
E.14.2	Interface properties.....	194
E.14.3	Notes.....	194
E.14.4	Model description.....	195
E.14.5	Rules.....	195
E.14.6	Example.....	195
E.15	PULSED_AC_TRAIN<type: Voltage Current Power>	196
E.15.1	Definition	196
E.15.2	Interface properties.....	196
E.15.3	Notes.....	197
E.15.4	Model description.....	197
E.15.5	Rules.....	197
E.15.6	Example.....	198
E.16	PULSED_DC_SIGNAL<type: Voltage Current Power>	198
E.16.1	Definition	198
E.16.2	Interface properties.....	199
E.16.3	Notes.....	199
E.16.4	Model description.....	199
E.16.5	Rules.....	200
E.16.6	Example.....	200
E.17	PULSED_DC_TRAIN<type: Voltage Current Power>	201
E.17.1	Definition	201
E.17.2	Interface properties.....	201
E.17.3	Notes.....	201
E.17.4	Model description.....	202
E.17.5	Rules.....	202

E.17.6	Example.....	202
E.18	RADAR_RX_SIGNAL	203
E.18.1	Definition	203
E.18.2	Interface properties.....	203
E.18.3	Notes.....	204
E.18.4	Model description.....	204
E.18.5	Rules.....	205
E.18.6	Example.....	205
E.19	RADAR_TX_SIGNAL<type: Current Voltage Power>	206
E.19.1	Definition	206
E.19.2	Interface properties.....	206
E.19.3	Notes.....	207
E.19.4	Model description.....	207
E.19.5	Rules.....	207
E.19.6	Example.....	207
E.20	RAMP_SIGNAL<type: Voltage Current Power>	208
E.20.1	Definition	208
E.20.2	Interface properties.....	209
E.20.3	Notes.....	209
E.20.4	Model description.....	209
E.20.5	Rules.....	209
E.20.6	Example.....	210
E.21	RANDOM_NOISE	210
E.21.1	Definition	210
E.21.2	Interface properties.....	210
E.21.3	Notes.....	211
E.21.4	Model description.....	211
E.21.5	Rules.....	211
E.21.6	Example.....	211
E.22	RESOLVER	212
E.22.1	Definition	212
E.22.2	Interface properties.....	213
E.22.3	Notes.....	213
E.22.4	Model description.....	213
E.22.5	Rules.....	214
E.22.6	Example.....	215
E.23	RS_232.....	215
E.23.1	Definition	215
E.23.2	Interface properties.....	215
E.23.3	Notes.....	216
E.23.4	Model description.....	216
E.23.5	Rules.....	216
E.24	SQUARE_WAVE<type: Current Voltage Power>	216
E.24.1	Definition	216
E.24.2	Interface properties.....	217
E.24.3	Notes.....	217
E.24.4	Model description.....	217
E.24.5	Rules.....	218
E.24.6	Example.....	218
E.25	SSR_INTERROGATION<type: Voltage Current Power>	219
E.25.1	Definition	219
E.25.2	Interface properties.....	219
E.25.3	Notes.....	220
E.25.4	Model description.....	221

E.25.5	Rules	221
E.25.6	Example	221
E.26	SSR_RESPONSE<type: Voltage Current Power>	222
E.26.1	Definition	222
E.26.2	Interface properties	222
E.26.3	Notes	223
E.26.4	Model description	224
E.26.5	Rules	225
E.26.6	Example	225
E.27	STEP_SIGNAL	226
E.27.1	Definition	226
E.27.2	Interface properties	226
E.27.3	Notes	227
E.27.4	Model description	227
E.27.5	Rules	227
E.27.6	Example	227
E.28	SUP_CAR_SIGNAL	228
E.28.1	Definition	228
E.28.2	Interface properties	228
E.28.3	Notes	229
E.28.4	Model description	229
E.28.5	Rules	230
E.28.6	Example	230
E.29	SYNCHRO	230
E.29.1	Definition	230
E.29.2	Interface properties	231
E.29.3	Notes	232
E.29.4	Model description	232
E.29.5	Rules	233
E.29.6	Example	233
E.30	TACAN	234
E.30.1	Definition	234
E.30.2	Interface properties	235
E.30.3	Notes	236
E.30.4	Model description	236
E.30.5	Rules	238
E.30.6	Example	238
E.31	TRIANGULAR_WAVE_SIGNAL<type: Voltage Current Power>	239
E.31.1	Definition	239
E.31.2	Interface properties	239
E.31.3	Notes	240
E.31.4	Model description	240
E.31.5	Rules	240
E.31.6	Example	240
E.32	VOR	241
E.32.1	Definition	241
E.32.2	Interface properties	242
E.32.3	Notes	242
E.32.4	Model description	243
E.32.5	Rules	244
E.32.6	Example	244
	Annex F (informative) IDL for TSF for ATLAS	245

F.1	Introduction.....	245
F.2	IDL for TSF for ATLAS library.....	245
Annex G (normative) Carrier language requirements		265
G.1	Carrier language requirements.....	265
G.1.1	General requirements	265
G.1.2	Human interface and communication.....	265
G.2	IDL.....	265
G.3	Data types	265
G.3.1	Enumeration data type.....	266
G.3.2	Integer data type	266
G.3.3	Real data type	266
G.3.4	Character data type.....	267
G.3.5	Boolean data type.....	267
G.3.6	File data type	267
G.3.7	Array data type	267
G.3.8	Record data type.....	267
G.3.9	Variables and constants	267
G.4	Data-processing requirements.....	267
G.4.1	Data manipulation	267
G.4.2	Arithmetic operators.....	268
G.4.3	Relational operators.....	268
G.4.4	Logical operators	268
G.4.5	Other operators	269
G.4.6	Mathematical functions	269
G.4.7	File-handling functions	269
G.4.8	Type conversion functions	270
G.4.9	String related functions	270
G.4.10	Other functions	271
G.5	Control structures.....	271
G.5.1	If.....	271
G.5.2	Else	271
G.5.3	Case	271
G.5.4	For	271
G.5.5	While	271
Annex H (normative) Test procedure language (TPL).....		273
H.1	TPL layer	273
H.2	Elements of the TPL	273
H.3	Structure of test requirements.....	273
H.4	Carrier language.....	273
H.5	Signal statements	273
H.5.1	Definition of signal statements.....	273
H.5.2	Structure of signal statements	274
H.5.3	Syntax of signal statements	274
H.6	Mapping of test statements to carrier language	275
H.7	Test statement definitions	275
H.7.1	Setup statements.....	275
H.7.2	Reset statement.....	283
H.7.3	Connect statement	284
H.7.4	Disconnect statement.....	286
H.7.5	Enable statement	287

H.7.6	Disable statement	288
H.7.7	Read statement	289
H.7.8	Change statement	290
H.7.9	Compare statement	290
H.7.10	Wait_For statement	292
H.8	Elements used in test statement definitions	292
H.8.1	<TSFClass>	292
H.8.2	Attribute-Value groups	293
H.9	Attributes with multiple properties	300
H.9.1	Entering literal data	300
H.9.2	Using arrays of data	302
H.9.3	Acquiring sensor data	303
H.10	Transferring data in digital signals	304
H.10.1	Representation of digital data	304
H.10.2	Transmitting digital data using digital sources	305
H.10.3	Acquiring digital sensor data	307
H.10.4	Bidirectional digital signals	307
H.11	Creating test requirements	308
H.11.1	Creating test statements	308
H.11.2	Use of gate in signal statements	309
H.12	Delimiting TPL statements	310
H.12.1	Introducing a group of one or more TPL statements	310
H.12.2	Indicating end of group of TPL statements	310
Annex I (normative)	Extensible markup language (XML) signal descriptions (XSDs)	312
I.1	Introduction	312
I.2	XML signal schema definition	312
Annex J (informative)	XML for TSF for ATLAS	350
J.1	Introduction	350
J.2	TSF XML schema	350
J.2.1	Library information (<TSFLibrary> tag)	350
J.2.2	TSF information (<TSF> tag)	350
J.2.3	Interface information (<interface> tag)	351
J.2.4	Model information (<model> tag)	351
J.2.5	XML schema	352
J.3	XML for TSF for ATLAS	353
Annex K (informative)	Support for ATLAS nouns and modifiers	398
K.1	STD support for ATLAS signals	398
K.2	STD support for ATLAS nouns	398
K.3	STD support for ATLAS noun modifiers	401
K.3.1	Example of noun modifier supported by combination of BSCs	411
K.3.2	Example of noun modifier supported by a technique	411
K.4	Support for ATLAS extensions	411
Annex L (informative)	Bibliography	412
Annex M (informative)	List of participants	413

INTERNATIONAL ELECTROTECHNICAL COMMISSION

STANDARD FOR SIGNAL AND TEST DEFINITION

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 Publication(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. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
- 2) The formal decisions or agreements 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.
- 3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC 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 transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
- 5) IEC provides no marking procedure to indicate its approval and cannot be rendered responsible for any equipment declared to be in conformity with an IEC Publication.
- 6) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC/IEEE 62529 has been processed through Technical Committee 93: Design automation.

The text of this standard is based on the following documents:

IEEE Std	FDIS	Report on voting
1641(2004)	93/251/FDIS	93/262/RVD

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

The committee has decided that the contents of this publication will remain unchanged until the maintenance result date indicated on the IEC web site under "<http://webstore.iec.ch>" in the data related to the specific publication. At this date, the publication will be

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

IEC/IEEE Dual Logo International Standards

This Dual Logo International Standard is the result of an agreement between the IEC and the Institute of Electrical and Electronics Engineers, Inc. (IEEE). The original IEEE Standard was submitted to the IEC for consideration under the agreement, and the resulting IEC/IEEE Dual Logo International Standard has been published in accordance with the ISO/IEC Directives.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The 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 the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an IEC/IEEE Dual Logo International Standard is wholly voluntary. The IEC and IEEE disclaim liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEC or IEEE Standard document.

The IEC and IEEE do not warrant or represent the accuracy or content of the material contained herein, and expressly disclaim any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEC/IEEE Dual Logo International Standards documents are supplied "AS IS".

The existence of an IEC/IEEE Dual Logo International 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 IEC/IEEE Dual Logo International 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.

Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, 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 publishing and making this document available, the IEC and IEEE are not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Neither the IEC nor IEEE is undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEC/IEEE Dual Logo International Standards or IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations – Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received 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 interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of IEC/IEEE Dual Logo International Standards are welcome from any interested party, regardless of membership affiliation with the IEC or IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA and/or General Secretary, IEC, 3, rue de Varembe, PO Box 131, 1211 Geneva 20, Switzerland.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, 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.

NOTE – 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 with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

IEEE Standard for Signal and Test Definition

Sponsor

**IEEE Standards Coordinating Committee 20 on
Test and Diagnosis for Electronic Systems**

Approved 2 February 2005

American National Standards Institute

Approved 23 September 2004

IEEE-SA Standards Board

Abstract: This standard provides the means to define and describe signals used in testing. It also provides a set of common basic signals, mathematically underpinned so that signals can be combined to form complex signals usable across all test platforms.

Keywords: ATE, ATLAS, automatic test equipment, signal definitions, test definitions, test requirements, test signals, unit under test, UUT

IEEE Introduction

This standard is the culmination of a radical review of the Abbreviated Test Language for All Systems (ATLAS) and the requirement to create truly portable test requirements. During the review process, it was determined that it would be impractical to revise the existing ATLAS standard to include the required improvements. The decision was made to formulate a new standard to resolve these issues.

The key feature of the signal and test definition (STD) in this standard is the ability to unambiguously define test signals. It includes a rigorous mathematical and definitive foundation for all of its signal components. Any signal defined using this standard will be the same whatever equipment is used to create it. The standard supports the implementation of new technologies by providing users with the ability to describe their own signals by combining existing signals. Thus, any desired signal may be described, and there is no limit on the extensibility of signals supported by this standard.

Signals defined using STD can be used in a programming environment of the user's choice provided that that environment fulfills the minimum requirements stated in this standard. This universality enables the user to take full advantage of modern program structures and development environments, including graphical programming environments.

This standard was developed by the Test Description Subcommittee, whose intention is to prepare a companion guide to explain how to implement signal definitions and test requirements in conformance with STD.

Notice to users

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: <http://standards.ieee.org/reading/ieee/updates/errata/index.html>. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations can be accessed at the following URL: <http://standards.ieee.org/reading/ieee/interp/index.html>.

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 with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents or patent applications for which a license may be required to implement an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

STANDARD FOR SIGNAL AND TEST DEFINITION

1. Overview

This standard, also known as the signal and test definition (STD) standard, is the culmination of a radical review of the Abbreviated Test Language for All Systems (ATLAS) test programming language and the requirement to create truly portable test requirements. STD will allow test information to pass more freely between the design, test, and maintenance phases of a project, enabling the same information to be used directly across project phases. This more efficient use of information will lead to reduced life-cycle costs.

STD provides the capability to describe and control signals, while permitting a choice of operating environment, including the choice of carrier language. STD permits signal operations to be embedded in any object-oriented environment and thus to be used by the architecture standards of various automatic test systems (ATSS).

Portability is extended beyond that of test specifications by virtue of a layered architecture.

STD defines a collection of objects and their interfaces. These objects describe signal components relevant to test requirements. The STD standard defines how to interconnect these objects using interfaces, through which the objects exchange information, so that a test model may be defined that describes actual test requirements.

Finally, the link to published ATLAS standards (such as IEEE Std 716™-1995 [B9]¹) is preserved in that the user can describe signal operations using very similar test-signal-related keywords. These keywords now have formal definitions. Furthermore, the parameters of the signals themselves also have a rigorous formal behavioral description.

1.1 Scope

This standard provides the means to define and describe signals used in testing. It also provides a set of common basic signals, mathematically underpinned so that signals can be combined to form complex signals usable across all test platforms. The provision of language elements supports test signal descriptions for interoperability.

¹The numbers in brackets correspond to the numbers of the bibliography in Annex L.

This standard is divided into seven clauses:

- Clause 1 provides an introduction to this standard.
- Clause 2 provides definitions of terms and lists abbreviations.
- Clause 3 describes the structure of the STD standard.
- Clause 4 specifies the signal modeling language (SML).
- Clause 5 specifies the STD basic signal components (BSCs).
- Clause 6 defines the test signal frameworks (TSFs).
- Clause 7 describes the test procedure language (TPL) layer.

This standard also contains the following annexes:

- a) Annex A provides the Signals Modeling Language that is used to construct the BSCs and the TSFs.
- b) Annex B provides BSC descriptions.
- c) Annex C provides dynamic signal model description, states, and state transitions.
- d) Annex D provides the interface definition language (IDL) description for the BSCs.
- e) Annex E provides a TSF. This framework provides a formal description of signals similar to the signals defined in IEEE Std 716-1995. It also serves to illustrate how complex test signal models can be built up from BSCs.
- f) Annex F provides the IDL description for the TSF provided in Annex E.
- g) Annex G defines the requirements for a carrier language.
- h) Annex H provides the formal TPL description.
- i) Annex I provides the extensible markup language (XML) description mapping signal models into XML descriptions.
- j) Annex J provides XML description mapping the TSF provided in Annex E into XML descriptions.
- k) Annex K provides a description of how ATLAS nouns and noun modifiers are supported by STD.
- l) Annex L provides a bibliography of related documents.

1.2 Purpose

The purpose of this standard is to provide a common signal reference for use throughout the life cycle of a unit under test (UUT) or test system. Such a reference will in turn facilitate information transfer, test reuse, and broader application of test information—accessible through commercially available development tools.

2. Definitions, abbreviations, and acronyms

2.1 Definitions

For the purposes of this document, the following terms and definitions apply. *The Authoritative Dictionary of IEEE Standards Terms* [B8] should be referenced for terms not defined in this clause.

2.1.1 Abbreviated Test Language for All Systems (ATLAS): A stylized, abbreviated English language used in the preparation and documentation of test requirements and test programs, which can be implemented either manually or with automatic or semi-automatic test equipment.

2.1.2 argument: Input values that can be passed to a function.

2.1.3 attribute: A property value that is used to define signal characteristics or behaviour.

2.1.4 automatic test system (ATS): A system that includes the automatic test equipment (ATE) and all support equipment, support software, test programs, and interface adapters.

2.1.5 base class: A class from which another class inherits attributes or properties.

2.1.6 basic signal component (BSC): The lowest level of building block used to define signals.

2.1.7 class: A generic set of predefined abstract test objects.

2.1.8 component: A part of a system, which may be hardware or software and which may be subdivided into other components. Components communicate their functionality through their interface definitions.

2.1.9 connection: The application of a signal to a unit under test (UUT).

2.1.10 data bus: A signal line or set of signal lines used by a data communication system to interconnect a number of devices and to communicate information.

2.1.11 dynamic signal: A signal whose definition changes over time, by use of the control interface. These changes must be initiated with one of the signal method calls or by changing the interconnections of a signal model.

2.1.12 function: A construct that is a logically separated block of code that operates upon test values (i.e., arguments). Another name for a function is method. *Syn:* **method**.

2.1.13 interface definition language (IDL): A machine-compilable language that is used to describe the interfaces that software objects call and object implementations provide. The language provides a neutral way to define software interfaces.

2.1.14 method: *Syn:* **function**.

2.1.15 model: A mathematical or physical representation (i.e., simulation) of system relationships for a process, device, or concept.

2.1.16 physical: Pertaining to the natural characteristics of the universe according to the natural laws of science.

2.1.17 procedural: The part of an signal and test definition (STD) test requirement that defines the tests in the manner and order required for testing.

2.1.18 property: The special form of method (or function) that supports the semantics of assignment (l-value) and reading (r-value).

2.1.19 reserved word: A keyword whose meaning and use are fixed by the semantics of a language. In certain or all contexts, a reserved word cannot be used for any other purpose than as defined for that language.

2.1.20 semantics: A branch of linguistics concerned with meaning. For the test procedure language (TPL), semantics is the connotative meaning of words in an TPL statement. For software, semantics is the relationships of symbols and their meaning, independent of the manner of their interpretation and use. For meta-languages, semantics is the discipline for expressing the meanings of computer-language constructs in a meta-language.

2.1.21 sensor: A transducer that converts a test parameter to a form suitable for measurement.

2.1.22 SignalFunction: The name of the base class, for all classes that provide signals.

2.1.23 SubClass: A class that inherits attributes or properties from a base class.

2.1.24 static signal: A signal whose definition does not change over time. All basic signal components (BSCs) and test signal frameworks (TSFs) are static signals.

2.1.25 syntax: The grammatical arrangement of words in a language statement.

2.1.26 system: A set of interconnected hardware and/or software entities that achieves a defined objective by performing specified functions.

2.1.27 system architecture: The structure of and relationship between the entities of a system. A system architecture may include the system interface with its operational environment.

2.1.28 template: A pattern or design that establishes the outline, dimensions, or process for subsequent users or implementers.

2.1.29 test: An action or group of actions that are performed on a unit under test (UUT) to evaluate its parameter(s) or characteristic(s).

2.1.30 test requirement: A definition of the tests and test conditions required to be performed on a unit under test (UUT) to verify conformance with its performance specification.

2.1.31 test specification: A definition of the tests to be performed on a unit under test (UUT) to verify conformance with its performance specification, with or without fault diagnostics, and without reference to any specific test equipment.

2.1.32 test procedure: A description of the tests, test methods, and test sequences to be performed on a unit under test (UUT) to verify conformance with its test specification, with or without fault diagnostics, and without reference to specific test equipment.

2.1.33 test program: An implementation of the tests, test methods, and test sequences to be performed on a unit under test (UUT) to verify conformance with its test specification, with or without fault diagnosis. A test program is configured for execution on a specific test system.

2.1.34 transducer: A device that converts a physical magnitude of one form of energy into another form, generally on a one-to-one correspondence or according to a specified mathematical formula.

2.1.35 unit under test (UUT): An entity that can be tested and that may range from a single component to a complete system.

2.1.36 value: The quantitative size of a signal attribute.

2.2 Abbreviations and acronyms

ARB	auxiliary reference burst
ARINC	Aeronautical Radio, Inc. ²
ASCII	American Standard Code for Information Interchange

²ARINC provides engineering specifications and services to airlines, airframe constructors, and avionics suppliers.

ATC	air traffic control
ATE	automatic test equipment
ATLAS	Abbreviated Test Language for All Systems
ATS	automatic test system
BSC	basic signal component
C/ATLAS	Common/Abbreviated Test Language for All Systems
DME	distance measuring equipment
IDL	interface definition language
IFF	identification, friend, or foe
ILS	instrument landing system
MRB	main reference burst
PCS	piecewise continuous signal
PRF	pulse repetition frequency
RF	radio frequency
rms	root mean square
SML	signal modeling language
SSR	secondary surveillance radar
STANAG	NATO Standardization Agreements
STD	signal and test definition
TACAN	tactical air navigation
TPL	test procedure language
TSF	test signal framework
UHF	ultrahigh frequency
UUT	unit under test
VHF	very high frequency
VOR	VHF omnidirectional range
XMD	XML signal definition
XML	extensible markup language

3. Structure of this standard

3.1 Layers

This standard has a layered format depicted below in Figure 1. Each layer and its function are described in this clause. Each layer builds on items defined in previous layers. This format does not require that each layer use only items in its immediate lower level, but does imply that each layer has to be fully defined in terms of its lower level layers.

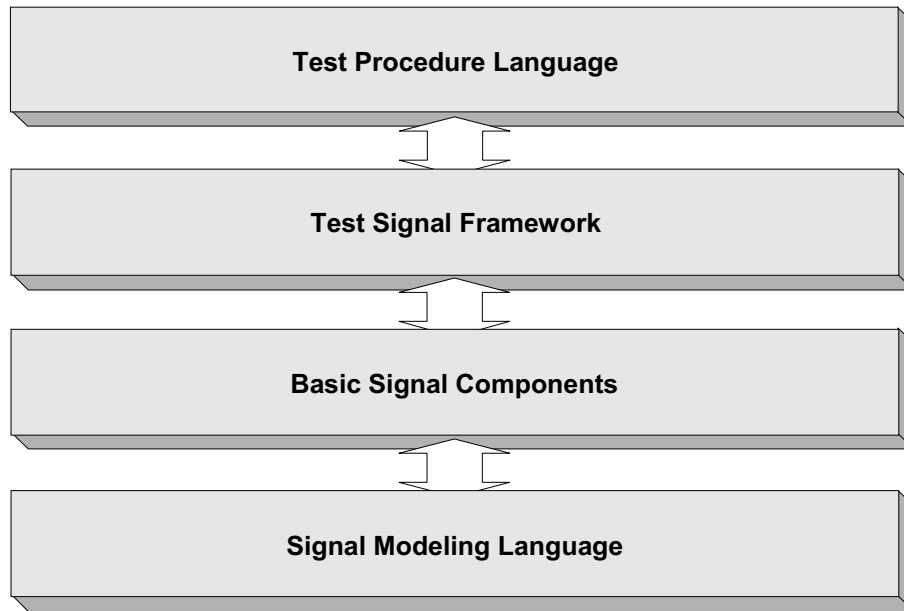


Figure 1—STD layers

The layer model provides additional support features over a purely textual language. This standard provides the capability to describe and control signals and allows the user to choose the operating environment, including the choice of programming language. The STD test requirements take the form of signal definitions, which can be written in any programming language or object-oriented environment. The link to ATLAS and Common/Abbreviated Test Language for All Systems (C/ATLAS) standards is preserved as test requirements are mapped to signals that have formal definitions tied to the TSF and BSC layers.

Throughout all the layers, there exists the concept of signals. Signals are characterized by their type, which may be typeless or may map onto a physical property. Examples of typeless signals are events or generic (such as SUM), and an example of a type that maps onto a physical property is voltage, such as in a Sinusoidal signal of type Voltage. The type of a signal also includes the reference type; most signals encountered in this standard have the reference type Time. Therefore, a Sinusoidal Voltage signal describes how the signal's voltage changes with respect to time, in a sinusoidal manner. The use of types is such that signals can be combined only when they have the same types or when one of the types is typeless or generic.

Extendibility is served by providing the capability to describe new signals formally by creating them from the existing signals in either the TSF or BSC layers.

3.1.1 SML layer

The SML layer provides the mathematical definitions that support the description of BSCs. This mathematical underpinning provides evidence that the signals are defined by BSCs. The SML signal definitions form the basis for reuse that is essential to the extension of STD capabilities without a corresponding explosion in nomenclature and complexity.

3.1.2 BSC layer

The BSC layer provides reusable, formally described, fundamental signal classes. These classes define the lowest level of signal building blocks available to the STD environment. Each BSC is described by its class name, class type, properties and default values, IDL description, and SML signal definition.

3.1.3 TSF layer

The TSF layer identifies how libraries of reusable, formally described signal classes are defined. The content of a TSF library is a collection of domain-specific signal definitions made up from other TSF signals and BSCs.

The TSF is the extendibility mechanism that allows the creation of additional signal class definitions. Each TSF class is described by its class name, class types, properties and default values, IDL description, and a static signal model definition.

3.1.4 TPL layer

The TPL layer allows test descriptions (e.g., test requirements, test procedures, or test programs) to be formally described in a textual format by combining STD signals with features that satisfy the carrier language requirements.

Test specifications and signal libraries conforming to the requirements of this layer may be ported between different ATSS with the same functional capability and carrier language. Minimal translation would be required to convert between different carrier languages.

4. Signal modeling language (SML) layer

The SML layer allows the definition of signals, both analog and digital, as well as their functions in any number of domains. It provides this capability by giving a number of predefined behaviors that can be combined as necessary to produce the desired signal definition. This clause describes the use of SML to define signals, the measurement of signal parameters, and the conditions that a signal must meet.

The SML provides an exact mathematical definition for each BSC, in terms of dependant and independent variables, by using the de-facto functional programming concepts of Haskell 98. This definition represents the functioning of an entity, which is a requirement for component use and reuse. This is accomplished by giving a formal definition of the syntax and predefined signals. An execution mechanism may be provided for simulating the modeled signal, plotting against its definition domain, and measuring its various properties. Within the SML layer, the type of a signal is known as the *dependant variable*, and the reference type is known as the *independent variable*.

5. Basic signal component (BSC) layer

This clause describes the methodology adopted to define signals with BSCs and the mechanisms by which they may be combined and synchronized.

5.1 BSC layer base classes

All BSC classes used to define signals are derived from one of the base classes shown in Table 1. This class approach is useful for categorizing BSCs according to their characteristics, behavior, and interfaces.

Table 1—BSC base classes

Base class	Description
SignalFunction	The base class of all BSCs
Signal	Allows BSCs to exchange information
Resource	Creates the BSC objects
PulseDefns	Defines a group of pulses
Physical	Real, dimensioned signal values

5.2 BSCs

BSCs are the fundamental components of this standard. The BSCs are the building blocks used to define more complex signals and cannot be decomposed into simpler components.

BSCs are used to build signal models, which define the required signal. A signal model can contain a single BSC to define a simple signal or combined BSCs to define a more complex signal.

BSCs can be used to either define static signal models or perform dynamic signal programming by programmatically changing signal models through a programming language.

Signal models represent static signal descriptions, where the signal model does not change over time (i.e., reference type). The BSC control interface (i.e., the IDL description) can also be used to define dynamic signal definitions, where the value of the attributes or the signal model changes while the signal is being used.

The signal described by a signal model can be used to create a source signal or to measure a signal characteristic attribute.

Unless otherwise stated, the default reference type for a BSC is time; and where required, the default signal type is voltage.

Each BSC is described using object orientation terminology as follows:

- A class derived from **SignalFunction** base class (or subclass)
- Class type and reference type description
- Attributes and default values
- A control interface (defined using an IDL description)
- A formal SML description

The following annexes describe BSC features:

- Annex A gives details of the BSC formal SML descriptions.
- Annex B gives details of the BSC classes and attributes.
- Annex C gives details of the dynamic signal model behavior.
- Annex D gives a list of the IDL descriptions for each BSCs.
- Annex I gives a list of the XML descriptions for mapping signal models.

NOTE—Annex D and Annex I are normative annexes in that they provide the normative descriptions for the BSCs in IDL and XML, respectively. This does not mean that the BSCs may not be described in other interface languages.³

5.3 SignalFunction template

A template is used as document shorthand to define types of derived classes of **SignalFunction**, where each class has similar behavior and supports the same IDL description, but describes signals of different types. Note the class name is always the same; only the class type changes.

The use of the template defines alternative class types, where each derived class name could equally have been written using “cut & paste” and replacing the keyword “**type**” with any physical class defined within this standard. Substituting `<type:...[,ref:...]>` with each alternative creates the class type that the template defines. The specific type alternatives provided in the template identify the more commonly used signal classes; once the template is defined, any type or reference can be used.

The format for the template header is as follows:

```
ClassName<type:typeName[|typeName]*[,ref:typeName[|typeName]*]>
```

Where no explicit types are provided, the default type is voltage, and the default reference type (ref) is time.

For example, **Sinusoid**<type:**Voltage**||**Current**||**Power**> defines the following classes, where each class supports the **Sinusoid** interface, through the IDL definition:

- a) **Sinusoid (Voltage)** full type Sinusoid (Voltage, Time)
- b) **Sinusoid (Current)** full type Sinusoid (Current, Time)
- c) **Sinusoid (Power)** full type Sinusoid (Power, Time)

NOTES

1—Alternatives are separated by the double-bar characters (||).

2—This template convention is adopted in the remainder of this standard.

6. Test signal framework (TSF) layer

The TSF layer describes how BSCs are combined into more complex signals and packaged for reuse in TSF libraries.

The TSF layer also provides a packaging mechanism for grouping signal models into library elements. These libraries are constructed from individual TSF classes where each class supplies an interface definition in IDL and a static signal model description. A TSF library will generally be a collection of domain-specific signal definitions.

If the TSF class is to be used utilizing the XML descriptions in the Require method or TPL setup statement, then the TSF class shall exist in an XML TSF library conforming to Annex J.

6.1 TSF classes

A TSF library shall be defined in terms of an IDL library module and will contain an entry for each TSF class within the library domain.

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

A TSF class shall define the IDL definition for its class and its attribute interface.

A TSF class shall be described only as a static signal model and may contain elements from the following:

- BSCs
- Other TSF classes

A TSF class's signal description utilizing a TSF component is identical to a signal description incorporating the complete TSF static signal model. As such, the following statements are true:

- A TSF component output is available only when its **Gate** (see Annex B) event is on.
- A TSF component restarts its operation when its **Sync** (see Annex B) event arrives.

TSFs provide the extendibility mechanism that allows the user community to create additional signal class definitions.

6.2 TSF signals

Each signal described in a TSF shall have the following information:

- Title, which indicates the syntax of the TSF class name (in upper case characters with underscores).
- Definition of the signal.
- Model diagram, which shows the component parts of the signal and their relationship. This diagram is provided to give a convenient pictorial representation of the signal model. In the event of any conflict between the model diagram and the model description table, the model description table takes precedence.
- Interface properties table listing the TSF interface properties.
- Notes as needed for any additional explanations.
- Model description table, which lists the component parts of the signal and their relationship. Each element in the model is supported by a definition in the BSC.
- Equations as needed to define the operation of the model.
- Rules as needed relating to the operation of the model.

6.2.1 Interface properties table

The interface properties table shall comprise of the following five columns:

- *Description*. The descriptive name of the attribute.
- *Name*. The syntactical name of the attribute as defined in the BSC.
- *Type*. The attribute type as defined in the BSC. If the type is given as physical, the actual type shall be chosen from the list provided with the signal title. If more than one attribute has the type given as physical, then all attributes shall be of the same type.
- *Default*. If a value is provided, the default is the value that the attribute will take if the attribute is not specifically defined. If no default value is given, then the user shall provide a value.
- *Range*. If a range is given, it indicates the valid range for the attribute. The attribute value must fall within this range.

Table 2 shows an interface property table for a sample signal (in this example, AC_SIGNAL). It indicates that the user should provide the physical type, amplitude, and frequency as a minimum. The physical type is usually determined from the units given with the value; for example, an amplitude of 10 V indicates that the physical type is voltage. If no DC Offset or initial phase angle is defined, each attribute will assume the default values given in the table. Note that the DC Offset is of the same type as the AC Signal amplitude.

Table 2—Example of TSF interface property table

Description	Name	Type	Default	Range
AC Signal amplitude	ac_ampl	Physical		
DC Offset	dc_offset	Physical	0	
AC Signal frequency	freq	Frequency		
AC Signal phase angle	phase	PlaneAngle	0 rad	0–2 π rad

An attribute of a TSF class that is subsequently not used in the signal model description cannot be used to change or control the signal. It may be used to describe the capabilities required from the signal; for example, Distortion Max 3% would mean distortion must be less than (or equal to) 3%. As such, all such capability attributes are optional.

6.2.2 Model description table

The model description table describes the signal model using a network list format. The model description table shall comprise of the following six columns:

- *Name*. The given name of the BSC or TSF within the model.
- *Type*. The type of the BSC or TSF.
- *Terminal*. The terminal names of the BSC or TSF, usually in the order of outputs followed by inputs.
- *Inputs*. The signal or attribute that is connected to the terminal listed in the Terminal column.
- *Output*. The output(s) of the BSC or TSF. Some BSC or TSF signal outputs will be inputs to other BSCs or TSF signals within the model, and at least one will be the output from the model.
- *Formula*. An optional mathematical definition (or constant value) of the function or input.

Table 3 shows a signal model for AC_SIGNAL. A BSC of type Sinusoid (named AC Component) is summed with a BCS of type Constant (named DC Offset) to create the AC_SIGNAL.

Table 3—Example of TSF model description table

Name	Type	Terminal	Inputs	Output	Formula
AC Signal	Sum	Signal [Out]		AC_SIGNAL	
		Signal [In]	DC Offset		
		Signal [In]	AC Component		
AC Component	Sinusoid	Signal [Out]		AC Signal	
		amplitude	ac_ampl		
		frequency	freq		
		phase	phase		
DC Offset	Constant	Signal [Out]		AC Signal	
		amplitude	dc_offset		

6.2.3 TSF figures

Each TSF figure provides a pictorial description of the model description and interface properties. These figures give an intuitive representation of a signal model. They do not infer the use of any specific signal resources. If the TSF figure is not consistent with the model description table or the interface properties table, then the tables take precedence.

7. Test procedure language (TPL) layer

The TPL layer provides a mechanism for users who want to document test requirements in a textual format. The use of the TPL to write test requirements is analogous to using C/ATLAS inasmuch as the TPL uses stylized English signal statements to describe tests and to manipulate signals. It differs from using C/ATLAS in that it does not provide a fully defined programming language. Instead, STD allows users to adopt their own preferred programming language in which the signal statements and the underlying semantics of tests can be written.

7.1 Goals of the TPL

The goals of the TPL are as follows:

- a) Its keywords have meanings that are normally accepted by the worldwide testing community.
- b) It is an effective means for communicating test information relating to the testing of a UUT between an originator of a test requirement and an implementer of a test requirement.
- c) Test requirements written according to the TPL rules shall be portable to implementations on different designs of test equipment that have the same testing capability no matter how it is controlled.

7.2 Elements of the TPL

The TPL comprises two elements:

- a) Signal statements, which are used to configure, manipulate, control, and measure signals.
- b) Carrier language, which is a programming language in which the signal statements can be written, sequenced, observed, and generally supported.

7.3 Use of the TPL

To produce test requirements using the TPL, users shall embed the test statements in their preferred carrier language.

Users must recognize that there must be some translation mechanism to convert from this neutral format of the signal statements into their preferred carrier language format before the test statements can be compiled and executed.

Use of a translator to convert the neutral representation of the test statements into the carrier language format offers certain benefits in that parameter type checking and semantics checks can be conducted prior to test execution.

Annex A

(normative)

Signal modeling language (SML)

A.1 Use of the SML

In general, unless a user needs to generate new keywords for the test methodology, there is no requirement for a user to refer to the SML in order to generate test requirements for a UUT. Occasionally, a user may need to refer to the SML either for the mathematical justification of some signal construct or for the introduction of a new keyword to cover some new test application that the test methodology does not embrace. In the latter instance, a user will need to refer to the SML in order to ensure that any new keyword that is introduced into the test methodology is coherent with the existing predefined keywords.

For convenience of using processing software suites that are freely available without any restrictions to generate the signal diagrams, a derived version of the functional programming language Haskell has been adopted.

A.2 Introduction

This annex describes how the SML can define the signal characteristics, signal measurements, and signal conditions to meet particular applications. A SML signal model is a mathematical definition of the signal and its properties. The definition represents the functioning of a signal entity for both its use and reuse. It provides a formal definition of the syntax and semantics of predefined signal types. An execution mechanism may be provided for simulating a modeled signal by plotting it against its definition domain and measuring its various properties.

This annex provides and builds upon the following:

- Definitions for the BSCs
- Definitions for the combining mechanism for piecewise continuous signals (PCSs) and others

As a convention, the reserved words for the SML entities are written in *italics* in the format descriptions.

Functional programming consists of building definitions that are subsequently used to evaluate expressions. Expressions represent questions that evaluate to answers or values, using rules or functions that represent the definitions, all of which obey normal mathematical principles.

Values are partitioned into organized collections called *types*. Examples of predefined types are integer and float. All type names shall start with a capital letter in the format descriptions. The double colon symbol (`::`) is used to define the type of a function or expression.

Example:

```

pi :: Float -- means pi is of type Float
pi = 3.14159
succ :: Int -> Int -- function succ that takes an Int value and returns an Int value
succ n = n+1

```

User types can be defined using the keyword *data* that introduces the name of the type and the type values by using a type constructor. All type constructor names shall start with a capital letter in the format descriptions.

Example:

```
data Resistance = OHM Float | KOHM Float | MOHM Float
```

In other words, a value of type *Resistance* is written using one of the three type constructor keywords, *OHM*, *KOHM*, *MOHM*, followed by a value of type *Float*, e.g., 5 k is written *KOHM 5.0*.

Type classes can be defined using the keyword *class* that introduces the name of the type class and the allowed functions that operate on any type belonging to this type class. All type class names shall start with a capital letter in the format descriptions. The type class definition can also constrain the types that are allowed to belong to the type class by ensuring that they belong to other type classes.

Example:

```
class (Eq a, Show a) => Physical a where ...
```

In other words, a type class *Physical* is defined so that only types that belong to the type classes *Eq* and *Show* may belong to the new type class *Physical*. The language word *where* starts the scope of the remaining definition.

A type definition uses the keyword *instance* to declare itself a member of a particular type class.

Example:

```
instance Physical Resistance where ...
```

A.3 Physical types

All physical types are held in the module *Physical*:

```
>module Physical where
```

Table A.1 defines the physical types used within the SML and the units involved. The column headers of the units indicate the exponent of 10 that is used for the unit; in other words, if that unit is used, the basic degree is multiplied by 10 raised to that exponent. The first unit in each list is the basic unit, i.e., the unit in terms of which other units are defined.

All of the type names in Table A.1 may appear in type signatures that give the type of a signal. Each of the unit names may be used with a floating-point number or expression to create a physical constant. The form in this case is (<unit_name> <expression>), where the expression is given in normal mathematical notation.

The specification in this clause does not include a complete definition of operations on values of a physical type. Operations are conducted on normal floating-point types and then converted into physical quantities as necessary. Conversion of a physical value into a floating-point value is accomplished using the following form:

```
>class (Eq a, Show a) => Physical a where  
>   fromPhysical:: a -> Float
```

Floating-point values may be converted to physical values by prefixing them by one of the units in Table A.1. In some cases, the appropriate unit is not clear (e.g., in a very general signal creation method). In

these cases, a general routine is used to convert the floating-point value into a physical value whose type is determined from the context. This conversion is accomplished by the following form:

```
> toPhysical:: Float -> a
```

All physical types are defined with a data declaration and the instance mapping *fromPhysical* and *toPhysical*.

In Table A.1 and Table A.2, the physical types are declared in the SML type column, and the alternative constructor names are provided in the other SML columns. Standard physical conversions may need to be used when multiple units are used.

Examples:

```
>data PlaneAngle = RAD Float | MRAD Float| URAD Float |
>                  DEG Float |
>                  REV Float deriving (Eq, Show)
>instance Physical PlaneAngle where
>    fromPhysical (RAD x) = x
>    fromPhysical (MRAD x) = x * 1.0e-3
>    fromPhysical (URAD x) = x * 1.0e-6
>    fromPhysical (DEG x) = x * (pi/180)
>    fromPhysical (REV x) = x * (2*pi)
>    toPhysical x = RAD x

>data Resistance = OHM Float | KOHM Float| MOHM Float deriving (Eq, Show)
>instance Physical Resistance where
>    fromPhysical (OHM x) = x
>    fromPhysical (KOHM x) = x * 1000
>    fromPhysical (MOHM x) = x * 1000000
>    toPhysical x = OHM x
```

Table A.1—SML physical types and their units

SML type	Unit	S/A*	10 ⁰ SML	10 ³ SML	10 ⁶ SML	10 ⁹ SML	10 ⁻³ SML	10 ⁻⁶ SML	10 ⁻⁹ SML	10 ⁻¹² SML
PlaneAngle	radian	rad	RAD				MRAD	URAD		
	degree	°	DEG							
	revolution		REV							
SolidAngle	stera- dian	sr	SR				MSR			
Capaci- tance	farad	F	FD					UFD	NFD	PFD
Charge	cou- lomb	C	C	KC				UC	NC	
Conduc- tance	sie- mens	S	S							
Current	ampere	A	A	KA			MA	UA	NA	

Table A.1—SML physical types and their units (*continued*)

SML type	Unit	S/A*	10 ⁰ SML	10 ³ SML	10 ⁶ SML	10 ⁹ SML	10 ⁻³ SML	10 ⁻⁶ SML	10 ⁻⁹ SML	10 ⁻¹² SML
Distance	meter	m	M	KM			MM	UM	NM	
	inch	in	IN							
	foot	ft	FT							
	stat. mile	mi	SMI							
	naut. mile	nmi	NMI							
Energy	joule	J	J	KJ			MJ			
	elec- tron- volt	eV	EV	KEV	MEV					
Magnetic- Flux	weber	Wb	WB				MWB			
Magnetic- Flux- Density	tesla	T	T				MT	UT		
Force	newton	N	N	KN			MN	UN		
Frequency	hertz	Hz	HZ	KHZ	MHZ	GHZ				
Illuminance	lux	lx	LX							
Inductance	henry	H	HEN				MH	UH	NH	PH
Luminance	can- dela per square meter	cd/m ²	NT							
Luminous- Flux	lumen	lm	LM							
Luminous- Intensity	candela	cd	CD							
Mass	kilo- gram	kg	KG				G	MG	UG	
Power	watt	W	W	KW			MW	UW		
Pressure	pascal	Pa	PA	KPA			MPA	UPA		
	milli- bar	mbar	MB							
Resistance	ohm	Ω	OHM	KOHM	MOHM					

Table A.1—SML physical types and their units (continued)

SML type	Unit	S/A*	10 ⁰ SML	10 ³ SML	10 ⁶ SML	10 ⁹ SML	10 ⁻³ SML	10 ⁻⁶ SML	10 ⁻⁹ SML	10 ⁻¹² SML
Temperature	kelvin	K	DEGK							
	deg. Celsius	°C	DEGC							
	deg. Fahrenheit	°F	DEGF							
Time	second	s	SEC				MSEC	USEC	NSEC	
	minute	min	MIN							
	hour	h	HR							
Voltage	volt	V	V	KV			MV	UV		
Volume	liter	L	LITER				ML			

*S/A = symbol or abbreviation.

Table A.2 provides two further special physical types that are provided to support the SML.

Table A.2—Special physical types and their units

SML type	Unit	S/A*	10 ⁰ SML	10 ³ SML	10 ⁶ SML	10 ⁹ SML	10 ⁻³ SML	10 ⁻⁶ SML	10 ⁻⁹ SML	10 ⁻¹² SML
Burst-Length	cycle		CYCLE							
	pulse		PULSE							
RatioInOut	decibel	dB	DB							

*S/A = symbol or abbreviation.

A.4 Signal definitions

All SML primitive signal definitions are held in the module Pure and make use of the previous module Physical and the Haskell system modules Complex and Fourier.

```
>module Pure where
>import Physical
>import Fourier
>import Complex
>import Random
>infixr 7 |>
```

Any data type can declare itself as a signal by declaring itself an instance of the class Signal.

```
>class Signal s where
> mapSignal:: (Physical a, Physical b) => (s a b) -> a -> b
> mapSigList:: (Physical a, Physical b) => (s a b) -> [a] -> [b]
> toSig:: (Physical a, Physical b) => (s a b) -> SignalRep a b
> isNull:: (Physical a, Physical b) => (s a b) -> a -> Bool
```

```

> mapSignal = mapSignal . toSig
> mapSigList = map . mapSignal
> toSig = FunctionRep . mapSignal
> isNull _ _ = False

```

An instance of a signal can be observed at a specific point in its domain, effectively calling the function associated with the signal by providing an argument to the function. This capability is referred to as mapping the signal onto a specific point and has the following form:

mapSignal <signal_name> <independent_value>

A common signal representation is used to define all signals. A signal representation can be represented

- By a function, or
- As piecewise continuous windows made up of other signal representations, or
- By the value Not Present and detected through the use of the *isNull* method.

```

>data SignalRep a b =
>  NullRep |
>  FunctionRep (a -> b) |
>  PieceContRep (PieceCont a b)

```

A signal representation is itself an instance of the class *Signal*.

```

>instance Signal SignalRep where
>  mapSignal NullRep = \t -> toPhysical 0.0
>  mapSignal (FunctionRep f) = f
>  mapSignal (PieceContRep f) = mapSignal f
>  mapSigList (FunctionRep f) = map f
>  mapSigList (PieceContRep f) = mapSigList f
>  toSig = id
>  isNull NullRep _ = True
>  isNull (FunctionRep f) _ = False
>  isNull (PieceContRep f) t = isNull f t

```

A new signal definition is created by either defining a function that returns a signal representation (e.g., *SignalRep a b*) or creating a new data class that supports the signal class interface. When defining a new signal, the name may be used twice in the definition:

- First, to give the type of the independent and dependent variables of the signal. This use is optional and can often be inferred from the use of the parameters in the signal definition.
- Second, to define the signal itself, using the mechanisms provided for basic signal definitions.

A.5 Pure signals

The SML provides a number of mathematically pure signal definitions, which represent the behavior without any representation of noise, distortion, or spurious phenomena. The user can take these signals and build more complex signals with them using the construction techniques. These pure signals are divided and presented as signals that are nonperiodic (i.e., do not have a given period or frequency) and signals that are periodic (i.e., have a given period or frequency).

A.5.1 Nonperiodic signals

The signals presented in this subclause have no implicit period. They identify specific, one-time events or functions that do not repeat themselves.

A.5.1.1 Constant

A constant signal retains its given level for all values of its independent variable. It has the following form:

```
>constant:: (Physical a, Physical b) => b -> SignalRep a b
>constant level = FunctionRep (\t -> level)
```

A.5.1.2 Linear

A linear signal forms a line within a plane. The line is defined by its slope and intercept. The equation is the standard $y = mx + b$. It has the following form:

```
>linear:: (Physical a, Physical b) => Float -> b -> SignalRep a b
>linear m b = FunctionRep (\x -> toPhysical (m*(fromPhysical x) + (fromPhysical b)))
```

A.5.1.3 Random

A random signal consists of an unbounded number of random levels between zero and one. It takes two parameters: an integer seed and a sampling interval, which is of the same type as the independent variable. The same random signal is given for the same seed; the seed enables deterministic testing. It has the following form:

```
>rand:: Integer -> [Float]
>rand i = randoms (mkStdGen (fromInteger i))
>
>random:: (Physical a, Physical b) => Integer -> a -> SignalRep a b
>random seed sample_interval = let
> waveform:: (Physical a, Physical b) => a -> [b] -> SignalRep a b
> waveform samp ampls =
>   let stepSlope y y' = ((fromPhysical y') - (fromPhysical y))/(fromPhysical samp)
>   makeWin (v,v') = Window LocalZero (TimeEvent (fromPhysical samp))
>   (linear (stepSlope v v') v)
>   points = cycle ampls
>   in pieceRep (Windows (map makeWin (zip points (tail points))))
>   in waveform sample_interval (map toPhysical (rand seed))
```

A.5.1.4 Exponential

An exponential is a damping factor, which is equivalent to the following function:

$$e^{-t/\tau}$$

where

- t is the time interval;
- τ is the damping factor.

An exponential allows any signal to be damped over a given time, according to a floating-point damping factor:

```
>expc:: (Physical a, Physical b) => Float -> SignalRep a b
>expc damp = FunctionRep (\t->toPhysical (exp (-((fromPhysical t)*damp))))
```

A.5.2 Periodic signals

The signals defined in this subclause have either a period or a frequency assigned to them. In other words, they repeat their values for some fixed value of their independent variable.

A.5.2.1 Sinusoid

A sinusoid is the familiar sine relationship. It takes an amplitude, a frequency, and a phase angle. The amplitude has the type of the dependent variable, the frequency is of type `Frequency`, and the phase angle is a `PlaneAngle`. The result is given as follows:

$$A * \sin(\omega t + \theta)$$

where

- A is the amplitude;
- ω is the frequency (multiplied by 2π);
- θ is the phase angle.

It has the following form:

```
>sine:: (Physical a, Physical b) => b -> Frequency -> PlaneAngle -> SignalRep a b
>sine mag omeg phase = FunctionRep (\x -> toPhysical ((fromPhysical mag)*
>      (sin(2*pi*(fromPhysical omeg)*(fromPhysical x) + (fromPhysical phase))))))
```

A sinusoid is a simpler form of a more complex function, whose amplitude, phase angle, and frequency are functions rather than scalars. This has the same format as that above:

```
>sineFunc:: (Physical a, Physical b) =>
>      SignalRep a b -> SignalRep a b -> SignalRep a b -> SignalRep a b
>sineFunc mag omeg phase = FunctionRep (\x -> toPhysical ((fromPhysical (mapSignal
mag x))*
>      (sin(2*pi*(fromPhysical (mapSignal omeg x))*(fromPhysical x)
>      + (fromPhysical (mapSignal phase x))))))
```

where the type of all three signals is from the independent to the dependent type.

A.5.2.2 Waveform

A waveform is defined by a sampling interval and a list of values. The waveform cycles through those values sequentially and infinitely, starting from zero. The width of each window is the same, and each window consists of a line segment.

A waveform has the following form

```
>waveform:: (Physical a, Physical b) => a -> [b] -> SignalRep a b
>waveform samp lev =
>   let s = fromPhysical samp
>       stepSlope y y' = ((fromPhysical y') - (fromPhysical y)) / s
>       makeWin (v,v') = Window LocalZero (TimeEvent s) (linear(stepSlope v v') v)
>       points = cycle lev
>   in pieceRep (Windows (map makeWin (zip points (tail points))))
```

A.6 Pure signal-combining mechanisms

Clause A.5 established a number of ways to create signals, and this clause presents some mechanisms of combining signals together. Each mechanism will be handled separately.

A.6.1 Piecewise continuous signals (PCSs)

A PCS is made up of a number of windows, each with its own signal defined within the window. Window boundaries are defined by events.

A.6.1.1 Window events

An event marks the transition from one window to another. An event can be a fixed amount of time (i.e., a time event), a function of another signal (i.e., function event), the moment when another signal becomes active (i.e., active event), or a fixed number of event occurrences (i.e., a burst event). Each type of event has a distinct form.

```
>data (Physical a, Physical b) => Event a b=
```

- a) The first kind of event is a time event, which is a specified period of time. The window lasts for the duration of time given. It has the following form:

```
> TimeEvent Float |
```

Although the type of event is called a time event, it may be of any physical type; thus, the use of the floating-point expression.

- b) The second kind of event is a function event. The argument of this function is a function that, in turn, takes a signal and produces a Boolean result. It takes the following form:

```
> FunctionEvent (Float -> Bool) |
```

- c) A third kind of event is an active event. The active event is when a signal representation transitions from a NullRep representation to a non-NullRep representation.

```
> ActiveEvent (SignalRep a b) |
```

- d) The fourth kind of event is the burst event. The burst event is triggered by a given number of triggers of another defined event. Its form is as follows:

```
> BurstEvent Int (Event a b)
```

A user may determine the time (or the value of the independent variable) when a given event occurs by the following expression:

```
>timeOccurs:: (Physical a, Physical b) => (Event a b) -> a
>timeOccurs e = toPhysical (eventOccurs e 0.0)
```

The Boolean function above may be any sort of function that takes physical value and produces a Boolean value. As an example, the transition functions are given below. The transition functions take a transition point, a signal, and a time (or value of the type of the independent variable of the signal) at which to perform the test; and they produce the value true if the signal has crossed that value since the last sample and the value false otherwise. The function hilo produces true on a falling edge, and the function lohi produces true on a rising edge. The forms of these functions are as follows:

```
hilo <transition_point> <signal_name> <test_point>
lohi <transition_point> <signal_name> <test-point>
```

A.6.1.2 Windows

A window is specified by an event, which gives the width of the window, and a function, which specifies the value of the signal within the window.

There is an additional complication when determining the width of a window. In some cases (e.g., the normal use of a time window), the beginning of the window is to be regarded as time 0.0 for the purposes of measuring the width and evaluating the signal. In other cases (e.g., selecting between two different signals at given points on a time line), it may be required to evaluate the signal against the global time zero (i.e., the

time zero of the entire PCS). Therefore, a flag is included that determines the zero against the event defining the width of the window and the value of the signal within that window.

The form of a window definition is, therefore, as follows:

```
>data FunctionWindow a b = Window ZeroIndicator (Event a b) (SignalRep a b)
```

A ZeroIndicator flag is one of two identifiers, LocalZero or GlobalZero, as follows:

```
>data ZeroIndicator = LocalZero | GlobalZero deriving (Eq, Show)
```

A.6.1.3 Piecewise continuous functions

A piecewise continuous function is built from windows and is an instance of a signal. The function getWindow is available to retrieve the local time and first window where the window event happens after the required time.

```
>getWindow:: (Physical a, Physical b) =>
>      Float -> Float -> [ FunctionWindow a b ] ->
>      (Float, FunctionWindow a b, [ FunctionWindow a b ])
>getWindow st t [] = (t, Window LocalZero (TimeEvent (2*t)) NullRep, [])
>getWindow st t (w:wl) = if t' < et then (t', w, wl)
>      else getWindow nt t wl
>      where et = eventOccurs e st'
>            wt = if t' < et then t' else et
>            (Window z e s) = w
>            t' = if z == LocalZero then t-st else t
>            nt = if z == LocalZero then st+wt else wt
>            st' = if z == LocalZero then 0.0 else st
>
>data PieceCont a b = Windows [FunctionWindow a b]
>instance Signal PieceCont where
>  mapSignal (Windows []) t = mapSignal NullRep t
>  mapSignal (Windows wl) t = (mapSignal s) (toPhysical t')
>      where (t', (Window z e s), wl') = getWindow 0.0 (fromPhysical t) wl
>  toSig = pieceRep
>  isNull (Windows []) t = True
>  isNull (Windows wl) t = (isNull s) (toPhysical t')
>      where (t', (Window z e s), wl') = getWindow 0.0 (fromPhysical t) wl
```

The operator for combining Windows is $|>$. A window that has no duration and is completely empty is called *nullWindow*.

```
>>nullWindow = Windows []
>(|>)::(Physical a, Physical b) => FunctionWindow a b->PieceCont a b->PieceCont a b
>(|>) w (Windows wl) = Windows (w:wl)
```

This operator, as well as the nullWindow, allows the specification of the PCS. The form of a PCS is, therefore, as follows:

```
[ cycleWindows ( ) <window> <<|> <window> >> |> nullWindow [ ] ]
```

The preceding PCS, cycleWindows, is used when the piecewise continuous function is intended to repeat infinitely for all time; otherwise, after the last window, the signal value returns to zero. Of course, the closing parenthesis is needed only if the signal specification is preceded by cycleWindows and the opening parenthesis.

```
>cycleWindows:: (Physical a, Physical b) => PieceCont a b -> PieceCont a b
>cycleWindows (Windows wl) = Windows (cycle wl)
```

A similar form is used when a set of windows is to be repeated N times:

`[ repNWindows ( ) <count> <window> <<|> <window> >>|> nullWindow [ ] ]`

where count represents the number of times the windows are to be replicated.

```
>repNWindows:: (Physical a, Physical b) => Int -> PieceCont a b -> PieceCont a b
>repNWindows i (Windows wl) = let
>  repN::(Physical a, Physical b)=> Int -> [FunctionWindow a b] -> [FunctionWindow
>  a b]
>  repN 0 _ = []
>  repN x ls = ls ++ (repN (x-1) ls)
>  in Windows (repN i wl)
```

The function `pieceRep` is used as a generator for normalized, or flattened, windows within PCSs and is used in preference to the constructor `PieceContRep` to allow for optimization of windows behavior.

```
>pieceRep:: (Physical a, Physical b) => PieceCont a b -> SignalRep a b
>pieceRep (Windows wl) = PieceContRep (Windows wl)

or an optimized version
pieceRep (Windows wl) = PieceContRep (Windows (flattenWindows 0.0 wl))
```

A.6.2 Sum

Another mechanism of making signals from other signals is to sum them together. This mechanism is identified by simply naming the signals to be summed, in the following form:

```
>sumSig:: (Physical a, Physical b, Signal s, Signal s') =>
>  (s a b) -> (s' a b) -> SignalRep a b
>sumSig f f' =
>  let s1 t = fromPhysical (mapSignal f t)
>      s2 t = fromPhysical (mapSignal f' t)
>  in FunctionRep (\t -> toPhysical ((s1 t) + (s2 t)))
```

An entire list of signals may be summed with a function of the following form:

```
>sumSigList:: (Physical a, Physical b, Signal s) => [ s a b ] -> SignalRep a b
>sumSigList ls = let zero = constant (toPhysical 0.0)
>  in foldr sumSig zero (map toSig ls)
```

A.6.3 Product

Two signals may be multiplied together via an operation of the following form:

```
>mulSig:: (Physical a, Physical b, Signal s, Signal s') => (s a b) -> (s' a b) ->
SignalRep a b
>mulSig f f' =
>  let f1 t = fromPhysical (mapSignal f t)
>      f2 t = fromPhysical (mapSignal f' t)
>  in FunctionRep (\t -> toPhysical ((f1 t) * (f2 t)))
```

A.7 Pure function transformations

Transformations take a signal and transform it, e.g., converting it from the time domain to the frequency domain. These transformations are pure in the sense that they are defined here in English.

A.7.1 Fourier transform

The Fourier transform converts time domain signals to frequency domain signals. It is, therefore, more restricted than other signal combination mechanisms. It takes a number of samples (which is always rounded up to the nearest power of two), the amount of time over which the signal will be sampled, and the signal to be converted. It has the following form:

```
>fourTrans:: (Physical a, Physical a', Physical b)=>
>             Int -> a -> SignalRep a b -> SignalRep a' b
>fourTrans sam t f =
>  let
>    waveform:: (Physical a, Physical b) => a -> [b] -> SignalRep a b
>    waveform samp ampls =
>      let stepSlope y y' = (/) ((fromPhysical y') - (fromPhysical y))
>                                (fromPhysical samp)
>      makeWin (v,v') = Window LocalZero (TimeEvent (fromPhysical samp))
>                                (linear (stepSlope v v') v)
>      points = ampls ++ (cycle [(toPhysical 0.0)])
>      in pieceRep (Windows (map makeWin (zip points (tail points))))
>    s = 2 ^ ((truncate (logBase 2 ((fromInteger (toInteger sam)) - 1.0))) + 1)
>    si = toPhysical (1.0 / (fromPhysical t))
>    trl = sampleCount (toPhysical 0.0) t s f
>    mc x = (fromPhysical x) :+ 0.0
>    til = map mc trl
>    fil = fft til
>    frl = map magnitude fil
>    in waveform si (map toPhysical frl)
```

where the functions `fft` and `fftinv` are imported from the module `Fourier`. The function `fft` provides the complex coefficients of a Fourier transform of a sample, and the function `fftinv` provides the complex coefficients of the original sample, as follows:

```
fft, fftinv:: [Complex Float] -> [Complex Float]
```

Note—The way by which SML defines the Fourier transform inherently utilizes a sampling technique. This technique is not rigorously identical to the Fourier transform, but tends towards the true transform as the number of samples is increased and when the time over which the samples are taken is the period of the signal.

A.8 Measuring, limiting, and sampling signals

The signals produced may be checked and their attributes measured. Two levels of checking are provided: a check upon the signal parameters and a check upon the signal itself. In addition to these checks, a number of measurements can be applied to the signals. Signals may also be sampled to return a list of values upon which functions may be defined.

Signal measurements are made on samplings of the signal over values of the independent variable. In other words, a window shall be specified and either an interval or the number of samples shall be given, just as with the sampling functions. The user may use the sampling functions given above as inputs into the measurement functions.

Measurements are performed on samplings of a signal. These samplings return a list of tuples consisting of signal values (extracted using the Haskell function `fst`) and independent values (extracted using the Haskell function `snd`).

A.8.1 Confining parameters to a limit

A parameter of any physical type may be limited to a particular range. In other words, if the given value is lower than the low value of the range, the parameter is made equal to that low value. If the parameter is greater than the high value of the range, the value is made equal to that parameter. No error is signaled.

The form of the limiting function is as follows:

limit <low_value> <high_value> <parameter_value>

```
>limit:: Physical a => a -> a -> a -> a
>limit low high val = let
>   rlow = fromPhysical low
>   rhigh = fromPhysical high
>   rval = fromPhysical val
>   in if rval <= rlow then low
>      else if rhigh <= rval then high
>      else val
```

A.8.2 Sampling signals

Signals are always sampled within a window, given as a low and a high value of the same type as the independent variable of the signal. Given this window and the signal, there are two ways to specify how the signals are to be sampled:

- a) The user can specify the number of points (i.e., signal value, independent variable) to be drawn from within the window:

```
>pointsCount:: (Physical a, Physical b, Signal s) => a -> a -> Int -> s a b -> [ (b,
a) ]
>pointsCount low high count sig = let
>   rlow = fromPhysical low
>   rhigh = fromPhysical high
>   toff = (rhigh - rlow) / ((fromIntegral count) - 1)
>   roff = toff - (toff / (2 * (fromIntegral count)))
>   creList low high off = if low <= high then
>                           (low : creList (low + off) high off)
>                           else []
>   appSig t = (mapSignal sig (toPhysical t), toPhysical t)
>   in map appSig (creList rlow rhigh roff)
```

- b) The user can also specify the number of samples (i.e., signal value) to be drawn from within the window:

```
>sampleCount:: (Physical a, Physical b, Signal s) => a -> a -> Int -> s a b -> [ b ]
>sampleCount low high count sig = map fst (pointsCount low high count sig)
```

Note—Substitute function snd for function fst to return the independent value.

A.9 Digital signals

A digital signal is a signal where information is represented in one of two values, which are sometimes called by names such as true and false, low and high, 1 and 0, etc. This representation, however, is complicated by the necessity to represent aspects of the behavior of digital signals within the electronic devices that operate on them.

A digital signal is an abstract representation of the values that are encountered in engineering design; these enumeration values are defined more precisely in A.9.1 through A.9.4.

Digital signals are unique in that their values do not take on physical values; rather, they take on enumeration values that represent physical values. A definition must be provided of what it means to operate on digital signals and what it means to convert a digital signal into an analog signal.

A.9.1 Defining Digital

```
>module Digital where
>import Physical
>import List
>import Pure
```

The definitions of the digital values that will be used are as follows:

- H corresponds to true or 1; it translates to 1 in a control signal (but see below).
- L corresponds to false or 0; it translates to 0 in a control signal (but see below).
- X represents the fact that the signal is in transition and, therefore, cannot be said to be at either value.
- Z represents the fact that the digital signal is providing very little current and will sink very little current. It represents a signal at a high impedance.

```
>data Digital = H | L | X | Z deriving (Eq, Show)
```

Four digital operations are defined for digital values, using an asterisk or the letter *d* to distinguish them from standard Boolean functions:

- **&*** is the digital function *and*.
- **|*** is the digital function *or*.
- **notd** is the digital function *not*.
- **=*** is the digital function *xor* (i.e., equal).

```
>class DigitalOps a where
>  (&*), (|*), (=*) :: a -> a -> a
>  notd :: a -> a

>instance DigitalOps Digital where
>  x &* y = case x of { L -> L; H -> y; X -> X; Z -> Z }
>  x |* y = case x of { L -> y; H -> H; X -> X; Z -> Z }
>  notd x = case x of { L -> H; H -> L; X -> X; Z -> Z }
>  x =* y = case x of
>    L -> if y == L then H else L
>    H -> if y == H then H else L
>    X -> X
>    Z -> Z
```

A.9.2 Defining DigitalSignal

A digital signal is specified as a time (which represents the transition period of the digital signal) and a list of digital values (i.e., Digital). A digital signal also supports the digital operation.

The definition for a digital signal is as follows:

```
>data DigitalSignal = Dig Time [ Digital ] deriving (Eq, Show)
>instance DigitalOps DigitalSignal where
>  d1@(Dig t1 l1) &* d2@(Dig t2 l2) =
>    let doAnd (x,y) = x &* y
>    t = if t1 == t2 then t1 else
>    error "Attempting to and two signals with different times"
>    in Dig t (map doAnd (zip l1 l2))
>  d1@(Dig t1 l1) |* d2@(Dig t2 l2) =
```

```
> let doOr (x,y) = x |* y
> t = if t1 == t2 then t1 else
>     error "Attempting to or two signals with different times"
> in Dig t (map doOr (zip l1 l2))
> notd (Dig t d1) = Dig t (map notd d1)
> d1@(Dig t1 l1) =* d2@(Dig t2 l2) =
>     let doXor (x,y) = x != y
>     t = if t1 == t2 then t1 else
>         error "Attempting to xor two signals with different times"
> in Dig t (map doXor (zip l1 l2))
```

Digital signals can also be generated using the function `str2dig` that allows digital strings containing the characters H, L, Z, X and white spaces to be converted into digital signals.

```
>str2dig:: Time -> String -> DigitalSignal
>str2dig t s = Dig t (map char2dig (filter (not.isSpace) s)) where
>     char2dig 'L' = L
>     char2dig 'H' = H
>     char2dig 'X' = X
>     char2dig 'Z' = Z
```

A.9.3 Conversion routines

Conversion routines convert from analog control signals to digital signals and vice versa. Digital signals can be combined with other digital signals, but need to be converted in order for them to be used with other SML signals. An analog control signal is an analog signal that uses the threshold low and high values and where the no signal value is used to detect tri-state Z values.

Two conversion routines are defined:

- Analog to digital (a2d)
- Digital to analog (d2a)

The conversion routines use physical threshold values to convert to and from low and high states.

The format of the conversion from analog signals to digital signals is as follows:

```
a2d <low_threshold> <high_threshold> <sample_rate> <analog_signal> <
```

Digital signals have distinct states, whereas their analog values are arbitrary, depending on the logic family thresholds:

- The Z digital state represents a high impedance signal with little or no current and is converted from the no signal, NullRep, SignalRep.
- The H digital state represents a logic high and is converted from values equal to or greater than the high threshold value.
- The L digital state represents a logic low and is converted from values equal to or less than the low threshold value.
- The X digital state represents all other values, i.e., values within the low-high thresholds.

Note—The description assumes that the high threshold is greater than the low threshold.

The signal has two thresholds; between the two thresholds, the X value is used. The representation NullRep signifies the presence of a Z; it is controlling in the sense that if there is no signal, then no current flows and it does not matter what the voltage level is. The sample rate simply gives the rate at which analog samples are taken and digital outputs produced.

```

>a2d::(Physical b, Signal s) => b -> b -> Time -> (s Time b) -> DigitalSignal
>a2d lowth highth sampRate s =
>  let lt = fromPhysical lowth
>      ht = fromPhysical highth
>      mt = (ht+lt)/2
>      dt = (ht-lt)/2
>      h = if dt<0 then L else H
>      sr = fromPhysical sampRate
>      val:: Float -> Digital
>      val x = let dv = fromPhysical (mapSignal s (toPhysical x)) - mt
>                in if isNull s (toPhysical x) then Z
>                   else if abs dv < abs dt then X
>                   else if dv > 0 then h else (notd h)
>      sl x = x : sl (x + sr)
>  in Dig sampRate (map val (sl 0.0))

```

The format of the conversion from digital to analog signal is as follows:

```
d2a <low_threshold> <high_threshold> <digital_signal>
```

The digital-to-analog conversion provides an analog signal as follows:

- A value of Z gives no signal (NullRep).
- A value of L produces a low threshold analog value.
- A value of H produces a high threshold analog value.
- A value of X produces a value that is the mean of the high and low thresholds.

In addition, when the signal goes to Z, the level of the voltage signal tends to “float” as necessary as follows:

```

>d2a::(Physical b) => b -> b -> DigitalSignal -> (SignalRep Time b)
>d2a lowth highth s@(Dig clock ds) =
>  let mth = toPhysical ((fromPhysical lowth + fromPhysical highth)/2)
>      win s = Window LocalZero (TimeEvent (fromPhysical clock)) s
>      makewin Z = win NullRep
>      makewin H = win (constant highth)
>      makewin L = win (constant lowth)
>      makewin X = win (constant mth)
>  in pieceRep (Windows (map makewin ds))

```

A.9.4 Patterns

Digital signals are often used in sets that represent related signals; for example, a set of 32 digital signals can represent a single, changing integer. Specifying sets of such digital signals is inconvenient using the form given above. In addition, such sets of digital signals usually share a common clock so that repeated specification of the time parameter is redundant.

A pattern is a convenient mechanism for specifying a number of parallel digital signals. It specifies the clock time once and then gives the digital values as a number of strings. An example of a parallel digital string utilizing white space is as follows:

```

Pattern (USEC 1) ["HLHL LLLL",
                  "LHLH HHHH",
                  "HHHH LLLL"]

```

The definition for a pattern is as follows:

```
>data Pattern = Pattern Time [ String ] deriving (Eq, Show)
```

Given a pattern, it may be converted into a list of digital signals:

```

>pat2diglist:: Pattern -> [ DigitalSignal ]
>pat2diglist (Pattern t ds) = map (str2dig t) (transpose ds)

```

A.10 Basic component SML

The BSC SML is defined in the module BSC

```
>module BSC where
```

making use of the previously defined types Physical, Pure, and Digital (see A.3, A.5 through A.7, and A.9, respectively)

```
>import Physical
>import Pure
>import Digital
```

by the following SML definitions: Source, Conditioner, EventFunction, Sensor, Digital, and Connection.

A.10.1 Source ::SignalFunction

A.10.1.1 NonPeriodic ::Source

A.10.1.1.1 Constant<type: Voltage|| Current|| Power> ::NonPeriodic

```
> (\amplitude -> constant amplitude)
```

A.10.1.1.2 Step<type: Voltage|| Current|| Power> ::NonPeriodic

```
> (\startTime amplitude -> --Step startTime amplitude
>   let st = fromPhysical startTime
>   zero = constant (toPhysical 0.0)
>   lvl = constant amplitude
>   wins = Window LocalZero (TimeEvent st) zero |>
>         Window LocalZero (TimeEvent inf) lvl |>
>         nullWindow
>   in pieceRep wins
> )
```

A.10.1.1.3 SingleTrapezoid<type: Voltage|| Current|| Power> ::NonPeriodic

```
> (\amplitude startTime riseTime pulseWidth fallTime ->let
>   startTime' = fromPhysical startTime
>   riseTime' = fromPhysical riseTime
>   pulseWidth' = fromPhysical pulseWidth
>   fallTime' = fromPhysical fallTime
>   wins = Window LocalZero (TimeEvent startTime') (constant (toPhysical 0)) |>
>         Window LocalZero (TimeEvent riseTime') (linear ((fromPhysical amplitude)/
>   riseTime') (toPhysical 0)) |>
>         Window LocalZero (TimeEvent pulseWidth') (constant amplitude) |>
>         Window LocalZero (TimeEvent fallTime') (linear (-(fromPhysical amplitude)/
>   fallTime') amplitude) |>
>   nullWindow
>   in pieceRep wins
> )
```

A.10.1.1.4 Noise<type: Voltage|| Current|| Power> ::NonPeriodic

```
> ((\freq amplitude seed -> --Noise {frequency=freq, amplitude=amplitude, seed=seed}
>   let pfive = constant (toPhysical (- 0.5))
>   amp = constant (toPhysical (2.0 * (fromPhysical amplitude)))
>   per = toPhysical ( 1.0 / (fromPhysical freq))
>   in mulSig amp (sumSig pfive (random seed per))
> )::(Physical a, Physical b)=>(Frequency -> b -> Integer -> (SignalRep a b)))
```

A.10.1.1.5 SingleRamp<type: Voltage|| Current|| Power> ::NonPeriodic

```

> (\amplitude startTime riseTime ->let {
>   ;wins = Window LocalZero (TimeEvent startTime') (constant (toPhysical 0)) |>
>   Window LocalZero (TimeEvent riseTime') (linear (amplitude'/
>   riseTime') (toPhysical 0)) |>
>   Window LocalZero (TimeEvent inf) (constant amplitude) |>
>   nullWindow
>   ;amplitude' = fromPhysical amplitude
>   ;startTime' = fromPhysical startTime
>   ;riseTime' = fromPhysical riseTime
>   } in pieceRep wins
> )

```

A.10.1.2 Periodic ::Source**A.10.1.2.1 Sinusoid<type: Voltage|| Current|| Power> ::Periodic**

```

> (\amplitude frequency phase -> --
>   Sine_wave {amplitude=amplitude, frequency=frequency, phase_angle=phase}
>   sine amplitude frequency phase
> )

```

A.10.1.2.2 Trapezoid<type: Voltage|| Current|| Power> ::Periodic

```

> (\amplitude period riseTime pulseWidth fallTime -> let
>   period' = fromPhysical period
>   riseTime' = fromPhysical riseTime
>   pulseWidth' = fromPhysical pulseWidth
>   fallTime' = fromPhysical fallTime
>   trapezoid = pieceRep $
>   Window LocalZero (TimeEvent riseTime') (linear ((fromPhysical amplitude)/
>   riseTime') (toPhysical 0)) |>
>   Window LocalZero (TimeEvent pulseWidth') (constant amplitude) |>
>   Window LocalZero (TimeEvent fallTime') (linear (-(fromPhysical amplitude)/
>   fallTime') amplitude) |>
>   nullWindow
>   in pieceRep $ cycleWindows $ Window LocalZero (TimeEvent period') trapezoid |>
>   nullWindow
> )

```

A.10.1.2.3 Ramp<type: Voltage|| Current|| Power> ::Periodic

```

> (\amplitude period riseTime -> --Ramp {period=period, rise_time=riseTime,
>   level=amplitude}
>   let per = fromPhysical period
>   rt = fromPhysical riseTime
>   v = fromPhysical amplitude
>   rsl = (v / rt)
>   fsl = (- v / (per - rt))
>   wins = Window LocalZero (TimeEvent rt) (linear rsl (toPhysical 0.0)) |>
>   Window LocalZero (TimeEvent (per - rt)) (linear fsl amplitude) |>
>   nullWindow
>   ramp = pieceRep wins
>   in pieceRep $ cycleWindows $ Window LocalZero (TimeEvent per) ramp |>
>   nullWindow
> )

```

A.10.1.2.4 Triangle<type: Voltage|| Current|| Power> ::Periodic

```

> (\amplitude period dutyCycle-> --Triangle {period=period, level=amplitude}
>   let per = fromPhysical period
>   v = fromPhysical amplitude
>   qper = per *dutyCycle / 2.0
>   sl = (v / qper)
>   nsl = 2.0* -v / (per - 2.0*qper)
>   wins = Window LocalZero (TimeEvent qper)
>   (linear sl (toPhysical 0.0)) |>
>   Window LocalZero (TimeEvent (per - 2.0*qper))

```

```
> (linear nsl amplitude) |>
> Window LocalZero (TimeEvent qper)
> (linear sl (toPhysical (- v))) |>
> nullWindow
> in pieceRep (cycleWindows wins)
> )
```

A.10.1.2.5 SquareWave<type: Voltage|| Current|| Power> ::Periodic

```
> (\amplitude period dutyCycle -> --Square {period=period, level=amplitude}
> let per = fromPhysical period
> lvl = fromPhysical amplitude
> trans = per * dutyCycle
> slvl = constant amplitude
> nslvl = constant (toPhysical (- lvl))
> wins = Window LocalZero (TimeEvent trans) slvl |>
> Window LocalZero (TimeEvent (per-trans)) nslvl |>
> nullWindow
> in pieceRep (cycleWindows wins)
> )
```

A.10.1.2.6 WaveformRamp<type: Voltage|| Current|| Power> ::Periodic

```
> (\amplitude samplintInt points ->
> let pts = map ((* (fromPhysical amplitude)) points in
> waveform samplintInt (map toPhysical pts)
> )
```

A.10.1.2.7 WaveformStep<type: Voltage|| Current|| Power> ::Periodic

```
> (\amplitude samplingInt points ->
> let pts = map ((* (fromPhysical amplitude)) points in
> FunctionRep (\t->cycle (map toPhysical pts) !! floor (fromPhysical t /
> (fromPhysical samplingInt)))
> )
```

A.10.2 Conditioner ::SignalFunction

A.10.2.1 Filter ::Conditioner

A.10.2.1.1 BandPass ::Filter

```
> (\interval samples centerFrequency frequencyBand signal -> let { t = toPhysical
> (interval);s=samples
> ;cf = centerFrequency; fb = frequencyBand
> ;x = truncate((cf-fb/2)*(fromPhysical t))
> ;y = truncate((cf+fb/2)*(fromPhysical t))
> ;z = cycle [0:+0]
> ;o = cycle [1:+0]
> ;mask = take (1+x) z ++ take (y-x+1) o ++ take (s-2*y-3) z ++ take (y-x+1) o
++ take (x) z
> --Remember only good for 0.5 sample freq
> --The top freq half is need to get good response. add padding to middle
> ;til = sampleCount (toPhysical 0.0) t s signal
> ;fil = fft $ map (\x->(fromPhysical x):+0.0) til
> ;frl = map realPart $ fftinv $ if x*2<s then mask`times` fil
> else take s z
> } in waveform (toPhysical ((fromPhysical t)/(fromInt s))) (map toPhysical
frl)
> )
```

A.10.2.1.2 LowPass ::Filter

```
> (\interval samples passband signal ->let { t = toPhysical (interval);s=samples
> ;x = truncate(passband*(fromPhysical t))
> ;z = cycle [0:+0]
> --Remember only good for 0.5 sample freq
```

```
>--The top freq half is need to get good response. add padding to middle
> ;til = sampleCount (toPhysical 0.0) t s signal
> ;fil = fft $ map (\x->(fromPhysical x):+0.0) til
> ;frl = map realPart $ fftinv $ if x*2<s then (take (1+x) fil) ++ (take (s-
x*2-1) z) ++ (drop (s-x) fil)
> else fil
> } in waveform (toPhysical ((fromPhysical t)/(fromInt s))) (map toPhysical
frl)
> )
```

A.10.2.1.3 HighPass ::Filter

```
> (\interval samples passband signal -> let { t = toPhysical interval;s=samples
> ;x = truncate(passband*(fromPhysical t))
> ;z = cycle [0:+0]
>--Remember only good for 0.5 sample freq
>--The top freq half is need to get good response. add padding to middle
> ;til = sampleCount (toPhysical 0.0) t s signal
> ;fil = fft $ map (\x->(fromPhysical x):+0.0) til
> ;frl = map realPart $ fftinv $ if x*2<s then (take (1+x) z) ++ drop (1+x)
(take (s-x) fil) ++ (take (x) z)
> else take s z
> } in waveform (toPhysical ((fromPhysical t)/(fromInt s))) (map toPhysical
frl)
> )
```

A.10.2.1.4 Notch ::Filter

```
> (\interval samples centerFrequency frequencyBand signal -> let { t = toPhysical
(interval);s=samples
> ;cf = centerFrequency; fb = frequencyBand
> ;x = truncate((cf-fb/2)*(fromPhysical t))
> ;y = truncate((cf+fb/2)*(fromPhysical t))
> ;z = cycle [0:+0]
> ;o = cycle [1:+0]
> ;mask = take (1+x) o ++ take (y-x+1) z ++ take (s-2*y-3) o ++ take (y-x+1) z
++ take (x) o
>--Remember only good for 0.5 sample freq
>--The top freq half is need to get good response. add padding to middle
> ;til = sampleCount (toPhysical 0.0) t s signal
> ;fil = fft $ map (\x->(fromPhysical x):+0.0) til
> ;frl = map realPart $ fftinv $ if x*2<s then mask`times` fil
> else take s z
> } in waveform (toPhysical ((fromPhysical t)/(fromInt s))) (map toPhysical
frl)
> )
```

A.10.2.2 Combiner ::Conditioner

A.10.2.2.1 Sum ::Combiner

```
> (\signals -> foldl1 sumSig ((constant (toPhysical 0)):signals))
```

A.10.2.2.2 Product ::Combiner

```
> (\signals -> foldl1 mulSig signals)
```

A.10.2.2.3 Diff ::Combiner

```
> (\signals -> foldl1 (\x y -
> sumSig x (negate y)) (signals++[constant (toPhysical 0)]))
```

A.10.2.3 Modulator ::Conditioner

A.10.2.3.1 FM ::Modulator

```

> (\freqDev carAmpP carFreq signal->
>   let {
>     ; phsfnc = mulSig (constant (toPhysical (freqDev*2*pi))) (intSig (16*carFreq) signa
1)
>     ; freqFn = constant (toPhysical carFreq)
>     ; fm = sineFunc (constant carAmpP) freqFn phsfnc
>   } in toSig fm
> )

```

A.10.2.3.2 AM ::Modulator

```

> (\modIndex carrier signal ->
>   let one = constant (toPhysical 1.0)
>   ; modsig = mulSig (constant (toPhysical modIndex)) signal
>   in mulSig carrier (sumSig one modsig)
> )

```

A.10.2.3.3 PM ::Modulator

```

> (\phaseDev carAmpP carFreq signal ->
>   let { phsfnc = mulSig (constant (toPhysical phaseDev)) signal
>         ; freqFn = constant (toPhysical carFreq)
>         ; pm = sineFunc (constant carAmpP) freqFn phsfnc
>       } in toSig pm
> )

```

A.10.2.4 Transformation ::Conditioner

A.10.2.4.1 SignalDelay ::Transformation

```

> (\delay rate acceleration signal -> let {
>   ; dt t = delay + rate*t + acceleration*t*t/2
>   ; t' t = max 0 (t-(dt t))
> --   ; delaySig s = FunctionRep (\t ->mapSignal s (toPhysical (t' (fromPhysical t))))
>
>   ; delaySig NullRep = NullRep
>   ; delaySig (FunctionRep fn) = FunctionRep (\t -
>fn (toPhysical (t' (fromPhysical t))))
>   ; delaySig (PieceContRep (Windows xs)) = PieceContRep (Windows ((Window GlobalZero
(TimeEvent (t'' 0)) NullRep):(delayWin 0 xs)))
>   ; delayWin gt ((Window GlobalZero (TimeEvent t) s):xs) = (Window GlobalZero (TimeEv
ent (t'' t)) (delaySig s):(delayWin t xs)
>   ; delayWin gt ((Window LocalZero (TimeEvent t) s):xs) = (Window LocalZero (TimeEvent
((t'' (gt+t))-(t'' gt))) (delaySig s):(delayWin (gt+t) xs)
>   ; sqe c 0 0 t = 0
>   ; sqe c b 0 t = -c/b
>   ; sqe c b a t = (-b + (sqrt (b*b-4*a*c)))/(2*a)
>   ; t'' t = max 0 (sqe (-delay-t) (1.0-rate) (-acceleration/2.0) t)
> } in delaySig signal)

```

A.10.2.4.2 Exponential ::Transformation

```

> mulSig (expc (1.0))

```

A.10.2.4.3 E ::Transformation

```

> (\f ->
>   let f1 t = fromPhysical (mapSignal f t)
>   in FunctionRep (\t -> toPhysical (exp (f1 t)))
> )

```

A.10.2.4.4 Negate ::Transformation

```

> (\signal -> mulSig signal (constant (toPhysical (-1))))

```

A.10.2.4.5 Inverse ::Transformation

```
> (\f ->
>   let f1 t = fromPhysical (mapSignal f t)
>   in FunctionRep (\t -> toPhysical (1.0 / (f1 t)))
> )
```

A.10.2.4.6 PulseTrain ::Transformation

```
> (\repetition pulses ->let
> {
>   rpt = let
>   {
>     pt ps= let
>     {
>       pulse (a, b, c) = let
>       {
>         zero = constant (toPhysical 0.0)
>         ;level = constant(toPhysical c)
>         ;wins = Window LocalZero (TimeEvent a) zero |>
>               Window LocalZero (TimeEvent (b)) level |>
>               nullWindow
>       }
>       in pieceRep (wins)
>     }
>     in sumSigList(map pulse ps)
>   }
>   ;width (a, b, _) = a + b
>   ;maxWidth ps = foldl (\v p->max v (width p)) 0 ps
>   ;win2 ps= Window LocalZero (TimeEvent (maxWidth ps)) (pt ps) |> nullWindow
>   ;repN 0 [] = nullWindow
>   ;repN 0 pts = cycleWindows (win2 pts)
>   ;repN rep pts = repNWindows rep (win2 pts)
> }
>   in pieceRep(repN repetition pulses)
> }
> in mulSig rpt )
```

A.10.2.4.7 Attenuator ::Transformation

```
> (\gain -> mulSig (constant (toPhysical gain)))
```

A.10.2.4.8 Load ::Transformation

```
> id
```

A.10.2.4.9 Limit<type: Voltage|| Current|| Power> ::Transformation

```
> (\lim sig -> FunctionRep (\t->limit (toPhysical (-
lim)) (toPhysical lim) (mapSignal sig t)))
```

A.10.2.4.10 FFT ::Transformation

```
> (\samples interval signal -> fourTrans samples interval (toSig (mulSig (constant
(toPhysical 2)) signal)))
```

A.10.3 EventFunction ::SignalFunction

A.10.3.1 EventSource ::EventFunction

A.10.3.1.1 Clock ::EventSource

```
> (\clock_rate->
>   let {
>     ;per = 0.5 / (fromPhysical clock_rate)
>     ;one = constant (toPhysical 1.0)
```

```
> ;zer = NullRep
> ;wins = Window LocalZero (TimeEvent per) one |>
> Window LocalZero (TimeEvent per) zer |>
>     nullWindow
> }in pieceRep (cycleWindows wins) )
```

A.10.3.1.2 TimedEvent ::EventSource

```
> (\delay repetition duration period -> let {
>   ;one = constant (toPhysical 1)
>   ;zero = NullRep
>   ;repNX 0 0 ls = cycleWindows ls
>   ;repNX delay 0 ls = Window LocalZero (TimeEvent delay) zero |> cycleWindows ls
>   ;repNX 0 x ls = repNWindows x ls
>   ;repNX delay x ls = Window LocalZero (TimeEvent delay) zero |> repNWindows x ls
>   } in pieceRep (repNX delay repetition
(Window LocalZero (TimeEvent duration) one |>
>   Window LocalZero (TimeEvent (period-duration)) zero |>
>   nullWindow))
> )
>
```

A.10.3.1.3 PulsedEvent ::EventSource

```
>(\repetition pulses -> let
>{
>   pt ps = let
>   {
>       pulse (a, b, c) = let
>       {
>           zero = NullRep
>           ;sig = constant (toPhysical 1)
>           ;funcwins = Window LocalZero (TimeEvent (a)) zero :
>               Window LocalZero (TimeEvent (b)) sig :
>               []
>       } in funcwins
>       ;xOr NullRep s = s
>       ;xOr s _ = s
>       ;sumEvnts a b = splice xOr a b 0.0
>   }
>   in Windows $ foldl1 (\a b->splice xOr a b 0.0) (map pulse ps)
>   ;repN 0 [] = nullWindow
>   ;repN 0 ps = cycleWindows ( pt ps)
>   ;repN rep ps = repNWindows rep ( pt ps)
>}
>in pieceRep(repN repetition pulses) )
```

A.10.3.2 EventConditioner ::EventFunction

A.10.3.2.1 EventedEvent :: EventConditioner

```
>(\events -> let
>{
>   ;one = constant (toPhysical 1)
>   ;ebe e d = let
>   {
>       ;enable = toSig e
>       ;disable = toSig d
>       ;wins = Window LocalZero (ActiveEvent enable) NullRep |>
>           Window LocalZero (ActiveEvent disable) one |> nullWindow
>   }
>   in pieceRep $ cycleWindows wins
>   ;sglEvt e = let
>   {
>       ;wins = Window LocalZero (ActiveEvent (toSig e)) NullRep |>
>           Window LocalZero (TimeEvent inf) one |> nullWindow
>   }
>   in pieceRep $ wins
>} in case events of
>   (e:[]) -> sglEvt e
```