

INTERNATIONAL STANDARD

NORME INTERNATIONALE



**OPC unified architecture –
Part 9: Alarms and conditions**

**Architecture unifiée OPC –
Partie 9: Alarmes et conditions**

Withdrawn



THIS PUBLICATION IS COPYRIGHT PROTECTED

Copyright © 2015 IEC, Geneva, Switzerland

All rights reserved. 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 either IEC or IEC's member National Committee in the country of the requester. If you have any questions about IEC copyright or have an enquiry about obtaining additional rights to this publication, please contact the address below or your local IEC member National Committee for further information.

Droits de reproduction réservés. Sauf indication contraire, aucune partie de cette publication ne peut être reproduite ni utilisée sous quelque forme que ce soit et par aucun procédé, électronique ou mécanique, y compris la photocopie et les microfilms, sans l'accord écrit de l'IEC ou du Comité national de l'IEC du pays du demandeur. Si vous avez des questions sur le copyright de l'IEC ou si vous désirez obtenir des droits supplémentaires sur cette publication, utilisez les coordonnées ci-après ou contactez le Comité national de l'IEC de votre pays de résidence.

IEC Central Office
3, rue de Varembe
CH-1211 Geneva 20
Switzerland

Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00
info@iec.ch
www.iec.ch

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.

IEC Catalogue - webstore.iec.ch/catalogue

The stand-alone application for consulting the entire bibliographical information on IEC International Standards, Technical Specifications, Technical Reports and other documents. Available for PC, Mac OS, Android Tablets and iPad.

IEC publications search - www.iec.ch/searchpub

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 It also gives information on projects, replaced and withdrawn publications.

IEC Just Published - webstore.iec.ch/justpublished

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

Electropedia - www.electropedia.org

The world's leading online dictionary of electronic and electrical terms containing more than 30 000 terms and definitions in English and French, with equivalent terms in 15 additional languages. Also known as the International Electrotechnical Vocabulary (IEV) online.

IEC Glossary - std.iec.ch/glossary

More than 60 000 electrotechnical terminology entries in English and French extracted from the Terms and Definitions clause of IEC publications issued since 2002. Some entries have been collected from earlier publications of IEC TC 37, 77, 86 and CISPR.

IEC Customer Service Centre - webstore.iec.ch/csc

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

A propos de l'IEC

La Commission Electrotechnique Internationale (IEC) est la première organisation mondiale qui élabore et publie des Normes internationales pour tout ce qui a trait à l'électricité, à l'électronique et aux technologies apparentées.

A propos des publications IEC

Le contenu technique des publications IEC est constamment revu. Veuillez vous assurer que vous possédez l'édition la plus récente, un corrigendum ou amendement peut avoir été publié.

Catalogue IEC - webstore.iec.ch/catalogue

Application autonome pour consulter tous les renseignements bibliographiques sur les Normes internationales, Spécifications techniques, Rapports techniques et autres documents de l'IEC. Disponible pour PC, Mac OS, tablettes Android et iPad.

Recherche de publications IEC - www.iec.ch/searchpub

La recherche avancée permet de trouver des publications IEC en utilisant différents critères (numéro de référence, texte, comité d'études,...). Elle donne aussi des informations sur les projets et les publications remplacées ou retirées.

IEC Just Published - webstore.iec.ch/justpublished

Restez informé sur les nouvelles publications IEC. Just Published détaille les nouvelles publications parues. Disponible en ligne et aussi une fois par mois par email.

Electropedia - www.electropedia.org

Le premier dictionnaire en ligne de termes électroniques et électriques. Il contient plus de 30 000 termes et définitions en anglais et en français, ainsi que les termes équivalents dans 15 langues additionnelles. Egalement appelé Vocabulaire Electrotechnique International (IEV) en ligne.

Glossaire IEC - std.iec.ch/glossary

Plus de 60 000 entrées terminologiques électrotechniques, en anglais et en français, extraites des articles Termes et Définitions des publications IEC parues depuis 2002. Plus certaines entrées antérieures extraites des publications des CE 37, 77, 86 et CISPR de l'IEC.

Service Clients - webstore.iec.ch/csc

Si vous désirez nous donner des commentaires sur cette publication ou si vous avez des questions contactez-nous: csc@iec.ch.

INTERNATIONAL STANDARD

NORME INTERNATIONALE



**OPC unified architecture –
Part 9: Alarms and conditions**

**Architecture unifiée OPC –
Partie 9: Alarmes et conditions**

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

COMMISSION
ELECTROTECHNIQUE
INTERNATIONALE

ICS 25.040.40; 35.100

ISBN 978-2-8322-2382-6

**Warning! Make sure that you obtained this publication from an authorized distributor.
Attention! Veuillez vous assurer que vous avez obtenu cette publication via un distributeur agréé.**

CONTENTS

FOREWORD.....	8
1 Scope.....	10
2 Normative references.....	10
3 Terms, definitions, and abbreviations	10
3.1 Terms and definitions	10
3.2 Abbreviations and symbols	12
3.3 Used data types	12
4 Concepts.....	12
4.1 General.....	12
4.2 Conditions.....	12
4.3 Acknowledgeable Conditions	14
4.4 Previous states of Conditions	15
4.5 Condition state synchronization	16
4.6 Severity, Quality, and Comment	16
4.7 Dialogs	17
4.8 Alarms	17
4.9 Multiple Active States	18
4.10 <i>Condition</i> Instances in the Address Space	19
4.11 Alarm and Condition Auditing	19
5 Model.....	19
5.1 General.....	19
5.2 Two-State State Machines.....	20
5.3 Condition Variables	21
5.4 Substate Reference Types	22
5.4.1 General.....	22
5.4.2 HasTrueSubState ReferenceType.....	22
5.4.3 HasFalseSubState ReferenceType	23
5.5 Condition Model.....	23
5.5.1 General.....	23
5.5.2 ConditionType	24
5.5.3 Condition and Branch Instances	27
5.5.4 Disable Method	27
5.5.5 Enable Method	28
5.5.6 AddComment Method	28
5.5.7 ConditionRefresh Method	29
5.6 Dialog Model.....	31
5.6.1 General.....	31
5.6.2 DialogConditionType	31
5.6.3 Respond Method	32
5.7 Acknowledgeable Condition Model	33
5.7.1 General.....	33
5.7.2 AcknowledgeableConditionType	33
5.7.3 Acknowledge Method	34
5.7.4 Confirm Method.....	35
5.8 Alarm Model.....	36
5.8.1 General.....	36

5.8.2	AlarmConditionType	37
5.8.3	ShelvedStateMachineType	39
5.8.4	LimitAlarmType	43
5.8.5	ExclusiveLimit Types	44
5.8.6	NonExclusiveLimitAlarmType	46
5.8.7	Level Alarm	48
5.8.8	Deviation Alarm	48
5.8.9	Rate of Change	49
5.8.10	Discrete Alarms	50
5.9	ConditionClasses	52
5.9.1	Overview	52
5.9.2	Base ConditionClassType	52
5.9.3	ProcessConditionClassType	53
5.9.4	MaintenanceConditionClassType	53
5.9.5	SystemConditionClassType	53
5.10	Audit Events	53
5.10.1	Overview	53
5.10.2	AuditConditionEventType	54
5.10.3	AuditConditionEnableEventType	55
5.10.4	AuditConditionCommentEventType	55
5.10.5	AuditConditionRespondEventType	55
5.10.6	AuditConditionAcknowledgeEventType	55
5.10.7	AuditConditionConfirmEventType	56
5.10.8	AuditConditionShelvingEventType	56
5.11	Condition Refresh Related Events	56
5.11.1	Overview	56
5.11.2	RefreshStartEventType	57
5.11.3	RefreshEndEventType	57
5.11.4	RefreshRequiredEventType	57
5.12	HasCondition Reference Type	58
5.13	Alarm and Condition Status Codes	58
5.14	Expected A&C Server Behaviours	59
5.14.1	General	59
5.14.2	Communication problems	59
5.14.3	Redundant A&C Servers	59
6	AddressSpace Organisation	60
6.1	General	60
6.2	Event Notifier and Source Hierarchy	60
6.3	Adding Conditions to the Hierarchy	61
6.4	Conditions in InstanceDeclarations	61
6.5	Conditions in a VariableType	62
Annex A	(informative) Recommended localized names	63
A.1	Recommended State Names for TwoState Variables	63
A.1.1	LocaleId “en”	63
A.1.2	LocaleId “de”	63
A.1.3	LocaleId “fr”	64
A.2	Recommended Dialog Response Options	64
Annex B	(informative) Examples	65
B.1	Examples for Event sequences from Condition instances	65

B.1.1	Overview.....	65
B.1.2	Server Maintains Current State Only.....	65
B.1.3	Server Maintains Previous States	65
B.2	Address Space Examples	67
Annex C (informative)	Mapping to EEMUA.....	71
Annex D (informative)	Mapping from OPC A&E to OPC UA A&C	72
D.1	Overview.....	72
D.2	Alarms and Events COM UA Wrapper.....	72
D.2.1	Event Areas	72
D.2.2	Event Sources.....	73
D.2.3	Event Categories.....	73
D.2.4	Event Attributes	74
D.2.5	Event Subscriptions.....	74
D.2.6	Condition Instances.....	76
D.2.7	Condition Refresh	76
D.3	Alarms and Events COM UA Proxy	77
D.3.1	General.....	77
D.3.2	Server Status Mapping	77
D.3.3	Event Type Mapping.....	77
D.3.4	Event Category Mapping	78
D.3.5	Event Category Attribute Mapping	79
D.3.6	Event Condition Mapping.....	82
D.3.7	Browse Mapping.....	82
D.3.8	Qualified Names.....	83
D.3.9	Subscription Filters	84
Bibliography		86
Figure 1 – Base Condition State Model.....		13
Figure 2 – AcknowledgeableConditions State Model		14
Figure 3 – Acknowledge State Model.....		15
Figure 4 – Confirmed Acknowledge State Model.....		15
Figure 5 – Alarm State Machine Model		17
Figure 6 – Multiple Active States Example		18
Figure 7 – ConditionType Hierarchy		20
Figure 8 – Condition Model		24
Figure 9 – DialogConditionType Overview		31
Figure 10 – AcknowledgeableConditionType Overview		33
Figure 11 – AlarmConditionType Hierarchy Model		37
Figure 12 – Alarm Model.....		37
Figure 13 – Shelve state transitions		39
Figure 14 – Shelved State Machine Model.....		40
Figure 15 – LimitAlarmType		43
Figure 16 – ExclusiveLimitStateMachine.....		44
Figure 17 – ExclusiveLimitAlarmType.....		46
Figure 18 – NonExclusiveLimitAlarmType.....		47
Figure 19 – DiscreteAlarmType Hierarchy		

Figure 20 – ConditionClass Type Hierarchy	52
Figure 21 – AuditEvent Hierarchy	54
Figure 22 – Refresh Related Event Hierarchy	57
Figure 23 – Typical Event Hierarchy	60
Figure 24 – Use of HasCondition in an Event Hierarchy	61
Figure 25 – Use of HasCondition in an InstanceDeclaration	62
Figure 26 – Use of HasCondition in a VariableType	62
Figure B.1 – Single State Example	65
Figure B.2 – Previous State Example	66
Figure B.3 – HasCondition used with Condition instances	68
Figure B.4 – HasCondition reference to a Condition Type	69
Figure B.5 – HasCondition used with an instance declaration	70
Figure D.1 – The Type Model of a Wrapped COM AE Server	74
Figure D.2 – Mapping UA Event Types to COM A&E Event Types	78
Figure D.3 – Example Mapping of UA Event Types to COM A&E Categories	79
Figure D.4 – Example Mapping of UA Event Types to A&E Categories with Attributes	82
Table 1 – Parameter Types defined in IEC 62541-3	12
Table 2 – Parameter Types defined in IEC 62541-4	12
Table 3 – TwoStateVariableType Definition	21
Table 4 – ConditionVariableType Definition	22
Table 5 – HasTrueSubState ReferenceType	22
Table 6 – HasFalseSubState ReferenceType	23
Table 7 – ConditionType Definition	25
Table 8 – Simple Attribute Operand	27
Table 9 – Disable Result Codes	28
Table 10 – Disable Method AddressSpace Definition	28
Table 11 – Enable Result Codes	28
Table 12 – Enable Method AddressSpace Definition	28
Table 13 – AddComment Arguments	29
Table 14 – AddComment result Codes	29
Table 15 – AddComment Method AddressSpace Definition	29
Table 16 – ConditionRefresh Parameters	30
Table 17 – ConditionRefresh ReturnCodes	30
Table 18 – ConditionRefresh Method AddressSpace Definition	31
Table 19 – DialogConditionType Definition	31
Table 20 – Repond Parameters	32
Table 21 – Respond ResultCodes	33
Table 22 – Respond Method AddressSpace Definition	33
Table 23 – AcknowledgeableConditionType Definition	34
Table 24 – Acknowledge Parameters	34
Table 25 – Acknowledge result codes	35
Table 26 – Acknowledge Method AddressSpace Definition	35

Table 27 – Confirm Method Parameters	35
Table 28 – Confirm Result Codes	36
Table 29 – Confirm Method AddressSpace Definition	36
Table 30 – AlarmConditionType Definition	38
Table 31 –ShelvedStateMachine Definition	40
Table 32 – ShelvedStateMachine Transitions	41
Table 33 – Unshelve Result Codes	41
Table 34 – Unshelve Method AddressSpace Definition	41
Table 35 – TimedShelve Parameters	42
Table 36 – TimedShelve Result Codes	42
Table 37 – TimedShelve Method AddressSpace Definition	42
Table 38 – OneShotShelve Result Codes	42
Table 39 – OneShotShelve Method AddressSpace Definition	43
Table 40 – LimitAlarmType Definition	43
Table 41 – ExclusiveLimitStateMachineType Definition	44
Table 42 – ExclusiveLimitStateMachineType Transitions	45
Table 43 – ExclusiveLimitAlarmType Definition	46
Table 44 – NonExclusiveLimitAlarmType Definition	47
Table 45 – NonExclusiveLevelAlarmType Definition	48
Table 46 – ExclusiveLevelAlarmType Definition	48
Table 47 – NonExclusiveDeviationAlarmType Definition	49
Table 48 – ExclusiveDeviationAlarmType Definition	49
Table 49 – NonExclusiveRateOfChangeAlarmType Definition	50
Table 50 – ExclusiveRateOfChangeAlarmType Definition	50
Table 51 – DiscreteAlarmType Definition	51
Table 52 – OffNormalAlarmType Definition	51
Table 53 – SystemOffNormalAlarmType Definition	51
Table 54 – TripAlarmType Definition	52
Table 55 – BaseConditionClassType Definition	52
Table 56 – ProcessConditionClassType Definition	53
Table 57 – MaintenanceConditionClassType Definition	53
Table 58 – SystemConditionClassType Definition	53
Table 59 – AuditConditionEventType Definition	54
Table 60 – AuditConditionEnableEventType Definition	55
Table 61 – AuditConditionCommentEventType Definition	55
Table 62 – AuditConditionRespondEventType Definition	55
Table 63 – AuditConditionAcknowledgeEventType Definition	56
Table 64 – AuditConditionConfirmEventType Definition	56
Table 65 – AuditConditionShelvingEventType Definition	56
Table 66 – RefreshStartEventType Definition	57
Table 67 – RefreshEndEventType Definition	57
Table 68 – RefreshRequiredEventType Definition	58
Table 69 – HasCondition ReferenceType	58

Table 70 – Alarm and Condition Result Codes	59
Table A.1 – Recommended state names for LocaleId “en”	63
Table A.2 – Recommended display names for LocaleId “en”	63
Table A.3 – Recommended state names for LocaleId “de”	63
Table A.4 – Recommended display names for LocaleId “de”	64
Table A.5 – Recommended state names for LocaleId “fr”	64
Table A.6 – Recommended display names for LocaleId “fr”	64
Table A.7 – Recommended Dialog Response Options.....	64
Table B.1 – Example of a Condition that only keeps the latest state	65
Table B.2 – Example of a <i>Condition</i> that maintains previous states via branches.....	67
Table C.1 – EEMUA Terms	71
Table D.1 – Mapping from Standard Event Categories to OPC UA Event Types	73
Table D.2 – Mapping from ONEVENTSTRUCT fields to UA BaseEventType Variables.....	75
Table D.3 – Mapping from ONEVENTSTRUCT fields to UA AuditEventType Variables.....	75
Table D.4 – Mapping from ONEVENTSTRUCT fields to UA AlarmType Variables	76
Table D.5 – Event Category Attribute Mapping Table.....	80

Withdrawn

INTERNATIONAL ELECTROTECHNICAL COMMISSION

OPC UNIFIED ARCHITECTURE –

Part 9: Alarms and conditions

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 itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
- 6) All users should ensure that they have the latest edition of this publication.
- 7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
- 8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
- 9) Attention is drawn to the possibility that 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 62541-9 has been prepared by subcommittee 65E: Devices and integration in enterprise systems, of IEC technical committee 65: Industrial-process measurement, control and automation.

This second edition cancels and replaces the first edition published in 2012. This edition constitutes a technical revision.

This edition includes the following significant technical changes with respect to the previous edition:

- a) added section to describe expect behaviour for A&C servers and the associated information model in the case of redundancy or communication faults, see 5.14 for additional details.[ref 698 & 967];
- b) changed the DialogConditionType to be not abstract since it is expect that instance of this type will exist in the system, see Table 19 for additional details [ref 1622];

- c) updated ConditionRefresh Method to allow the use of the well know NodeIds associated with the types for the MethodId and ConditionId instead of requiring the call to use only the MethodId and ConditionId that is part of an instance. Without this change, servers that do not expose instance may have problems with ConditionRefresh, see 5.5.7 for additional details [ref 2091];
- d) Fixed ExclusiveLimitStateMachineType and ShelvedStateMachineType to be sub-types of FinitStateMachineType not StateMachineType. See 5.8.3 and 5.8.5.2 for additional details [ref 2091].

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

CDV	Report on voting
65E/382/CDV	65E/408/RVC

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

This publication has been drafted in accordance with the ISO/IEC Directives, Part 2.

A list of all parts of the IEC 62541 series, published under the general title *OPC Unified Architecture*, can be found on the IEC website.

The committee has decided that the contents of this publication will remain unchanged until the stability 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.

IMPORTANT – The 'colour inside' logo on the cover page of this publication indicates that it contains colours which are considered to be useful for the correct understanding of its contents. Users should therefore print this document using a colour printer.

OPC UNIFIED ARCHITECTURE –

Part 9: Alarms and conditions

1 Scope

This part of IEC 62541 specifies the representation of *Alarms* and *Conditions* in the OPC Unified Architecture. Included is the *Information Model* representation of *Alarms* and *Conditions* in the OPC UA address space.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC TR 62541-1, *OPC Unified Architecture – Part 1: Overview and Concepts*

IEC 62541-3, *OPC Unified Architecture – Part 3: Address Space Model*

IEC 62541-4, *OPC Unified Architecture – Part 4: Services*

IEC 62541-5, *OPC Unified Architecture – Part 5: Information Model*

IEC 62541-6, *OPC Unified Architecture – Part 6: Mappings*

IEC 62541-8, *OPC Unified Architecture – Part 8: Data Access*

EEMUA: 2nd Edition EEMUA 191 – *Alarm System – A guide to design, management and procurement* (Appendixes 6, 7, 8, 9), available at <http://www.eemua.co.uk/>

3 Terms, definitions, and abbreviations

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in IEC TR 62541-1, IEC 62541-3, IEC 62541-4, and IEC 62541-5 as well as the following apply.

3.1.1

acknowledge

operator action that indicates recognition of a new Alarm

Note 1 to entry: This definition is copied from EEMUA. The term “Accept” is another common term used to describe *Acknowledge*. They can be used interchangeably. This standard will use *Acknowledge*.

3.1.2

active

state for an Alarm that indicates that the situation the Alarm is representing currently exists

Note 1 to entry: Other common terms defined by EEMUA are “Standing” for an *Active Alarm* and “Cleared” when the *Condition* has returned to normal and is no longer *Active*.

3.1.3

ConditionClass

Condition grouping that indicates in which domain or for what purpose a certain *Condition* is used

Note 1 to entry: Some top-level *ConditionClasses* are defined in this specification. Vendors or organisations may derive more concrete classes or define different top-level classes.

3.1.4

ConditionBranch

specific state of a *Condition*

Note 1 to entry: The *Server* can maintain *ConditionBranches* for the current state as well as for previous states.

3.1.5

ConditionSource

element which a specific *Condition* is based upon or related to

Note 1 to entry: Typically, it will be a *Variable* representing a process tag (e.g. FIC101) or an *Object* representing a device or subsystem.

In *Events* generated for *Conditions*, the *SourceNode Property* (inherited from the *BaseEventType*) will contain the *NodeId* of the *ConditionSource*.

3.1.6

confirm

operator action informing the *Server* that a corrective action has been taken to address the cause of the *Alarm*

3.1.7

disable

system is configured such that the *Alarm* will not be generated even though the base *Alarm Condition* is present

Note 1 to entry: This definition is copied from EEMUA and is further described in EEMUA.

3.1.8

operator

special user who is assigned to monitor and control a portion of a process

Note 1 to entry: "A Member of the operations team who is assigned to monitor and control a portion of the process and is working at the control system's Console" as defined in EEMUA. In this standard an Operator is a special user. All descriptions that apply to general users also apply to Operators.

3.1.9

refresh

act of providing an update to an *Event Subscription* that provides all *Alarms* which are considered to be *Retained*

Note 1 to entry: This concept is further described in EEMUA.

3.1.10

retain

alarm in a state that is interesting for a *Client* wishing to synchronize its state of *Conditions* with the *Server's* state.

3.1.11

shelving

facility where the *Operator* is able to temporarily prevent an *Alarm* from being displayed to the *Operator* when it is causing the *Operator* a nuisance

Note 1 to entry: A Shelved *Alarm* will be removed from the list and will not re-annunciate until un-shelved. This definition is copied from EEMUA..

3.1.12

suppress

act of determining whether an *Alarm* should not occur

Note 1 to entry: “An *Alarm* is suppressed when logical criteria are applied to determine that the *Alarm* should not occur, even though the base *Alarm Condition* (e.g. *Alarm* setting exceeded) is present” as defined in EEMUA.

3.2 Abbreviations and symbols

A&E	Alarm&Event (as used for OPC COM)
COM	(Microsoft Windows) Component Object Model
DA	Data Access
UA	Unified Architecture

3.3 Used data types

The following tables describe the data types that are used throughout this standard. These types are separated into two tables. Base data types defined in IEC 62541-3 are given in Table 1. The base types and data types defined in IEC 62541-4 are given in Table 2.

Table 1 – Parameter Types defined in IEC 62541-3

Parameter Type
Argument
BaseDataType
NodeId
LocalizedText
Boolean
ByteString
Double
Duration
String
UInt16
Int32
UtcTime

Table 2 – Parameter Types defined in IEC 62541-4

Parameter Type
IntegerId
StatusCode

4 Concepts

4.1 General

This standard defines an *Information Model* for *Conditions*, *Dialog Conditions*, and *Alarms* including acknowledgement capabilities. It is built upon and extends base Event handling which is defined in IEC 62541-3, IEC 62541-4 and IEC 62541-5. This *Information Model* can also be extended to support the additional needs of specific domains. The details of what aspects of the Information Model are supported are provided via Profiles (see IEC 62541-7). It is expected that systems will provide historical Events and Conditions via the standard Historical Access framework (see IEC 62541-11).

4.2 Conditions

Conditions are used to represent the state of a system or one of its components. Some common examples are:

- a temperature exceeding a configured limit;

- a device needing maintenance;
- a batch process that requires a user to confirm some step in the process before proceeding.

Each *Condition* instance is of a specific *ConditionType*. The *ConditionType* and derived types are sub-types of the *BaseEventType* (see IEC 62541-3 and IEC 62541-5). This part defines types that are common across many industries. It is expected that vendors or other standardisation groups will define additional *ConditionTypes* deriving from the common base types defined in this part. The *ConditionTypes* supported by a *Server* are exposed in the *AddressSpace* of the *Server*.

Condition instances are specific implementations of a *ConditionType*. It is up to the *Server* whether such instances are also exposed in the *Server's AddressSpace*. Subclause 4.10 provides additional background about *Condition* instances. *Condition* instances shall have a unique identifier to differentiate them from other instances. This is independent of whether they are exposed in the *AddressSpace*.

As mentioned above, *Conditions* represent the state of a system or one of its components. In certain cases, however, previous states that still need attention also have to be maintained. *ConditionBranches* are introduced to deal with this requirement and distinguish current state and previous states. Each *ConditionBranch* has a *BranchId* that differentiates it from other branches of the same *Condition* instance. The *ConditionBranch* which represents the current state of the *Condition* (the trunk) has a Null *BranchId*. *Servers* can generate separate *Event Notifications* for each branch. When the state represented by a *ConditionBranch* does not need further attention, a final *Event Notification* for this branch will have the *Retain Property* set to False. Subclause 4.4 provides more information and use cases. Maintaining previous states and therefore also the support of multiple branches is optional for *Servers*.

Conceptually, the lifetime of the *Condition* instance is independent of its state. However, *Servers* may provide access to *Condition* instances only while *ConditionBranches* exist.

The base *Condition* state model is illustrated in Figure 1. It is extended by the various *Condition* subtypes defined in this standard and may be further extended by vendors or other standardisation groups. The primary states of a *Condition* are disabled and enabled. The *Disabled* state is intended to allow *Conditions* to be turned off at the *Server* or below the *Server* (in a device or some underlying system). The *Enabled* state is normally extended with the addition of sub-states.

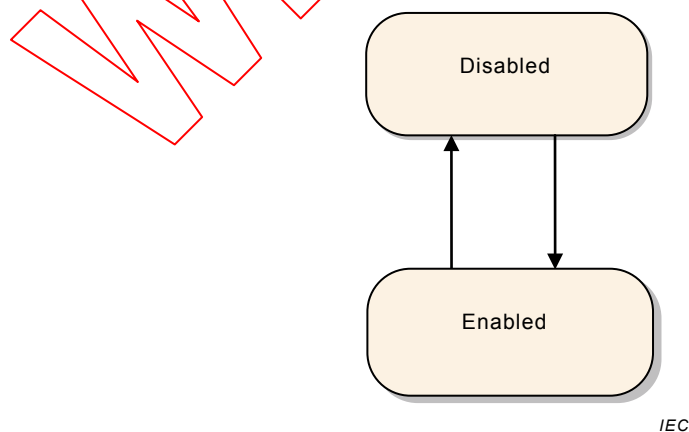


Figure 1 – Base Condition State Model

A transition into the *Disabled* state results in a *Condition Event* however no subsequent *Event Notifications* are generated until the *Condition* returns to the *Enabled* state.

When a *Condition* enters the Enabled state, that transition and all subsequent transitions result in *Condition Events* being generated by the *Server*.

If *Auditing* is supported by a *Server*, the following *Auditing* related action shall be performed. The *Server* will generate *AuditEvents* for *Enable* and *Disable* operations (either through a *Method* call or some *Server* / vendor – specific means), rather than generating an *AuditEvent Notification* for each *Condition* instance being enabled or disabled. For more information, see the definition of *AuditConditionEnableEventType* in 5.10.2. *AuditEvents* are also generated for any other *Operator* action that results in changes to the *Conditions*.

4.3 Acknowledgeable Conditions

AcknowledgeableConditions are sub-types of the base *ConditionType*. *AcknowledgeableConditions* expose states to indicate whether a *Condition* has to be acknowledged or confirmed.

An *AckedState* and a *ConfirmedState* extend the *EnabledState* defined by the *Condition*. The state model is illustrated in Figure 2. The enabled state is extended by adding the *AckedState* and (optionally) the *ConfirmedState*.

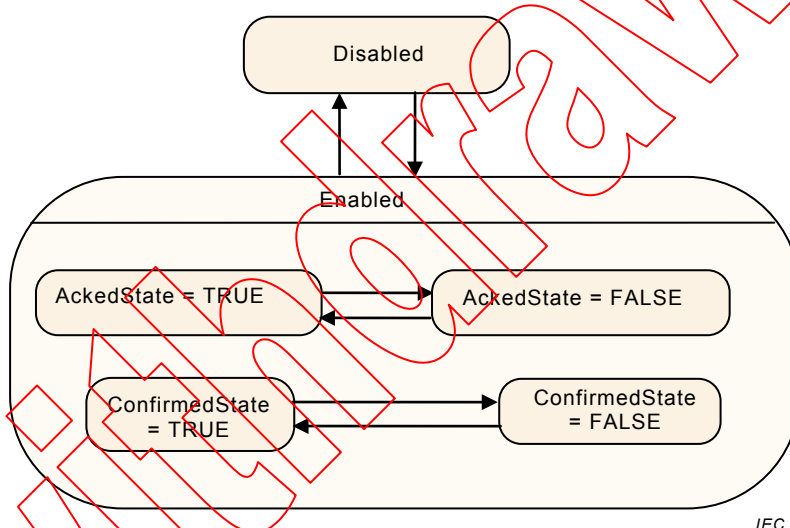


Figure 2 – AcknowledgeableConditions State Model

Acknowledgment of the transition may come from the *Client* or may be due to some logic internal to the *Server*. For example, acknowledgment of a related *Condition* may result in this *Condition* becoming acknowledged, or the *Condition* may be set up to automatically acknowledge itself when the acknowledgeable situation disappears.

Two *Acknowledge* state models are supported by this standard. Either of these state models can be extended to support more complex acknowledgement situations.

The basic *Acknowledge* state model is illustrated in Figure 3. This model defines an *AckedState*. The specific state changes that result in a change to the state depend on a *Server*'s implementation. For example, in typical *Alarm* models the change is limited to a transition to the *Active* state or transitions within the *Active* state. More complex models however can also allow for changes to the *AckedState* when the *Condition* transitions to an inactive state.

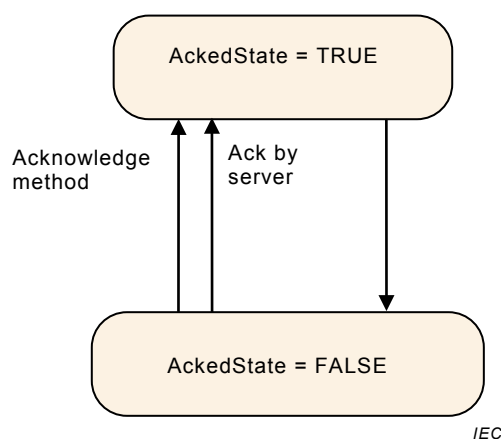


Figure 3 – Acknowledge State Model

A more complex state model which adds a confirmation to the basic *Acknowledge* is illustrated in Figure 4. The *Confirmed Acknowledge* model is typically used to differentiate between acknowledging the presence of a *Condition* and having done something to address the *Condition*. For example an *Operator* receiving a motor high temperature *Notification* calls the *Acknowledge Method* to inform the *Server* that the high temperature has been observed. The *Operator* then takes some action such as lowering the load on the motor in order to reduce the temperature. The *Operator* then calls the *Confirm Method* to inform the *Server* that a corrective action has been taken.

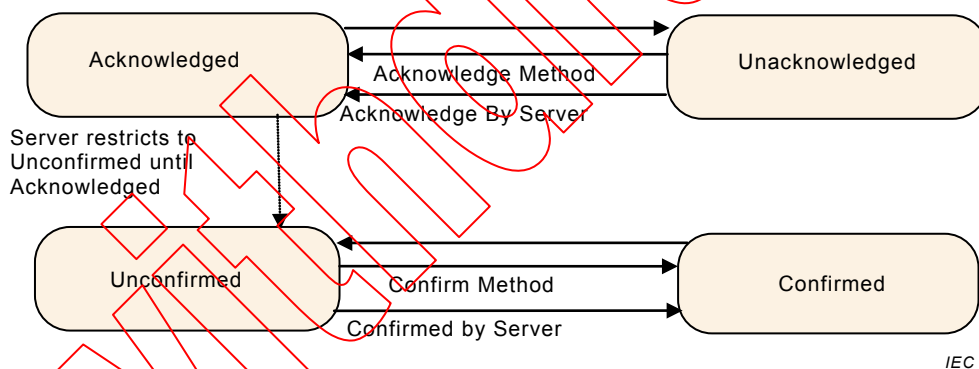


Figure 4 – Confirmed Acknowledge State Model

4.4 Previous states of Conditions

Some systems require that previous states of a *Condition* are preserved for some time. A common use case is the acknowledgement process. In certain environments it is required to acknowledge both the transition into *Active* state and the transition into an inactive state. Systems with strict safety rules sometimes require that every transition into *Active* state has to be acknowledged. In situations where state changes occur in short succession there can be multiple unacknowledged states and the *Server* has to maintain *ConditionBranches* for all previous unacknowledged states. These branches will be deleted after they have been acknowledged or if they reached their final state.

Multiple ConditionBranches can also be used for other use cases where snapshots of previous states of a *Condition* require additional actions.

4.5 Condition state synchronization

When a *Client* subscribes for *Events*, the *Notification* of transitions will begin at the time of the *Subscription*. The currently existing state will not be reported. This means for example that *Clients* are not informed of currently *Active Alarms* until a new state change occurs.

Clients can obtain the current state of all *Condition* instances that are in an interesting state, by requesting a *Refresh* for a *Subscription*. It should be noted that *Refresh* is not a general replay capability since the *Server* is not required to maintain an *Event* history.

Clients request a *Refresh* by calling the *ConditionRefresh Method*. The *Server* will respond with a *RefreshStartEvent*. This *Event* is followed by the *Retained Conditions*. The *Server* may also send new *Event Notifications* interspersed with the *Refresh* related *Event Notifications*. After the *Server* is done with the *Refresh*, a *RefreshEndEvent* is issued marking the completion of the *Refresh*. *Clients* shall check for multiple *Event Notifications* for a *ConditionBranch* to avoid overwriting a new state delivered together with an older state from the *Refresh* process. If a *ConditionBranch* exists, then the current *Condition* shall be reported. This is true even if the only interesting item regarding the *Condition* is that *ConditionBranches* exist. This allows a *Client* to accurately represent the current *Condition* state.

A *Client* that wishes to display the current status of *Alarms* and *Conditions* (known as a “current *Alarm* display”) would use the following logic to process *Refresh Event Notifications*. The *Client* flags all *Retained Conditions* as suspect on reception of the *Event* of the *RefreshStartEvent*. The *Client* adds any new *Events* that are received during the *Refresh* without flagging them as suspect. The *Client* also removes the suspect flag from any *Retained Conditions* that are returned as part of the *Refresh*. When the *Client* receives a *RefreshEndEvent*, the *Client* removes any remaining suspect *Events*, since they no longer apply.

The following items should be noted with regard to *ConditionRefresh*:

- As described in 4.4 some systems require that previous states of a *Condition* are preserved for some time. Some *Servers* – in particular if they require acknowledgement of previous states – will maintain separate *ConditionBranches* for prior states that still need attention.
ConditionRefresh shall issue *Event Notifications* for all interesting states (current and previous) of a *Condition* instance and *Clients* can therefore receive more than one *Event* for a *Condition* instance with different *BranchIds*.
- Under some circumstances a *Server* may not be capable of ensuring the *Client* is fully in sync with the current state of *Condition* instances. For example if the underlying system represented by the *Server* is reset or communications are lost for some period of time the *Server* may need to resynchronize itself with the underlying system. In these cases the *Server* shall send an *Event* of the *RefreshRequiredEventType* to advise the *Client* that a *Refresh* may be necessary. A *Client* receiving this special *Event* should initiate a *ConditionRefresh* as noted in this clause.
- To ensure a *Client* is always informed, the three special *EventTypes* (*RefreshEndEventType*, *RefreshStartEventType* and *RefreshRequiredEventType*) ignore the *Event* content filtering associated with a *Subscription* and will always be delivered to the *Client*.

4.6 Severity, Quality, and Comment

Comment, Severity and Quality are important elements of *Conditions* and any change to them will cause *Event Notifications*.

The Severity of a *Condition* is inherited from the base *Event* model defined in IEC 62541-5. It indicates the urgency of the *Condition* and is also commonly called ‘priority’, especially in relation to *Alarms* in the *ProcessConditionClass*.

A Comment is a user generated string that is to be associated with a certain state of a *Condition*.

Quality refers to the quality of the data value(s) upon which this *Condition* is based. Since a *Condition* is usually based on one or more *Variables*, the *Condition* inherits the quality of these *Variables*. E.g., if the process value is “Uncertain”, the “LevelAlarm” *Condition* is also questionable. If more than one variable is represented by a given condition or if the condition is from an underlining system and no direct mapping to a variable is available, it is up to the application to determine what quality is displayed as part of the condition.

4.7 Dialogs

Dialogs are *ConditionTypes* used by a *Server* to request user input. They are typically used when a *Server* has entered some state that requires intervention by a *Client*. For example a *Server* monitoring a paper machine indicates that a roll of paper has been wound and is ready for inspection. The *Server* would activate a Dialog *Condition* indicating to the user that an inspection is required. Once the inspection has taken place the user responds by informing the *Server* of an accepted or unaccepted inspection allowing the process to continue.

4.8 Alarms

Alarms are specializations of *AcknowledgeableConditions* that add the concepts of an *Active* state, a *Shelving* state and a *Suppressed* state to a *Condition*. The state model is illustrated in Figure 5

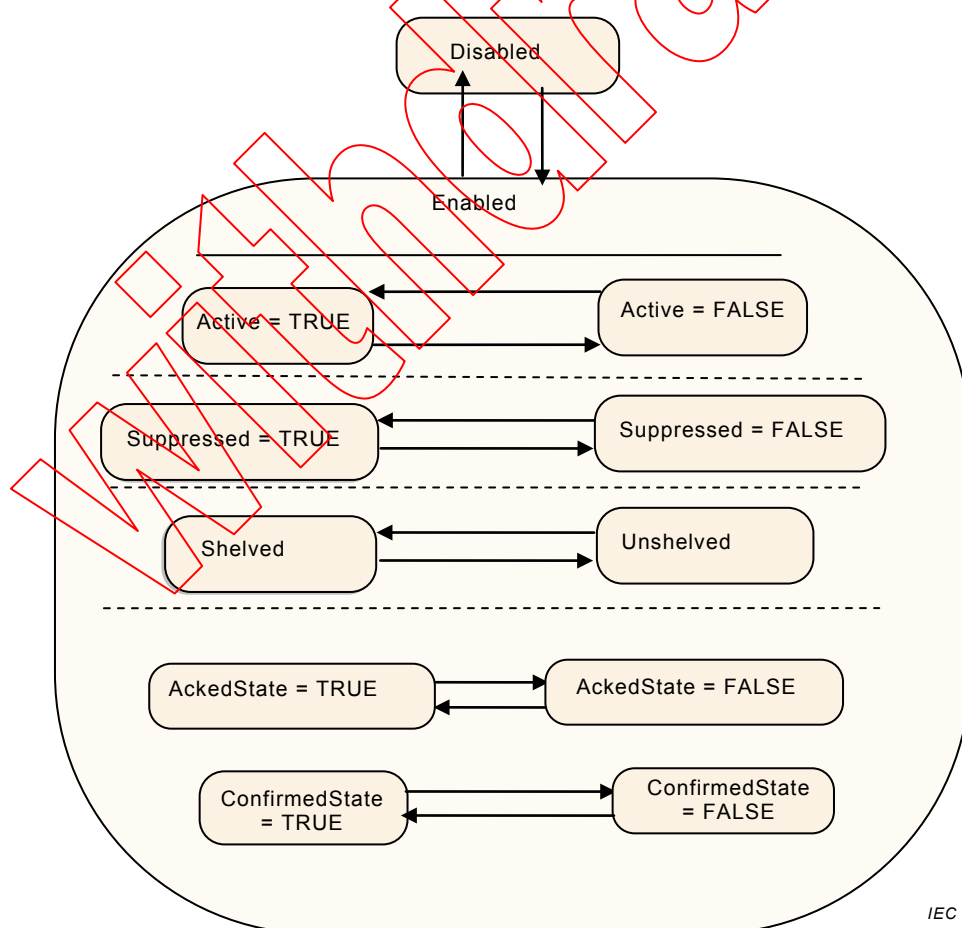


Figure 5 – Alarm State Machine Model

An *Alarm* in the *Active* state indicates that the situation the *Condition* is representing currently exists. When an *Alarm* is in an inactive state it is representing a situation that has returned to a normal state.

Some *Alarm* subtypes introduce sub-states of the *Active* state. For example an *Alarm* representing a temperature may provide a high level state as well as a critically high state (see following Clause).

The *Shelving* state can be set by an *Operator* via OPC UA *Methods*. The *Suppressed* state is set internally by the *Server* due to system specific reasons. *Alarm* systems typically implement the *Suppress* and *Shelve* features to help keep *Operators* from being overwhelmed during *Alarm* “storms” by limiting the number of *Alarms* an *Operator* sees on a current *Alarm* display. This is accomplished by setting the *SuppressedOrShelved* flag on second order dependent *Alarms* and/or *Alarms* of less severity, leading the *Operator* to concentrate on the most critical issues.

The *Shelved* and *Suppressed* states differ from the *Disabled* state in that *Alarms* are still fully functional and can be included in *Subscription Notifications* to a *Client*.

4.9 Multiple Active States

In some cases it is desirable to further define the *Active* state of an *Alarm* by providing a sub-state machine for the *Active* State. For example a multi-state level *Alarm* when in the *Active* state may be in one of the following sub-states: LowLow, Low, High or HighHigh. The state model is illustrated in Figure 6.

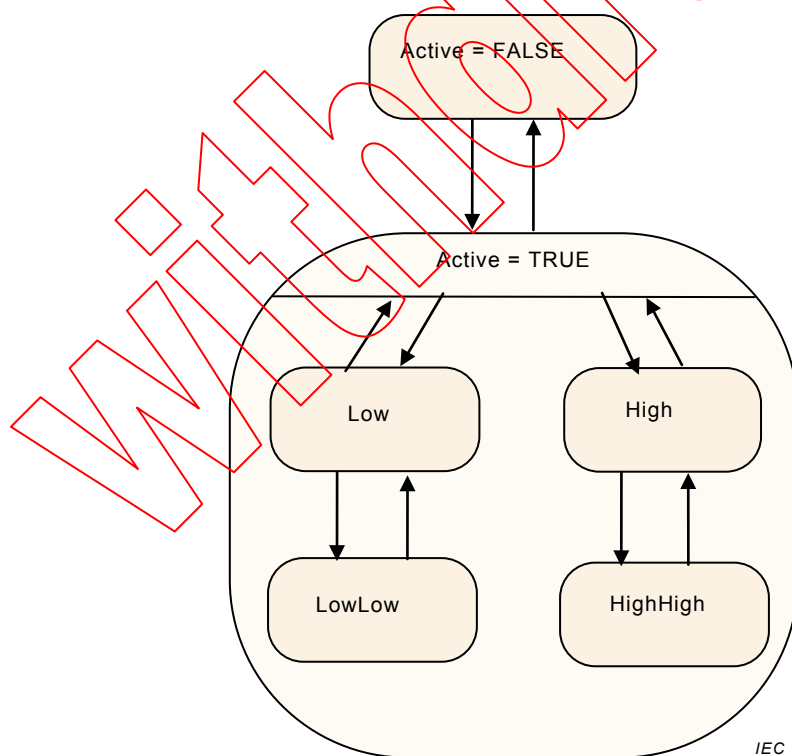


Figure 6 – Multiple Active States Example

With the multi-state *Alarm* model, state transitions among the sub-states of *Active* are allowed without causing a transition out of the *Active* state.

To accommodate different use cases both a (mutually) exclusive and a non-exclusive model are supported.

Exclusive means that the *Alarm* can only be in one sub-state at a time. If for example a temperature exceeds the HighHigh limit the associated exclusive LevelAlarm will be in the HighHigh sub-state and not in the High sub-state.

Some *Alarm* systems, however, allow multiple sub-states to exist in parallel. This is called non-exclusive. In the previous example where the temperature exceeds the HighHigh limit a non-exclusive LevelAlarm will be both in the High and the HighHigh sub-state.

4.10 Condition Instances in the Address Space

Because *Conditions* always have a state (*Enabled* or *Disabled*) and possibly many sub-states it makes sense to have instances of *Conditions* present in the *AddressSpace*. If the *Server* exposes *Condition* instances they usually will appear in the *AddressSpace* as components of the *Objects* that “own” them. For example a temperature transmitter that has a built-in high temperature *Alarm* would appear in the *AddressSpace* as an instance of some temperature transmitter *Object* with a *HasComponent Reference* to an instance of a *LevelAlarmType*.

The availability of instances allows Data Access *Clients* to monitor the current *Condition* state by subscribing to the *Attribute* values of *Variable Nodes*.

While exposing *Condition* instances in the *AddressSpace* is not always possible, doing so allows for direct interaction (read, write and *Method* invocation) with a specific *Condition* instance. For example, if a *Condition* instance is not exposed, there is no way to invoke the *Enable* or *Disable Method* for the specific *Condition* instance.

4.11 Alarm and Condition Auditing

The OPC UA Standards include provisions for auditing. Auditing is an important security and tracking concept. Audit records provide the “Who”, “When” and “What” information regarding user interactions with a system. These audit records are especially important when *Alarm* management is considered. *Alarms* are the typical instrument for providing information to a user that something needs the user’s attention. A record of how the user reacts to this information is required in many cases. Audit records are generated for all *Method* calls that affect the state of the system, for example an *Acknowledge Method* call would generate an *AuditConditionAck Event*.

The standard *AuditEventTypes* defined in IEC 62541-5 already includes the fields required for *Condition* related audit records. To allow for filtering and grouping, this standard defines a number of sub-types of the *AuditEventTypes* but without adding new fields to them.

This standard describes the *AuditEventType* that each *Method* is required to generate. For example, the *Disable Method* has an *AlwaysGeneratesEvent Reference* to an *AuditConditionEnableEventType*. An *Event* of this type shall be generated for every invocation of the *Method*. The audit *Event* describes the user interaction with the system, in some cases this interaction may affect more than one *Condition* or be related to more than one state.

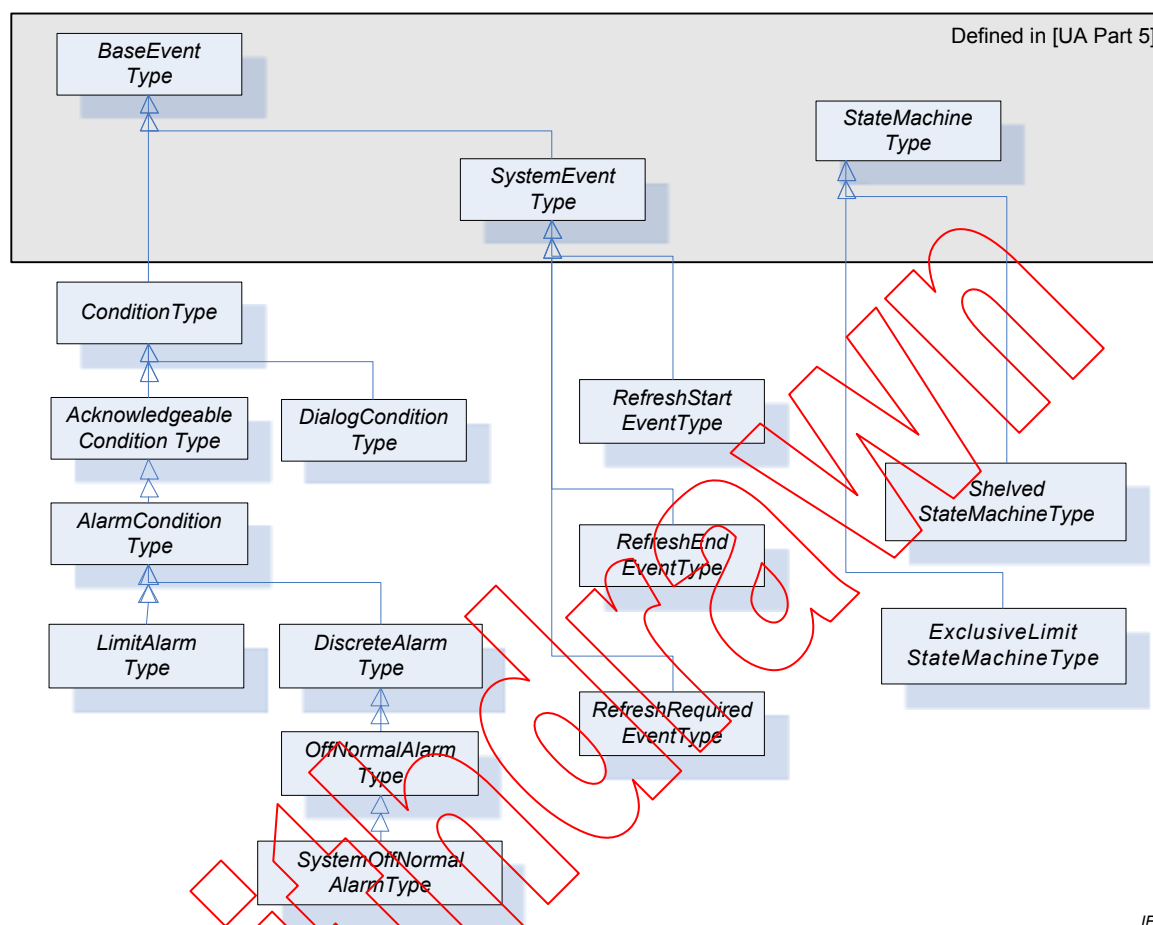
5 Model

5.1 General

The *Alarm* and *Condition* model extends the OPC UA base *Event* model by defining various *Event Types* based on the *BaseEventType*. All of the *Event Types* defined in this standard can be further extended to form domain or *Server* specific *Alarm* and *Condition Types*.

Instances of *Alarm* and *Condition Types* may be optionally exposed in the *AddressSpace* in order to allow direct access to the state of an *Alarm* or *Condition*.

The following subclauses define the OPC UA *Alarm* and *Condition* Types. Figure 7 informally describes the hierarchy of these *Types*. Subtypes of the *LimitAlarmType* and the *DiscreteAlarmType* are not shown. The full *AlarmConditionType* hierarchy can be found in Figure 11.



IEC

Figure 7 – ConditionType Hierarchy

5.2 Two-State State Machines

Most states defined in this standard are simple – i.e. they are either TRUE or FALSE. The *TwoStateVariableType* is introduced specifically for this use case. More complex states are modelled by using a *StateMachineType* defined in IEC 62541-5.

The *TwoStateVariableType* is derived from the *StateVariableType* defined in IEC 62541-5 and formally defined in Table 3.

Table 3 – TwoStateVariableType Definition

Attribute	Value				
BrowseName	TwoStateVariableType				
DataType	LocalizedText				
ValueRank	-1 (-1 = Scalar)				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	Modelling Rule
Subtype of the <i>StateVariableType</i> defined in IEC 62541-5. Note that a <i>Reference</i> to this subtype is not shown in the definition of the <i>StateVariableType</i>					
HasProperty	Variable	Id	Boolean	PropertyType	Mandatory
HasProperty	Variable	TransitionTime	UtcTime	PropertyType	Optional
HasProperty	Variable	EffectiveTransitionTime	UtcTime	PropertyType	Optional
HasProperty	Variable	TrueState	LocalizedText	PropertyType	Optional
HasProperty	Variable	FalseState	LocalizedText	PropertyType	Optional
HasTrueSubState	StateMachine or TwoStateVariableType	<StateIdentifier>	Defined in 5.4.2		Optional
HasFalseSubState	StateMachine or TwoStateVariableType	<StateIdentifier>	Defined in 5.4.3		Optional

The *Value Attribute* of a *TwoStateVariable* contains the current state as a human readable name. The *EnabledState* for example, might contain the name "Enabled" when TRUE and "Disabled" when FALSE.

Id is inherited from the *StateVariableType* and overridden to reflect the required *DataType* (Boolean). The value shall be the current state, i.e. either TRUE or FALSE.

TransitionTime specifies the time when the current state was entered.

EffectiveTransitionTime specifies the time when the current state or one of its sub states was entered. If, for example, a LevelAlarm is active and – while active – switches several times between High and HighHigh, then the *TransitionTime* stays at the point in time where the Alarm became active whereas the *EffectiveTransitionTime* changes with each shift of a sub state.

The optional *Property EffectiveDisplayName* from the *StateVariableType* is used if a state has sub states. It contains a human readable name for the current state after taking the state of any *SubStateMachines* in account. As an example, the *EffectiveDisplayName* of the *EnabledState* could contain "Active/HighHigh" to specify that the *Condition* is active and has exceeded the HighHigh limit.

Other optional *Properties* of the *StateVariableType* have no defined meaning for *TwoStateVariables*.

TrueState and *FalseState* contain the localized string for the *TwoStateVariable* value when its *Id Property* has the value TRUE or FALSE, respectively. Since the two *Properties* provide meta-data for the *Type*, *Servers* may not allow these *Properties* to be selected in the *Event* filter for a monitored item. *Clients* can use the Read *Service* to get the information from the specific *ConditionType*.

A *HasTrueSubState Reference* is used to indicate that the TRUE state has sub states.

A *HasFalseSubState Reference* is used to indicate that the FALSE state has sub states.

5.3 Condition Variables

Various information elements of a *Condition* are not considered to be states. However, a change in their value is considered important and supposed to trigger an *Event Notification*. These information elements are called *ConditionVariables*.

ConditionVariables are represented by a *ConditionVariableType* formally defined in Table 4.

Table 4 – ConditionVariableType Definition

Attribute	Value				
BrowseName	ConditionVariableType				
DataType	BaseDataType				
ValueRank	-2 (-2 = Any)				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	Modelling Rule
Subtype of the <i>BaseDataVariableType</i> defined in IEC 62541-5.					
HasProperty	Variable	SourceTimestamp	UtcTime	PropertyType	Mandatory

SourceTimestamp indicates the time of the last change of the *Value* of this *ConditionVariable*. It shall be the same time that would be returned from the *Read Service* inside the *DataValue* structure for the *ConditionVariable Value Attribute*.

5.4 Substate Reference Types

5.4.1 General

This Clause defines *ReferenceTypes* that are needed beyond those already specified as part of IEC 62541-3 and IEC 62541-5 to extend *TwoState* state machines with substates. These *References* will only exist when substates are available. For example if a *TwoState* machine is in a FALSE State, then any SubStates referenced from the TRUE state will not be available. If an Event is generated while in the FALSE state and information from the TRUE state substate is part of the data that is to be reported then this data would be reported as a NULL. With this approach *TwoStateVariables* can be extended with subordinate state machines in a similar fashion to the *StateMachineType* defined in IEC 62541-5.

5.4.2 HasTrueSubState ReferenceType

The *HasTrueSubState ReferenceType* is a concrete *ReferenceType* that can be used directly. It is a subtype of the *NonHierarchicalReferences ReferenceType*.

The semantics indicate that the sub state (the target *Node*) is a subordinate state of the TRUE super state. If more than one state within a *Condition* is a sub state of the same super state (i.e. several *HasTrueSubState References* exist for the same super state) they are all treated as independent substates. The representation in the *AddressSpace* is specified in Table 5.

The *SourceNode* of the *Reference* shall be an instance of a *TwoStateVariableType* and the *TargetNode* shall either be an instance of a *TwoStateVariableType* or an instance of a subtype of a *StateMachineType*.

It is not required to provide the *HasTrueSubState Reference* from super state to sub state, but it is required that the sub state provides the inverse *Reference* (*IsTrueSubStateOf*) to its super state.

Table 5 – HasTrueSubState ReferenceType

Attributes	Value		
BrowseName	HasTrueSubState		
InverseName	IsTrueSubStateOf		
Symmetric	False		
IsAbstract	False		
References	NodeClass	BrowseName	Comment

5.4.3 HasFalseSubState ReferenceType

The *HasFalseSubState ReferenceType* is a concrete *ReferenceType* that can be used directly. It is a subtype of the *NonHierarchicalReferences ReferenceType*.

The semantics indicate that the sub state (the target *Node*) is a subordinate state of the FALSE super state. If more than one state within a *Condition* is a sub state of the same super state (i.e. several *HasFalseSubState References* exist for the same super state) they are all treated as independent substates. The representation in the *AddressSpace* is specified in Table 6.

The *SourceNode* of the *Reference* shall be an instance of a *TwoStateVariableType* and the *TargetNode* shall either be an instance of a *TwoStateVariableType* or an instance of a subtype of a *StateMachineType*.

It is not required to provide the *HasFalseSubState Reference* from super state to sub state, but it is required that the sub state provides the inverse *Reference* (*IsFalseSubStateOf*) to its super state.

Table 6 – HasFalseSubState ReferenceType

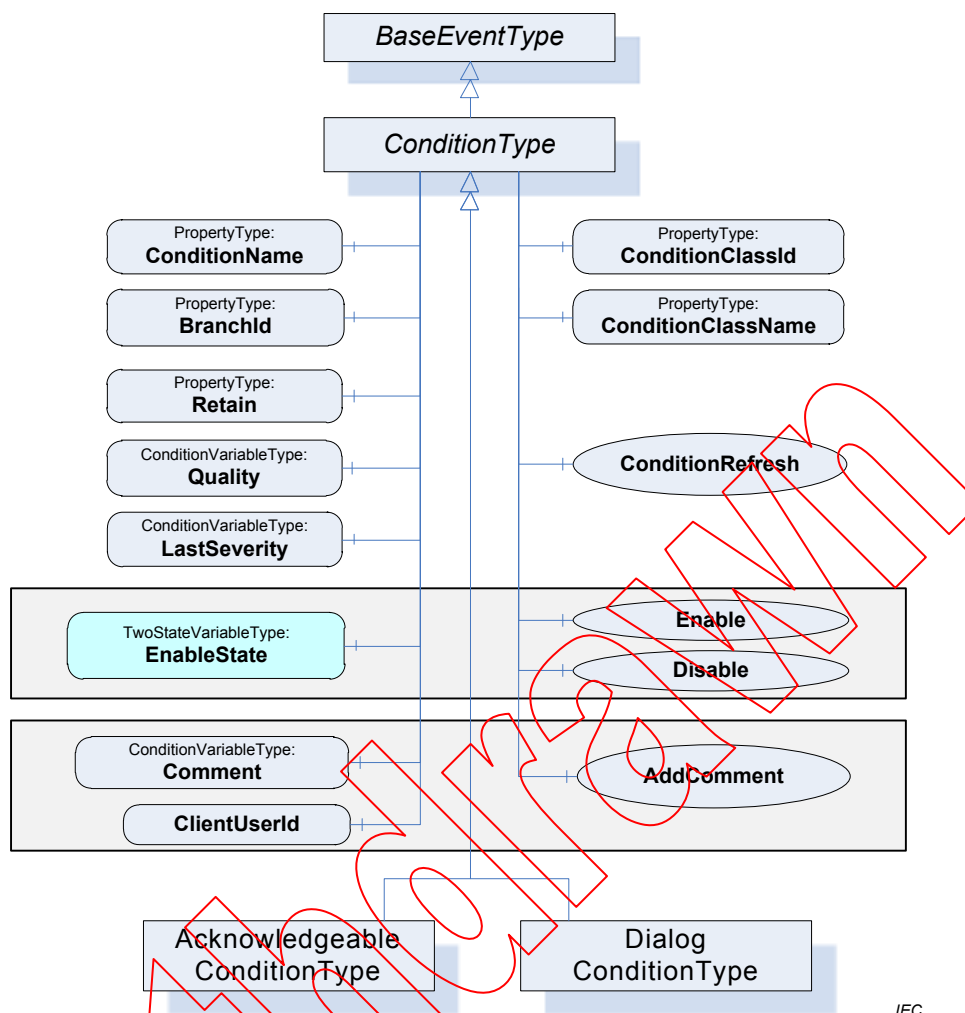
Attributes	Value		
BrowseName	HasFalseSubState		
InverseName	IsFalseSubStateOf		
Symmetric	False		
IsAbstract	False		
References	NodeClass	BrowseName	Comment

5.5 Condition Model

5.5.1 General

The *Condition* model extends the *Event* model by defining the *ConditionType*. The *ConditionType* introduces the concept of states differentiating it from the base *Event* model. Unlike the *BaseEventTypes*, *Conditions* are not transient. The *ConditionType* is further extended into *Dialog* and *AcknowledgeableConditionTypes*, each of which have their own subtypes.

The *Condition* model is illustrated in Figure 8 and formally defined in the subsequent tables. It is worth noting that this figure, like all figures in this standard, is not intended to be complete. Rather, the figures only illustrate information provided by the formal definitions.



IEC

Figure 8 – Condition Model

5.5.2 ConditionType

The *ConditionType* defines all general characteristics of a *Condition*. All other *ConditionTypes* derive from it. It is formally defined in Table 7. The FALSE state of the *EnabledState* shall not be extended with a sub state machine.

Table 7 – ConditionType Definition

Attribute	Value				
BrowseName	ConditionType				
IsAbstract	True				
References	NodeClass	BrowseName	Data Type	TypeDefinition	ModellingRule
Subtype of the <i>BaseEventType</i> defined in IEC 62541-5					
HasSubtype	ObjectType	DialogConditionType	Defined in 5.6.2		
HasSubtype	ObjectType	AcknowledgeableConditionType	Defined in 5.7.2		
HasProperty	Variable	ConditionClassId	NodeId	PropertyType	Mandatory
HasProperty	Variable	ConditionClassName	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	ConditionName	String	PropertyType	Mandatory
HasProperty	Variable	BranchId	NodeId	PropertyType	Mandatory
HasProperty	Variable	Retain	Boolean	PropertyType	Mandatory
HasComponent	Variable	EnabledState	LocalizedText	TwoStateVariableType	Mandatory
HasComponent	Variable	Quality	StatusCode	ConditionVariableType	Mandatory
HasComponent	Variable	LastSeverity	UInt16	ConditionVariableType	Mandatory
HasComponent	Variable	Comment	LocalizedText	ConditionVariableType	Mandatory
HasProperty	Variable	ClientUserId	String	PropertyType	Mandatory
HasComponent	Method	Disable	Defined in 5.5.4		Mandatory
HasComponent	Method	Enable	Defined in 5.5.5		Mandatory
HasComponent	Method	AddComment	Defined in 5.5.6		Mandatory
HasComponent	Method	ConditionRefresh	Defined in 5.5.7		None

The *ConditionType* inherits all *Properties* of the *BaseEventType*. Their semantic is defined in IEC 62541-5. *SourceNode* identifies the *ConditionSource*. See 5.12 for more details. If the *ConditionSource* is not a *Node* in the *AddressSpace* the *NodeId* is set to null. The *SourceNode* is the *Node* which the condition is associated with, it may be the same as the *InputNode* for an alarm, but it may be a separate node. For example a motor, which is a variable with a value that is an RPM, may be the *ConditionSource* for *Conditions* that are related to the motor as well as a temperature sensor associated with the motor. In the former the *InputNode* for the High RPM alarm is the value of the Motor RPM, while in the later the *InputNode* of the High Alarm would be the value of the temperature sensor that is associated with the motor.

ConditionClassId specifies in which domain this *Condition* is used. It is the *NodeId* of the corresponding *ConditionClassType*. See 5.9 for the definition of *ConditionClass* and a set of *ConditionClasses* defined in this standard. When using this *Property* for filtering, *Clients* have to specify all individual *ConditionClassType* *NodeIds*. The *OfType* operator cannot be applied. *BaseConditionClassType* is used as class whenever a *Condition* cannot be assigned to a more concrete class.

ConditionClassName provides the display name of the *ConditionClassType*.

ConditionName identifies the *Condition* instance that the *Event* originated from. It can be used together with the *SourceName* in a user display to distinguish between different *Condition* instances. If a *ConditionSource* has only one instance of a *ConditionType*, and the *Server* has no instance name, the *Server* shall supply the *ConditionType* browse name.

BranchId is Null for all *Event Notifications* that relate to the current state of the *Condition* instance. If *BranchId* is not Null it identifies a previous state of this *Condition* instance that still needs attention by an *Operator*. If the current *ConditionBranch* is transformed into a previous *ConditionBranch* then the *Server* needs to assign a non-null *BranchId*. An initial *Event* for the branch will generated with the values of the *ConditionBranch* and the new *BranchId*. The *ConditionBranch* can be updated many times before it is no longer needed. When the *ConditionBranch* no longer requires *Operator* input the final *Event* will have *Retain* set to FALSE. The retain bit on the current *Event* is TRUE, as long as any *ConditionBranches* require *Operator* input. See 4.4 for more information about the need for creating and maintaining previous *ConditionBranches* and Clause B.1 for an example using branches. The *BranchId* *Data Type* is *NodeId* although the *Server* is not required to have *ConditionBranches*.

in the *Address Space*. The use of a *NodeId* allows the *Server* to use simple numeric identifiers, strings or arrays of bytes.

Retain when TRUE describes a *Condition* (or *ConditionBranch*) as being in a state that is interesting for a *Client* wishing to synchronize its state with the *Server's* state. The logic to determine how this flag is set is *Server* specific. Typically all *Active Alarms* would have the *Retain* flag set; however, it is also possible for inactive *Alarms* to have their *Retain* flag set to TRUE.

In normal processing when a *Client* receives an *Event* with the *Retain* flag set to FALSE, the *Client* should consider this as a *ConditionBranch* that is no longer of interest, in the case of a “current *Alarm* display” the *ConditionBranch* would be removed from the display.

EnabledState indicates whether the *Condition* is enabled. *EnabledState/Id* is TRUE if enabled, FALSE otherwise. *EnabledState/TransitionTime* defines when the *EnabledState* last changed. Recommended state names are described in Annex A.

A *Condition's EnabledState* effects the generation of *Event Notifications* and as such results in the following specific behaviour:

- When the *Condition* instance enters the *Disabled* state, the *Retain Property* of this *Condition* shall be set to FALSE by the *Server* to indicate to the *Client* that the *Condition* instance is currently not of interest to *Clients*.
- When the *Condition* instance enters the enabled state, the *Condition* shall be evaluated and all of its *Properties* updated to reflect the current values. If this evaluation causes the *Retain Property* to transition to TRUE for any *ConditionBranch*, then an *Event Notification* shall be generated for that *ConditionBranch*.
- The *Server* may choose to continue to test for a *Condition* instance while it is *Disabled*. However, no *Event Notifications* will be generated while the *Condition* instance is disabled.
- For any *Condition* that exists in the *AddressSpace* the *Attributes* and the following *Variables* will continue to have valid values even in the *Disabled* state; *EventId*, *Event Type*, *Source Node*, *Source Name*, *Time*, and *EnabledState*. Other properties may no longer provide current valid values. All *Variables* that are no longer provided shall return a status of *Bad_ConditionDisabled*.

When enabled, changes to the following components shall cause a *ConditionType Event Notification*:

- *Quality*;
- *Severity* (inherited from *BaseEventType*);
- *Comment*.

This may not be the complete list. Sub-Types may define additional *Variables* that trigger *Event Notifications*. In general changes to *Variables* of the types *TwoStateVariableType* or *ConditionVariableType* trigger *Event Notifications*.

Quality reveals the status of process values or other resources that this *Condition* instance is based upon. If, for example, a process value is “Uncertain”, the associated “*LevelAlarm*” *Condition* is also questionable. Values for the *Quality* can be any of the OPC *StatusCodes* defined in IEC 62541-8 as well as *Good*, *Uncertain* and *Bad* as defined in IEC 62541-4. These *StatusCodes* are similar to but slightly more generic than the description of data quality in the various *Fieldbus Specifications*. It is the responsibility of the *Server* to map internal status information to these codes. A *Server* which supports no quality information shall return *Good*. This quality can also reflect the communication status associated with the system that this value or resource is based on and from which this *Alarm* was received. For communication errors to the underlying system, especially those that result in some unavailable *Event* fields, the quality shall be *Bad_NoCommunication* error.

Events are only generated for *Conditions* that have their *Retain* field set to true.

LastSeverity provides the previous severity of the *ConditionBranch*. Initially this *Variable* contains a zero value; it will return a value only after a severity change. The new severity is supplied via the *Severity Property* which is inherited from the *BaseEventType*.

Comment contains the last comment provided for a certain state (*ConditionBranch*). It may have been provided by an *AddComment Method*, some other *Method* or in some other manner. The initial value of this *Variable* is null, unless it is provided in some other manner. If a *Method* provides as an option the ability to set a *Comment*, then the value of this *Variable* is reset to null if an optional comment is not provided.

ClientUserId is related to the *Comment* field and contains the identity of the user who inserted the most recent *Comment*. The logic to obtain the *ClientUserId* is defined in IEC 62541-5.

The *NodeId* of the *Condition* instance is used as *ConditionId*. It is not explicitly modelled as a component of the *ConditionType*. However, it can be requested with the following *SimpleAttributeOperand*, Table 8, in the *SelectClause* of the *EventFilter*.

Table 8 – Simple Attribute Operand

Name	Type	Description
SimpleAttributeOperand		
typeId	NodeId	NodeId of the <i>ConditionType</i> Node
browsePath[]	QualifiedName	empty
attributeId	IntegerId	Id of the <i>NodeId</i> Attribute

5.5.3 Condition and Branch Instances

Conditions are *Objects* which have a state which changes over time. Each *Condition* instance has the *ConditionId* as identifier which uniquely identifies it within the *Server*.

A *Condition* instance may be an *Object* that appears in the *Server Address Space*. If this is the case the *ConditionId* is the *NodeId* for the *Object*.

The state of a *Condition* instance at any given time is the set values for the *Variables* that belong to the *Condition* instance. If one or more *Variable* values change the *Server* generates an *Event* with a unique *EventId*.

If a *Client* calls *Refresh* the *Server* will report the current state of a *Condition* instance by re-sending the last *Event* (i.e. the same *EventId* and *Time* is sent).

A *ConditionBranch* is a copy of the *Condition* instance state that can change independently of the current *Condition* instance state. Each *Branch* has an identifier called a *BranchId* which is unique among all active *Branches* for a *Condition* instance. *Branches* are typically not visible in the *Address Space* and this standard does not define a standard way to make them visible.

5.5.4 Disable Method

Disable used to change a *Condition* instance to the *Disabled* state. Normally, the *MethodId* passed to the *Call Service* is found by browsing the *Condition* instance in the *AddressSpace*. However, some *Servers* do not expose *Condition* instances in the *AddressSpace*. Therefore all *Servers* shall allow *Clients* to call the *Disable* Method by specifying *ConditionId* as the *ObjectId* and the well known *NodeId* of the *Method* declaration on the *ConditionType* as the *MethodId*.

Signature

`Disable () ;`

Method Result Codes in Table 9 (defined in Call Service)

Table 9 – Disable Result Codes

ResultCode	Description
Bad_ConditionAlreadyDisabled	See Table 70 for the description of this result code.

Table 10 specifies the *AddressSpace* representation for the *Disable Method*.

Table 10 – Disable Method AddressSpace Definition

Attribute	Value				
BrowseName	Disable				
References	NodeClass	BrowseName	Data Type	TypeDefinition	ModellingRule
AlwaysGeneratesEvent	Defined in 5.10.2			AuditConditionEnableEventType	

5.5.5 Enable Method

Enable is used to change a *Condition* instance to the enabled state. Normally, the *MethodId* passed to the Call Service is found by browsing the *Condition* instance in the *AddressSpace*. However, some *Servers* do not expose *Condition* instances in the *AddressSpace*. Therefore all *Servers* shall allow *Clients* to call the *Enable Method* by specifying *ConditionId* as the *ObjectId* and the well known *NodeId* of the *Method* declaration on the *ConditionType* as the *MethodId*. If the *Condition* instance is not exposed, then it may be difficult for a *Client* to determine the *ConditionId* for a disabled *Condition*.

Signature

Enable () ;

Method Result Codes in Table 11 (defined in Call Service)

Table 11 – Enable Result Codes

ResultCode	Description
Bad_ConditionAlreadyEnabled	See Table 70 for the description of this result code.

Table 12 specifies the *AddressSpace* representation for the *Enable Method*.

Table 12 – Enable Method AddressSpace Definition

Attribute	Value				
BrowseName	Enable				
References	NodeClass	BrowseName	Data Type	TypeDefinition	ModellingRule
AlwaysGeneratesEvent	Defined in 5.10.2			AuditConditionEnableEventType	

5.5.6 AddComment Method

AddComment is used to apply a comment to a specific state of a *Condition* instance. Normally, the *MethodId* passed to the Call Service is found by browsing the *Condition* instance in the *AddressSpace*. However, some *Servers* do not expose *Condition* instances in the *AddressSpace*. Therefore all *Servers* shall allow *Clients* to call the *AddComment Method* by specifying *ConditionId* as the *ObjectId* and the well known *NodeId* of the *Method* declaration on the *ConditionType* as the *MethodId*. The *Method* cannot be called on the *ConditionType Node*.

Signature

```

AddComment (
    [in] ByteString      EventId
    [in] LocalizedText   Comment
);

```

The parameters are defined in Table 13

Table 13 – AddComment Arguments

Argument	Description
EventId	EventId identifying a particular <i>Event Notification</i> where a state was reported for a <i>Condition</i> .
Comment	A localized text to be applied to the <i>Condition</i> .

Method Result Codes in Table 14 (defined in Call Service).

Table 14 – AddComment result Codes

ResultCode	Description
Bad_MethodInvalid	See IEC 62541-4 for the description of this result code. The addressed <i>Condition</i> does not support adding comments.
Bad_EventIdUnknown	See Table 70 for the description of this result code.
Bad_NodeIdUnknown	See IEC 62541-4 for the description of this result code. Used to indicate that the specified <i>Condition</i> is not valid or that the <i>Method</i> was called on the <i>ConditionType Node</i> .

Comments

Comments are added to *Event* occurrences identified via an *EventId*. *EventIds* where the related *EventType* does not support *Comments* at all are rejected.

A *ConditionEvent* – where the *Comment Variable* contains this text – will be sent for the identified state. If a comment is added to a previous state (i.e. a state for which the Server has created a branch), the *BranchId* and all *Condition* values of this branch will be reported.

Table 15 specifies the *AddressSpace* representation for the *AddComment Method*.

Table 15 – AddComment Method AddressSpace Definition

Attribute	Value				
BrowseName	AddComment				
References	NodeClass	BrowseName	Data Type	TypeDefinition	ModellingRule
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGenerates Event	Defined in 5.10.4			AuditConditionCommentEventType	

5.5.7 ConditionRefresh Method

ConditionRefresh allows a *Client* to request a *Refresh* of all *Condition* instances that currently are in an interesting state (they have the *Retain* flag set). This includes previous states of a *Condition* instance for which the *Server* maintains *Branches*. A *Client* would typically invoke this *Method* when it initially connects to a *Server* and following any situations, such as communication disruptions, in which it would require resynchronization with the *Server*. This *Method* is only available on the *ConditionType* or its subtypes. To invoke this *Method*, the call shall pass the well known *MethodId* of the *Method* on the *ConditionType* and the *ObjectId* shall be the well known *ObjectId* of the *ConditionType Object*.

Signature

```
ConditionRefresh (
    [in] IntegerId      SubscriptionId
);
```

The parameters are define in Table 16

Table 16 – ConditionRefresh Parameters

Argument	Description
SubscriptionId	A valid <i>Subscription</i> Id of the <i>Subscription</i> to be refreshed.

Method Result Codes in Table 17 (defined in Call Service).

Table 17 – ConditionRefresh ReturnCodes

ResultCode	Description
Bad_SubscriptionIdInvalid	See IEC 62541-4 for the description of this result code
Bad_RefreshInProgress	See Table 70 for the description of this result code

Comments

Subclause 4.5 describes the concept, use cases and information flow in more detail.

The input argument provides a *Subscription* identifier indicating which *Client Subscription* shall be refreshed. If the *Subscription* is accepted the *Server* will react as follows:

- 1) The *Server* issues a *RefreshStartEvent* (defined in 5.11.2) marking the start of *Refresh*. A copy of the *RefreshStartEvent* is queued into the *Event* stream for every *Notifier MonitoredItem* in the *Subscription*. Each of the *Event* copies shall contain the same *EventId*.
- 2) The *Server* issues *Event Notifications* of any *Retained Conditions* and *Retained Branches* of *Conditions* that meet the *Subscriptions* content filter criteria. Note that the *EventId* for such a refreshed *Notification* shall be identical to the one for the original *Notification*.
- 3) The *Server* may intersperse new *Event Notifications* that have not been previously issued to the notifier along with those being sent as part of the *Refresh* request. *Clients* shall check for multiple *Event Notifications* for a *ConditionBranch* to avoid overwriting a new state delivered together with an older state from the *Refresh* process.
- 4) The *Server* issues a *RefreshEndEvent* (defined in 5.11.3) to signal the end of the *Refresh*. A copy of the *RefreshEndEvent* is queued into the *Event* stream for every *Notifier MonitoredItem* in the *Subscription*. Each of the *Events* copies shall contain the same *EventId*.

If more than one *Subscription* is to be refreshed, then the standard call *Service* array processing can be used.

As mentioned above, *ConditionRefresh* shall also issue *Event Notifications* for prior states if they still need attention. In particular this is true for *Condition* instances where previous states still need acknowledgement or confirmation.

Table 18 specifies the *AddressSpace* representation for the *ConditionRefresh Method*.

Table 18 – ConditionRefresh Method AddressSpace Definition

Attribute	Value				
BrowseName	ConditionRefresh				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGeneratesEvent	Defined in 5.11.2			RefreshStartEvent	
AlwaysGeneratesEvent	Defined in 5.11.3			RefreshEndEvent	

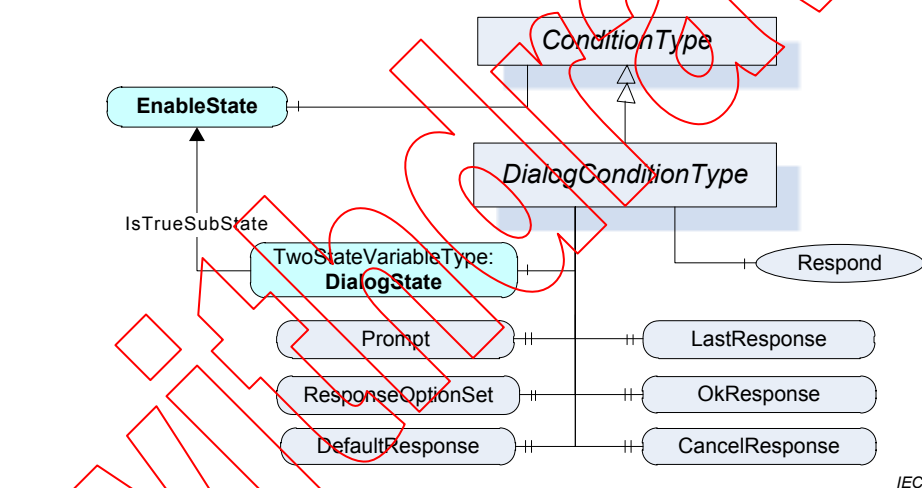
5.6 Dialog Model

5.6.1 General

The Dialog Model is an extension of the *Condition* model used by a *Server* to request user input. It provides functionality similar to the standard *Message* dialogs found in most operating systems. The model can easily be customized by providing *Server* specific response options in the *ResponseOptionSet* and by adding additional functionality to derived *Condition* Types.

5.6.2 DialogConditionType

The *DialogConditionType* is used to represent *Conditions* as dialogs. It is illustrated in Figure 9 and formally defined in Table 19.

**Figure 9 – DialogConditionType Overview****Table 19 – DialogConditionType Definition**

Attribute	Value				
BrowseName	DialogConditionType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	Modelling Rule
Subtype of the ConditionType defined in 5.5.2					
HasComponent	Variable	DialogState	LocalizedText	TwoStateVariableType	Mandatory
HasProperty	Variable	Prompt	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	ResponseOptionSet	LocalizedText []	PropertyType	Mandatory
HasProperty	Variable	DefaultResponse	Int32	PropertyType	Mandatory
HasProperty	Variable	LastResponse	Int32	PropertyType	Mandatory
HasProperty	Variable	OkResponse	Int32	PropertyType	Mandatory
HasProperty	Variable	CancelResponse	Int32	PropertyType	Mandatory
HasComponent	Method	Respond	Defined in 5.6.3.		Mandatory

The *DialogConditionType* inherits all *Properties* of the *ConditionType*.

DialogState/Id when set to TRUE indicates that the *Dialog* is active and waiting for a response. Recommended state names are described in Annex A.

Prompt is a dialog prompt to be shown to the user.

ResponseOptionSet specifies the desired set of responses as array of *LocalizedText*. The index in this array is used for the corresponding fields like *DefaultResponse*, *LastResponse* and *SelectedOption* in the *Respond Method*. The recommended localized names for the common options are described in Annex A.

Typical combinations of response options are

- OK
- OK, Cancel
- Yes, No, Cancel
- Abort, Retry, Ignore
- Retry, Cancel
- Yes, No

DefaultResponse identifies the response option that should be shown as default to the user. It is the index in the *ResponseOptionSet* array. If no response option is the default, the value of the *Property* is -1.

LastResponse contains the last response provided by a *Client* in the *Respond Method*. If no previous response exists then the value of the *Property* is -1.

OkResponse provides the index of the OK option in the *ResponseOptionSet* array. This choice is the response that will allow the system to proceed with the operation described by the prompt. This allows a *Client* to identify the OK option if a special handling for this option is available. If no OK option is available the value of this *Property* is -1.

CancelResponse provides the index of the response in the *ResponseOptionSet* array that will cause the Dialog to go into the inactive state without proceeding with the operation described by the prompt. This allows a *Client* to identify the Cancel option if a special handling for this option is available. If no Cancel option is available the value of this *Property* is -1.

5.6.3 Respond Method

Respond is used to pass the selected response option and end the dialog. *DialogState/Id* will return to FALSE.

Signature

```
Respond (
    [in] Int32 SelectedResponse
);
```

The parameters are defined in Table 20.

Table 20 – Repond Parameters

Argument	Description
SelectedResponse	Selected index of the <i>ResponseOptionSet</i> array.

Method Result Codes in Table 21 (defined in Call Service).

Table 21 – Respond ResultCodes

ResultCode	Description
Bad_DialogNotActive	See Table 70 for the description of this result code.
Bad_DialogResponseInvalid	See Table 70 for the description of this result code.

Table 22 specifies the *AddressSpace* representation for the *Respond Method*.

Table 22 – Respond Method AddressSpace Definition

Attribute	Value				
BrowseName	Respond				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGeneratesEvent	Defined in 5.10.5			AuditConditionRespondEventType	

5.7 Acknowledgeable Condition Model

5.7.1 General

The *Acknowledgeable Condition* Model extends the *Condition* model. States for acknowledgement and confirmation are added to the *Condition* model.

AcknowledgeableConditions are represented by the *AcknowledgeableConditionType* which is a subtype of the *ConditionType*. The model is formally defined in the following subclauses.

5.7.2 AcknowledgeableConditionType

The *AcknowledgeableConditionType* extends the *ConditionType* by defining acknowledgement characteristics. It is an abstract type. The *AcknowledgeableConditionType* is illustrated in Figure 10 and formally defined in Table 23.

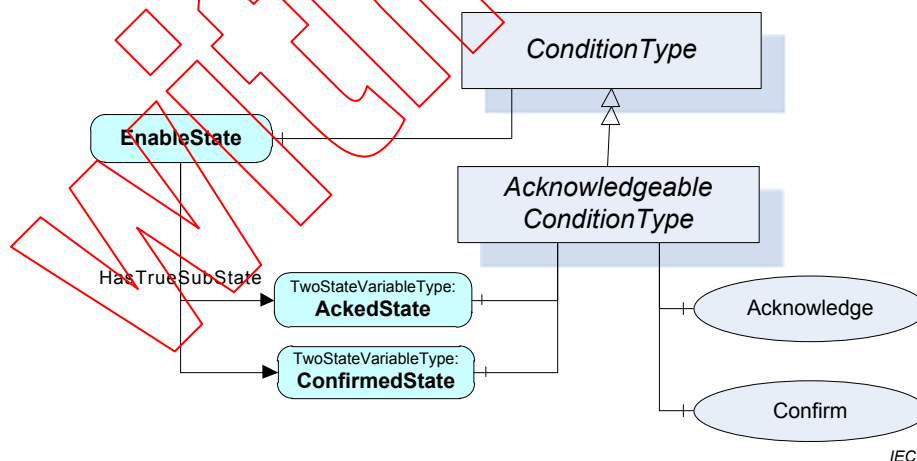
**Figure 10 – AcknowledgeableConditionType Overview**

Table 23 – AcknowledgeableConditionType Definition

Attribute	Value				
BrowseName	AcknowledgeableConditionType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the <i>ConditionType</i> defined in 5.5.2.					
HasSubtype	ObjectType	AlarmConditionType	Defined in 5.8.2		
HasComponent	Variable	AckedState	LocalizedText	TwoStateVariableType	Mandatory
HasComponent	Variable	ConfirmedState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Method	Acknowledge	Defined in 5.7.3		Mandatory
HasComponent	Method	Confirm	Defined in 5.7.4		Optional

The *AcknowledgeableConditionType* inherits all *Properties* of the *ConditionType*.

AckedState when FALSE indicates that the *Condition* instance requires acknowledgement for the reported *Condition* state. When the *Condition* instance is acknowledged the *AckedState* is set to TRUE. *ConfirmedState* indicates whether it requires confirmation. Recommended state names are described in Annex A. The two states are sub-states of the TRUE *EnabledState*. See 4.3 for more information about acknowledgement and confirmation models. The *EventId* used in the *Event Notification* is considered the identifier of this state and has to be used when calling the *Methods* for acknowledgement or confirmation.

A *Server* may require that previous states be acknowledged. If the acknowledgement of a previous state is still open and a new state also requires acknowledgement, the *Server* shall create a branch of the *Condition* instance as specified in 4.4. *Clients* are expected to keep track of all *ConditionBranches* where *AckedState/Id* is FALSE to allow acknowledgement of those. See also 5.5.2 for more information about *ConditionBranches* and the examples in Clause B.1. The handling of the *AckedState* and branches also applies to the *ConfirmState*.

5.7.3 Acknowledge Method

Acknowledge is used to acknowledge an *Event Notification* for a *Condition* instance state where *AckedState* was set to FALSE. Normally, the *MethodId* passed to the *Call Service* is found by browsing the *Condition* instance in the *AddressSpace*. However, some *Servers* do not expose *Condition* instances in the *AddressSpace*. Therefore all *Servers* shall allow *Clients* to call the *Acknowledge Method* by specifying *ConditionId* as the *ObjectId* and the well known *NodeId* of the *Method* declaration on the *AcknowledgeableConditionType* as the *MethodId*. The *Method* cannot be called on the *AcknowledgeableConditionType Node*.

Signature

```

Acknowledge (
    [in] ByteString      EventId
    [in] LocalizedText   Comment
);

```

The parameters are defined in Table 24.

Table 24 – Acknowledge Parameters

Argument	Description
EventId	EventId identifying a particular <i>Event Notification</i> . Only <i>Event Notifications</i> where <i>AckedState/Id</i> was FALSE can be acknowledged.
Comment	A localized text to be applied to the <i>Condition</i> .

Method Result Codes in Table 25 (defined in *Call Service*).

Table 25 – Acknowledge result codes

ResultCode	Description
Bad_ConditionBranchAlreadyAked	See Table 70 for the description of this result code.
Bad_EventIdUnknown	See Table 70 for the description of this result code.
Bad_NodeIdUnknown	See IEC 62541-4 for the description of this result code. Used to indicate that the specified <i>Condition</i> is not valid or that the <i>Method</i> was called on the <i>ConditionType Node</i> .

Comments

A *Server* is responsible to ensure that each *Event* has a unique *EventId*. This allows *Clients* to identify and acknowledge a particular *Event Notification*.

The *EventId* identifies a specific *Event Notification* where a state to be acknowledged was reported. Acknowledgement and the optional comment will be applied to the state identified with the *EventId*. If the comment field is NULL (both locale and text are empty) it will be ignored and any existing comments will remain unchanged. If the comment is to be reset, an empty text with a locale shall be provided.

A valid *EventId* will result in an *Event Notification* where *AckedState/Id* is set to TRUE and the *Comment Property* contains the text of the optional comment argument. If a previous state is acknowledged, the *BranchId* and all *Condition* values of this branch will be reported. Table 26 specifies the *AddressSpace* representation for the *Acknowledge Method*.

Table 26 – Acknowledge Method AddressSpace Definition

Attribute	Value				
BrowseName	Acknowledge				
References	NodeClass	BrowseName	Data Type	TypeDefinition	ModellingRule
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGeneratesEvent	Defined in 5.10.5			AuditConditionAcknowledge EventType	

5.7.4 Confirm Method

Confirm is used to confirm an *Event Notifications* for a *Condition* instance state where *ConfirmedState* was set to FALSE. Normally, the *MethodId* passed to the *Call Service* is found by browsing the *Condition* instance in the *AddressSpace*. However, some *Servers* do not expose *Condition* instances in the *AddressSpace*. Therefore all *Servers* shall allow *Clients* to call the *Confirm Method* by specifying *ConditionId* as the *ObjectId* and the well known *NodeId* of the *Method* declaration on the *AcknowledgeableConditionType* as the *MethodId*. The *Method* cannot be called on the *AcknowledgeableConditionType Node*.

Signature

```
Confirm (
    [in] ByteString      EventId
    [in] LocalizedText   Comment
);
```

The parameters are defined in Table 27.

Table 27 – Confirm Method Parameters

Argument	Description
EventId	<i>EventId</i> identifying a particular <i>Event Notification</i> . Only <i>Event Notifications</i> where <i>ConfirmedState/Id</i> was TRUE can be confirmed.
Comment	A localized text to be applied to the <i>Conditions</i> .

Method Result Codes in Table 28 (defined in *Call Service*).

Table 28 – Confirm Result Codes

ResultCode	Description
Bad_ConditionBranchAlreadyConfirmed	See Table 70 for the description of this result code.
Bad_EventIdUnknown	See Table 70 for the description of this result code.
Bad_NodeIdUnknown	See IEC 62541-4 for the description of this result code. Used to indicate that the specified <i>Condition</i> is not valid or that the <i>Method</i> was called on the <i>ConditionType Node</i> .

Comments

A *Server* is responsible to ensure that each *Event* has a unique *EventId*. This allows *Clients* to identify and confirm a particular *Event Notification*.

The *EventId* identifies a specific *Event Notification* where a state to be confirmed was reported. A *Comment* can be provided which will be applied to the state identified with the *EventId*.

A valid *EventId* will result in an *Event Notification* where *ConfirmedStateId* is set to TRUE and the *Comment Property* contains the text of the optional comment argument. If a previous state is confirmed, the *BranchId* and all *Condition* values of this branch will be reported.

Table 29 specifies the *AddressSpace* representation for the *Confirm Method*.

Table 29 – Confirm Method AddressSpace Definition

Attribute	Value				
BrowseName	Confirm				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGeneratesEvent	Defined in 5.10.7			AuditConditionConfirm EventType	

5.8 Alarm Model

5.8.1 General

Figure 11 informally describes the *AlarmConditionType*, its sub-types and where it is in the hierarchy of *Event Types*.

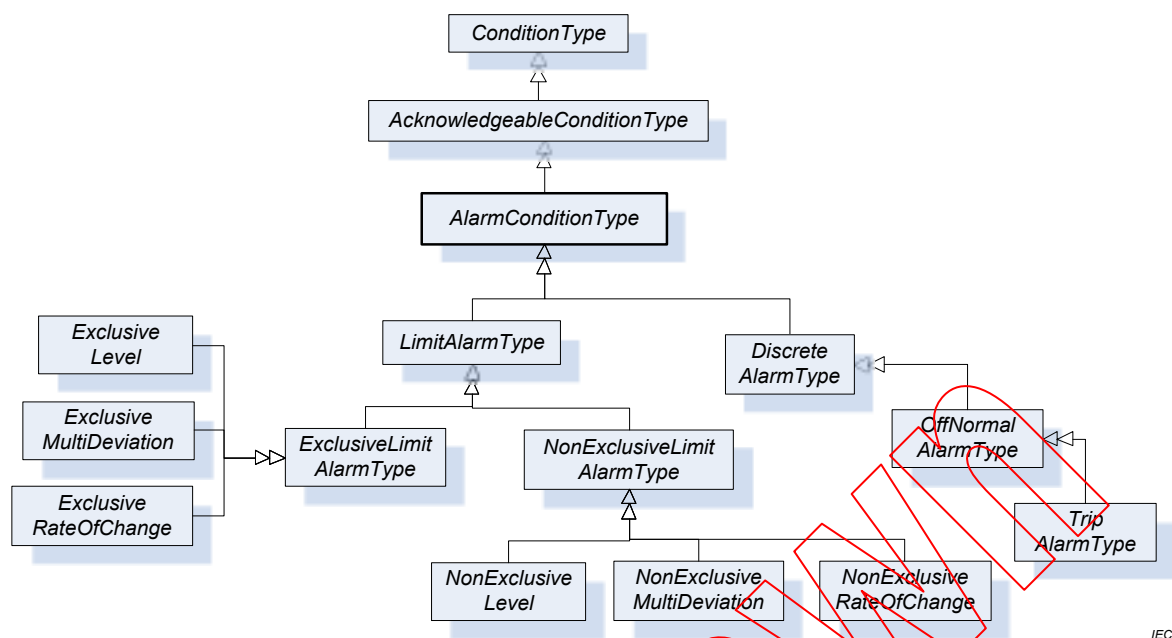


Figure 11 – AlarmConditionType Hierarchy Model

5.8.2 AlarmConditionType

The *AlarmConditionType* is an abstract type that extends the *AcknowledgeableConditionType* by introducing an *ActiveState*, *SuppressedState* and *ShelvingState*. The *Alarm* model is illustrated in Figure 12. This illustration is not intended to be a complete definition. It is formally defined in Table 30.

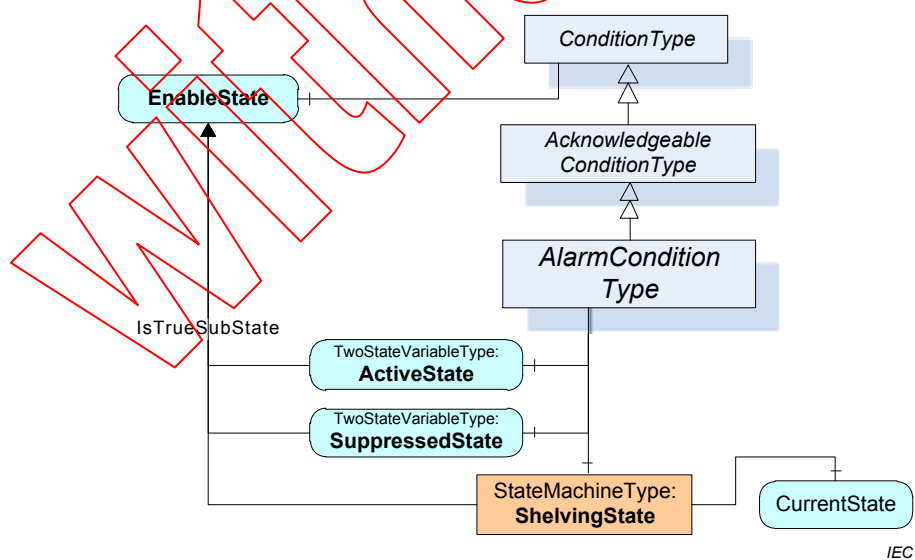


Figure 12 – Alarm Model

Table 30 – AlarmConditionType Definition

Attribute	Value				
BrowseName	AlarmConditionType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	Modelling Rule
Subtype of the AcknowledgeableConditionType defined in 5.7.2					
HasComponent	Variable	ActiveState	LocalizedText	TwoStateVariableType	Mandatory
HasProperty	Variable	InputNode	NodeId	PropertyType	Mandatory
HasComponent	Variable	SuppressedState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Object	ShelvingState		ShelvedStateMachineType	Optional
HasProperty	Variable	SuppressedOrShelved	Boolean	PropertyType	Mandatory
HasProperty	Variable	MaxTimeShelved	Duration	PropertyType	Optional

The *AlarmConditionType* inherits all *Properties* of the *AcknowledgeableConditionType*. The following states are sub-states of the TRUE *EnabledState*.

ActiveState/Id when set to TRUE indicates that the situation the *Condition* is representing currently exists. When a *Condition* instance is in the inactive state (*ActiveState/Id* when set to FALSE) it is representing a situation that has returned to a normal state. The transitions of *Conditions* to the inactive and *Active* states are triggered by *Server* specific actions. Sub-Types of the *AlarmConditionType* specified later in this standard will have sub-state models that further define the *Active* state. Recommended state names are described in Annex A.

The *InputNode Property* provides the *NodeId* of the *Variable* the *Value* of which is used as primary input in the calculation of the *Alarm* state. If this *Variable* is not in the *AddressSpace*, a Null *NodeId* shall be provided. In some systems, an *Alarm* may be calculated based on multiple *Variables Values*, it is up to the system to determine which *Variable's* *NodeId* is used.

SuppressState is used internally by a *Server* to automatically suppress *Alarms* due to system specific reasons. For example a system may be configured to suppress *Alarms* that are associated with machinery that is shutdown, such as a low level *Alarm* for a tank that is currently not in use. Recommended state names are described in Annex A.

ShelvingState suggests whether an *Alarm* shall (temporarily) be prevented from being displayed to the user. It is quite often used to block nuisance *Alarms*. The *ShelvingState* is defined in 5.8.3.

The *SuppressedState* and the *ShelvingState* together result in the *SuppressedOrShelved* status of the *Condition*. When an *Alarm* is in one of the states, the *SuppressedOrShelved* property will be set TRUE and this *Alarm* is then typically not displayed by the *Client*. State transitions associated with the *Alarm* do occur, but they are not typically displayed by the *Clients* as long as the *Alarm* remains in either the *Suppressed* or *Shelved* state.

The optional *Property MaxTimeShelved* is used to set the maximum time that an *Alarm Condition* may be shelved. The value is expressed as duration. Systems can use this *Property* to prevent permanent *Shelving* of an *Alarm*. If this *Property* is present it will be an upper limit on the duration passed into a *TimedShelve Method* call. If a value that exceeds the value of this property is passed to the *TimedShelve Method*, then a *Bad_ShelvingTimeOutOfRange* error code is returned on the call. If this *Property* is present it will also be enforced for the *OneShotShelved* state, in that an *Alarm Condition* will transition to the *Unshelved* state from the *OneShotShelved* state if the duration specified in this *Property* expires following a *OneShotShelve* operation without a change of any of the other items associated with the *Condition*.

More details about the *Alarm Model* and the various states can be found in 4.8.

5.8.3 ShelvedStateMachineType

5.8.3.1 Overview

The *ShelvedStateMachineType* defines a sub-state machine that represents an advanced *Alarm* filtering model. This model is illustrated in Figure 14.

The state model supports two types of *Shelving*: *OneShotShelving* and *TimedShelving*. They are illustrated in Figure 13. The illustration includes the allowed transitions between the various sub-states. *Shelving* is an *Operator* initiated activity.

In *OneShotShelving*, a user requests that an *Alarm* be Shelved for its current *Active* state. This type of *Shelving* is typically used when an *Alarm* is continually occurring on a boundary (i.e. a *Condition* is jumping between High *Alarm* and HighHigh *Alarm*, always in the *Active* state). The One Shot *Shelving* will automatically clear when an *Alarm* returns to an inactive state. Another use for this type of *Shelving* is for a plant area that is shutdown i.e. a long running *Alarm* such as a low level *Alarm* for a tank that is not in use. When the tank starts operation again the *Shelving* state will automatically clear.

In *TimedShelving*, a user specifies that an *Alarm* be shelved for a fixed time period. This type of *Shelving* is quite often used to block nuisance *Alarms*. For example, an *Alarm* that occurs more than 10 times in a minute may get shelved for a few minutes.

In all states, the *Unshelve* can be called to cause a transition to the Unshelve state; this includes *Un-shelving* an *Alarm* that is in the *TimedShelve* state before the time has expired and the *OneShotShelve* state without a transition to an inactive state.

All but two transitions are caused by *Method* calls as illustrated in Figure 13. The “Time Expired” transition is simply a system generated transition that occurs when the time value defined as part of the “Timed Shelved Call” has expired. The “Any Transition Occurs” transition is also a system generated transition; this transition is generated when the *Condition* goes to an inactive state.

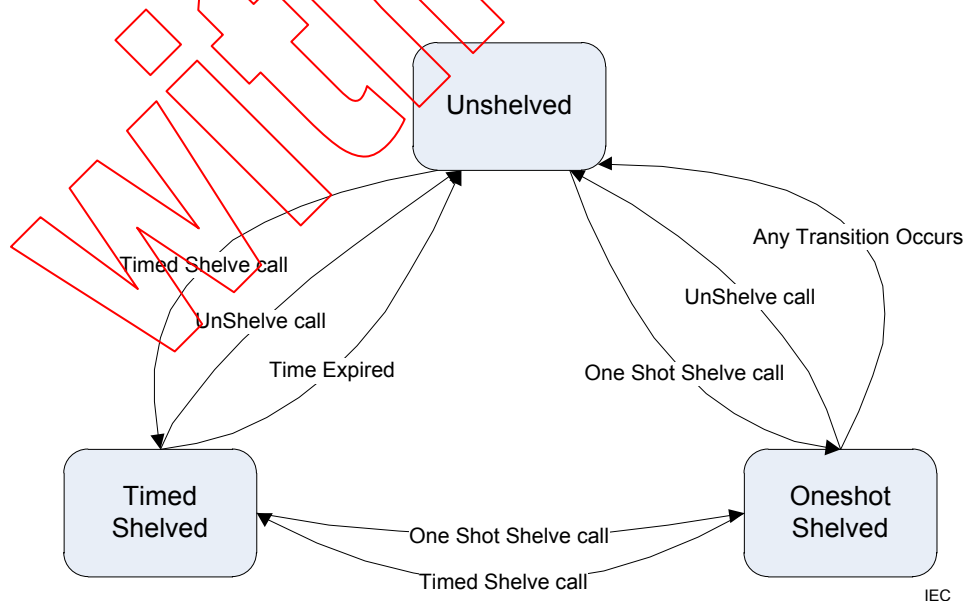


Figure 13 – Shelf state transitions

The *ShelvedStateMachine* includes a hierarchy of sub-states. It supports all transitions between Unshelved, OneShotShelved and TimedShelved.

The state machine is illustrated in Figure 14 and formally defined in Table 31.

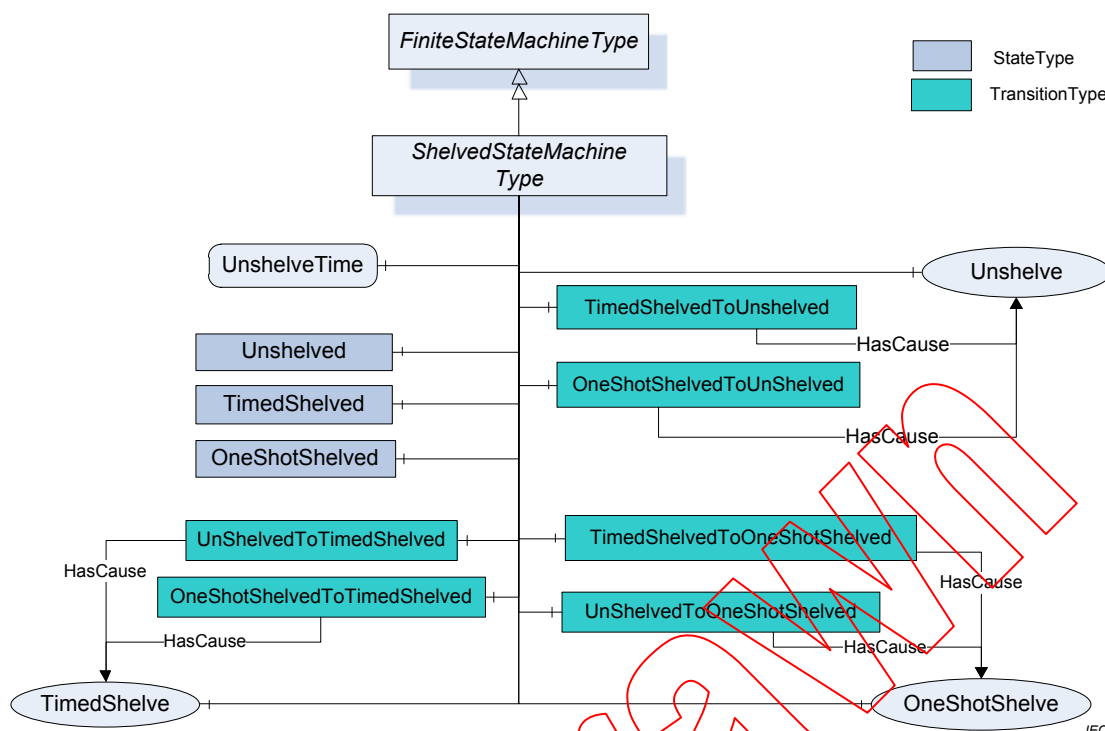


Figure 14 – Shelved State Machine Model

Table 31 –ShelvedStateMachine Definition

Attribute	Value				
BrowseName	ShelvedStateMachineType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the FiniteStateMachineType defined in IEC 62541-5					
HasProperty	Variable	UnshelveTime	Duration	PropertyType	Mandatory
HasComponent	Object	Unshelved		StateType	Mandatory
HasComponent	Object	TimedShelved		StateType	Mandatory
HasComponent	Object	OneShotShelved		StateType	Mandatory
HasComponent	Object	UnshelvedToTimedShelved		TransitionType	Mandatory
HasComponent	Object	TimedShelvedToUnshelved		TransitionType	Mandatory
HasComponent	Object	TimedShelvedToOneShotShelved		TransitionType	Mandatory
HasComponent	Object	UnshelvedToOneShotShelved		TransitionType	Mandatory
HasComponent	Object	OneShotShelvedToUnshelved		TransitionType	Mandatory
HasComponent	Object	OneShotShelvedToTimedShelved		TransitionType	Mandatory
HasComponent	Method	TimedShelve	Defined in 5.8.3.3		Mandatory
HasComponent	Method	OneShotShelve	Defined in 5.8.3.4		Mandatory
HasComponent	Method	Unshelve	Defined in 5.8.3.2		Mandatory

UnshelveTime specifies the remaining time in milliseconds until the *Alarm* automatically transitions into the *Un-shelved* state. For the *TimedShelved* state this time is initialised with the *ShelvingTime* argument of the *TimedShelve Method* call. For the *OneShotShelved* state the *UnshelveTime* will be a constant set to the maximum *Duration* except if a *MaxTimeShelved* Property is provided.

This *FiniteStateMachine* supports three *Active* states; *Unshelved*, *TimedShelved* and *OneShotShelved*. It also supports six transitions. The states and transitions are described in Table 32. This *FiniteStateMachine* also supports three *Methods*; *TimedShelve*, *OneShotShelve* and *Unshelve*.

Table 32 – ShelvedStateMachine Transitions

BrowseName	References	BrowseName	TypeDefinition
Transitions			
UnshelvedToTimedShelved	FromState	Unshelved	StateType
	ToState	TimedShelved	StateType
	HasEffect	AlarmConditionType	
	HasCause	TimedShelve	Method
UnshelvedToOneShotShelved	FromState	Unshelved	StateType
	ToState	OneShotShelved	StateType
	HasEffect	AlarmConditionType	
	HasCause	OneShotShelve	Method
TimedShelvedToUnshelved	FromState	TimedShelved	StateType
	ToState	Unshelved	StateType
	HasEffect	AlarmConditionType	
TimedShelvedToOneShotShelved	FromState	TimedShelved	StateType
	ToState	OneShotShelved	StateType
	HasEffect	AlarmConditionType	
	HasCause	OneShotShelving	Method
OneShotShelvedToUnshelved	FromState	OneShotShelved	StateType
	ToState	Unshelved	StateType
	HasEffect	AlarmConditionType	
OneShotShelvedToTimedShelved	FromState	OneShotShelved	StateType
	ToState	TimedShelved	StateType
	HasEffect	AlarmConditionType	
	HasCause	TimedShelve	Method

5.8.3.2 Unshelve Method

Unshelve sets the *AlarmCondition* to the Unshelved state

Signature

```
Unshelve ( );
```

Method Result Codes in Table 33 (defined in Call Service).

Table 33 – Unshelve Result Codes

ResultCode	Description
Bad_ConditionNotShelved	See Table 70 for the description of this result code.

Table 34 specifies the *AddressSpace* representation for the *Unshelve Method*.

Table 34 – Unshelve Method AddressSpace Definition

Attribute	Value				
BrowseName	Unshelve				
References	NodeClass	BrowseName	Data Type	TypeDefinition	ModellingRule
AlwaysGeneratesEvent	Defined in 5.10.7			AuditConditionShelvingEventType	

5.8.3.3 TimedShelve Method

TimedShelve sets the *AlarmCondition* to the TimedShelved state (Parameters are defined in Table 35 and result code are described in Table 36).

Signature

```
TimedShelve (
    [in] Duration ShelvingTime
);
```

Table 35 – TimedShelve Parameters

Argument	Description
ShelvingTime	Specifies a fixed time for which the <i>Alarm</i> is to be shelved. The <i>Server</i> may refuse the provided duration. If a <i>MaxTimeShelved</i> Property exist on the <i>Alarm</i> than the <i>Shelving</i> time shall be less than or equal to the value of this Property.

Method Result Codes (defined in Call Service).

Table 36 – TimedShelve Result Codes

ResultCode	Description
Bad_ConditionAlreadyShelved	See Table 70 for the description of this result code. The <i>Alarm</i> is already in TimedShelved state and the system does not allow a reset of the shelved timer.
Bad_ShelvingTimeOutOfRange	See Table 70 for the description of this result code.

Comments

Shelving for some time is quite often used to block nuisance *Alarms*. For example, an *Alarm* that occurs more than 10 times in a minute may get shelved for a few minutes.

In some systems the length of time covered by this duration may be limited and the *Server* may generate an error refusing the provided duration. This limit may be exposed as the *MaxTimeShelved* Property.

Table 37 specifies the *AddressSpace* representation for the *TimedShelve* Method.

Table 37 – TimedShelve Method AddressSpace Definition

Attribute	Value				
BrowseName	TimedShelve				
References	NodeClass	BrowseName	Data Type	TypeDefinition	ModellingRule
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGenerates Event	Defined in 5.10.7			AuditConditionShelvingEventType	

5.8.3.4 OneShotShelve Method

OneShotShelve sets the *AlarmCondition* to the OneShotShelved state.

Signature

```
OneShotShelve( );
```

Method Result Codes, in Table 38 (defined in Call Service).

Table 38 – OneShotShelve Result Codes

ResultCode	Description
Bad_AlreadyShelved	See Table 70 for the description of this result code. The <i>Alarm</i> is already in OneShotShelved state.

Table 39 specifies the *AddressSpace* representation for the *OneShotShelve* Method.

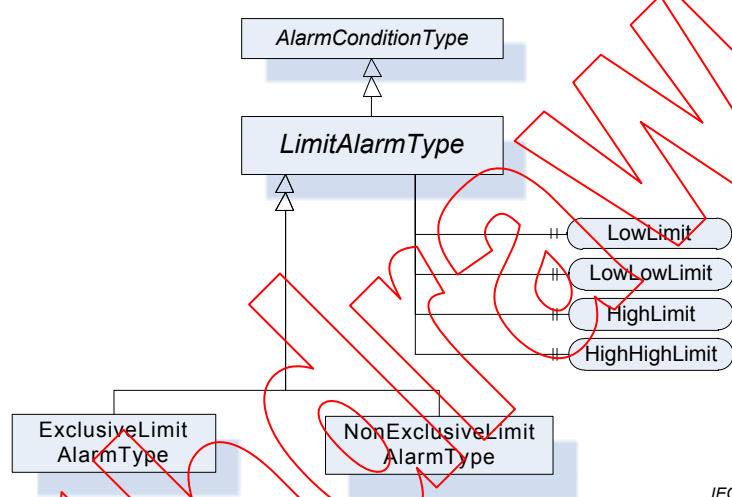
Table 39 – OneShotShelve Method AddressSpace Definition

Attribute	Value				
BrowseName	OneShotShelve				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
AlwaysGeneratesEvent	Defined in 5.10.7			AuditConditionShelvingEventType	

5.8.4 LimitAlarmType

Alarms can be modelled with multiple exclusive sub-states and assigned limits or they may be modelled with non exclusive limits that can be used to group multiple states together.

The *LimitAlarmType* is an abstract type used to provide a base *Type* for *AlarmConditions* with multiple limits. The *LimitAlarmType* is illustrated in Figure 15.

**Figure 15 – LimitAlarmType**

The *LimitAlarmType* is formally defined in Table 40.

Table 40 – LimitAlarmType Definition

Attribute	Value				
BrowseName	LimitAlarmType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the AlarmConditionType defined in 5.8.2.					
HasSubtype	ObjectType	ExclusiveLimitAlarmType	Defined in 5.8.5.3		
HasSubtype	ObjectType	NonExclusiveLimitAlarmType	Defined in 5.8.6		
HasProperty	Variable	HighHighLimit	Double	PropertyType	Optional
HasProperty	Variable	HighLimit	Double	PropertyType	Optional
HasProperty	Variable	LowLimit	Double	PropertyType	Optional
HasProperty	Variable	LowLowLimit	Double	PropertyType	Optional

Four optional limits are defined that configure the states of the derived limit *Alarm* Types. These *Properties* shall be set for any *Alarm* limits that are exposed by the derived limit *Alarm* Types. These *Properties* are listed as optional but at least one is required. For cases where an underlying system cannot provide the actual value of a limit, the limit *Property* shall still be provided, but will have its *AccessLevel* set to not readable.

The *Alarm* limits listed may cause an *Alarm* to be generate when a value equals the limit or it may generate the *Alarm* when the limit is exceeded, (i.e. the Value is above the limit for

HighLimit and below the limit for LowLimit). The exact behaviour when the value is equal to the limit is *Server* specific.

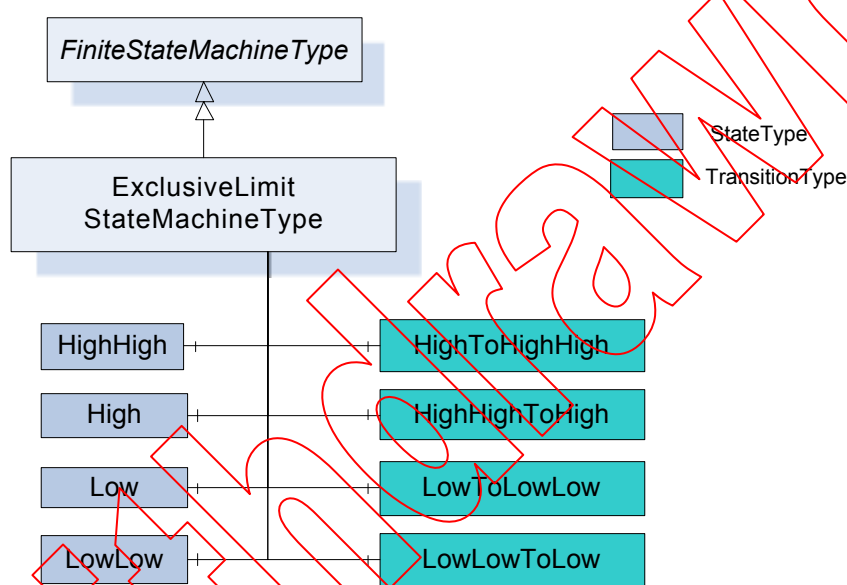
5.8.5 ExclusiveLimit Types

5.8.5.1 Overview

This Clause describes the state machine and the base *Alarm* Type behaviour for *Alarm* Types with multiple mutually exclusive limits.

5.8.5.2 ExclusiveLimitStateMachineType

The *ExclusiveLimitStateMachineType* defines the state machine used by *AlarmTypes* that handle multiple mutually exclusive limits. It is illustrated in Figure 16.



IEC

Figure 16 – ExclusiveLimitStateMachine

It is created by extending the *FiniteStateMachineType*. It is formally defined in Table 41 and the state transitions are described in Table 42.

Table 41 – ExclusiveLimitStateMachineType Definition

Attribute	Value				
BrowseName	ExclusiveLimitStateMachineType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
Subtype of the <i>FiniteStateMachineType</i>					
HasComponent	Object	HighHigh		StateType	Optional
HasComponent	Object	High		StateType	Optional
HasComponent	Object	Low		StateType	Optional
HasComponent	Object	LowLow		StateType	Optional
HasComponent	Object	LowToLowLow		TransitionType	Optional
HasComponent	Object	LowLowToLow		TransitionType	Optional
HasComponent	Object	HighToHighHigh		TransitionType	Optional
HasComponent	Object	HighHighToHigh		TransitionType	Optional

Table 42 – ExclusiveLimitStateMachineType Transitions

BrowseName	References	BrowseName	TypeDefinition
Transitions			
HighHighToHigh	FromState	HighHigh	StateType
	ToState	High	StateType
	HasEffect	AlarmConditionType	
HighToHighHigh	FromState	High	StateType
	ToState	HighHigh	StateType
	HasEffect	AlarmConditionType	
LowLowToLow	FromState	LowLow	StateType
	ToState	Low	StateType
	HasEffect	AlarmConditionType	
LowToLowLow	FromState	Low	StateType
	ToState	LowLow	StateType
	HasEffect	AlarmConditionType	

The *ExclusiveLimitStateMachine* defines the sub state machine that represents the actual level of a multilevel *Alarm* when it is in the *Active* state. The sub state machine defined here includes High, Low, HighHigh and LowLow states. This model also includes in its transition state a series of transition to and from a parent state, the inactive state. This state machine as it is defined shall be used as a sub state machine for a state machine which has an *Active* state. This *Active* state could be part of a “level” *Alarm* or “deviation” *Alarm* or any other *Alarm* state machine.

The LowLow, Low, High, HighHigh are typical for many industries. Vendors can introduce sub-state models that include additional limits; they may also omit limits in an instance.

5.8.5.3 ExclusiveLimitAlarmType

The *ExclusiveLimitAlarmType* is used to specify the common behaviour for *Alarm Types* with multiple mutually exclusive limits. The *ExclusiveLimitAlarmType* is illustrated in Figure 17.

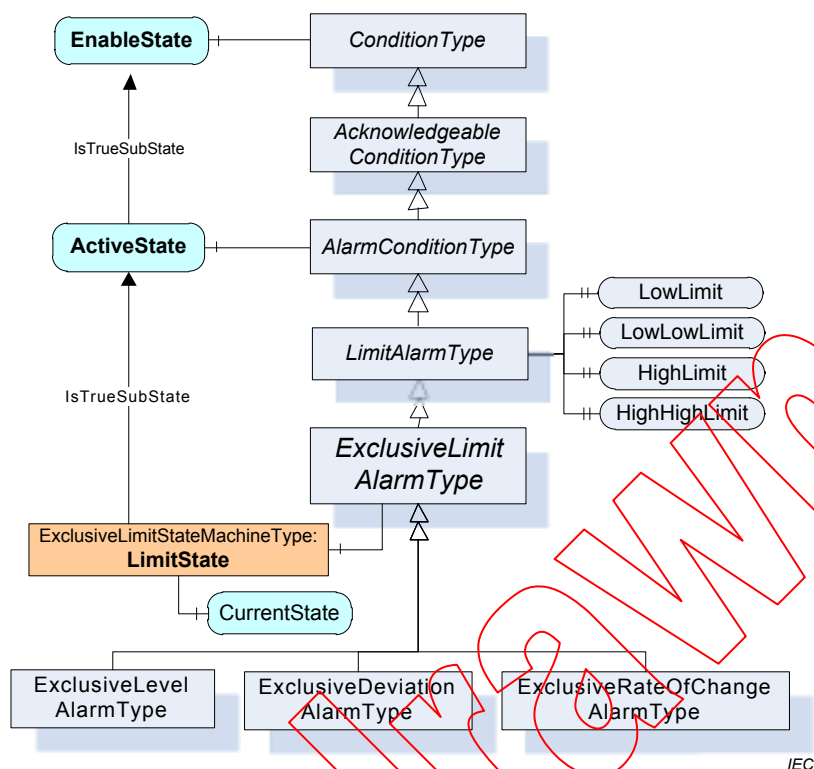


Figure 17 – ExclusiveLimitAlarmType

The *ExclusiveLimitAlarmType* is formally defined in Table 43.

Table 43 – ExclusiveLimitAlarmType Definition

Attribute	Value				
BrowseName	ExclusiveLimitAlarmType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	Modelling Rule
Subtype of the <i>LimitAlarmType</i> defined in 5.8.4.					
HasSubtype	ObjectType	ExclusiveLevelAlarmType	Defined in 5.8.7.3		
HasSubtype	ObjectType	ExclusiveDeviationAlarmType	Defined in 5.8.8.3		
HasSubtype	ObjectType	ExclusiveRateOfChangeAlarmType	Defined in 5.8.9.3		
HasComponent	Object	LimitState		<i>ExclusiveLimitStateMachineType</i>	Mandatory

The *LimitState* is a Substate of the *ActiveState* and has a *IsTrueSubstate* reference to the *ActiveState*. The *LimitState* represents the actual limit that is violated in an *ExclusiveLimitAlarm*. When the *ActiveState* of the *AlarmConditionType* is inactive the *LimitState* shall not be available and shall return NULL on read. Any *Events* that subscribe for fields from the *LimitState* when the *ActiveState* is inactive shall return a NULL for these unavailable fields

5.8.6 NonExclusiveLimitAlarmType

The *NonExclusiveLimitAlarmType* is used to specify the common behaviour for *AlarmTypes* with multiple non-exclusive limits. The *NonExclusiveLimitAlarmType* is illustrated in Figure 18.

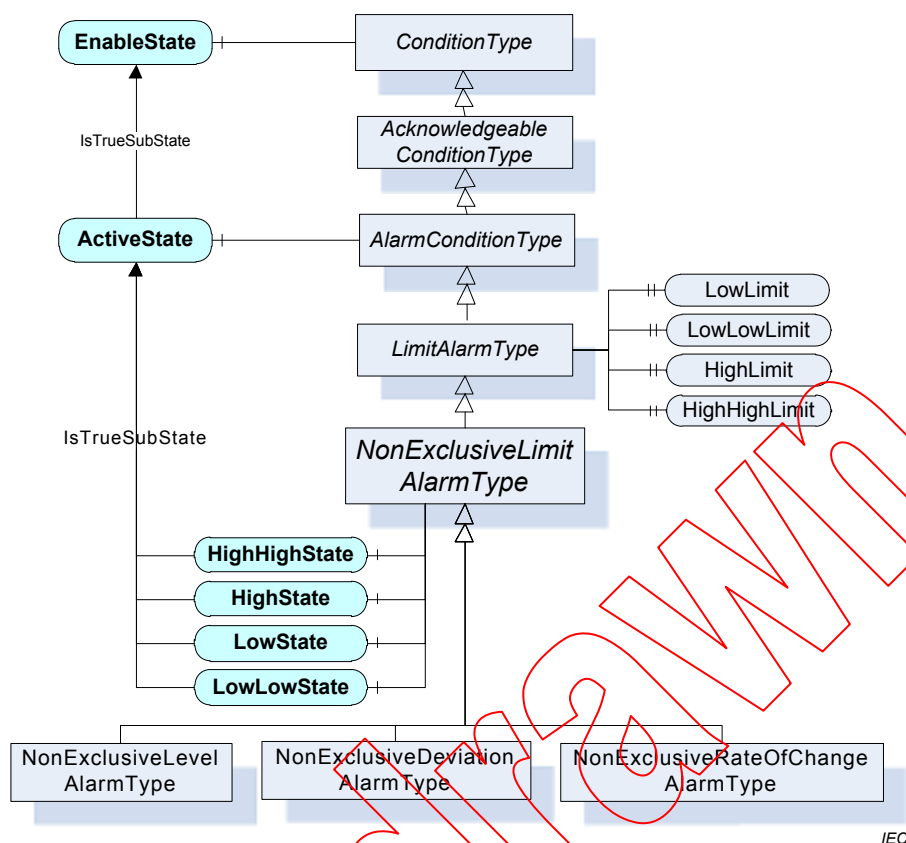


Figure 18 – NonExclusiveLimitAlarmType

The *NonExclusiveLimitAlarmType* is formally defined in Table 44.

Table 44 – NonExclusiveLimitAlarmType Definition

Attribute	Value				
BrowseName	NonExclusiveLimitAlarmType				
IsAbstract	False				
References	NodeClass	BrowseName	DataType	TypeDefinition	Modelling Rule
Subtype of the LimitAlarmType defined in 5.8.4.					
HasSubtype	ObjectType	NonExclusiveLevelAlarmType	Defined in 5.8.7.2		
HasSubtype	ObjectType	NonExclusiveDeviationAlarmType	Defined in 5.8.8.2		
HasSubtype	ObjectType	NonExclusiveRateOfChangeAlarmType	Defined in 5.8.9.2		
HasComponent	Variable	HighHighState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Variable	HighState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Variable	LowState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Variable	LowLowState	LocalizedText	TwoStateVariableType	Optional

HighHighState, *HighState*, *LowState*, and *LowLowState* represent the non-exclusive states. As an example, it is possible that both *HighState* and *HighHighState* are in their TRUE state. Vendors may choose to support any subset of these states. Recommended state names are described in Annex A.

Four optional limits are defined that configure these states. At least the *HighState* or the *LowState* shall be provided even though all states are optional. It is implied by the definition of a *HighState* and a *LowState*, that these groupings are mutually exclusive. A value cannot exceed both a *HighState* value and a *LowState* value simultaneously.

5.8.7 Level Alarm

5.8.7.1 Overview

A level *Alarm* is commonly used to report when a limit is exceeded. It typically relates to an instrument – e.g. a temperature meter. The level *Alarm* becomes active when the observed value is above a high limit or below a low limit.

5.8.7.2 NonExclusiveLevelAlarmType

The *NonExclusiveLevelAlarmType* is a special level *Alarm* utilized with one or more non-exclusive states. If for example both the High and HighHigh states need to be maintained as active at the same time this *AlarmType* should be used.

The *NonExclusiveLevelAlarmType* is based on the *NonExclusiveLimitAlarmType*. It is formally defined in Table 45.

Table 45 – NonExclusiveLevelAlarmType Definition

Attribute	Value				
BrowseName	NonExclusiveLevelAlarmType				
IsAbstract	False				
References	NodeClass	BrowseName	Data Type	TypeDefinition	ModellingRule
Subtype of the NonExclusiveLimitAlarmType defined in 5.8.6.					

No additional *Properties* to the *NonExclusiveLimitAlarmType* are defined.

5.8.7.3 ExclusiveLevelAlarmType

The *ExclusiveLevelAlarmType* is a special level *Alarm* utilized with multiple mutually exclusive limits. It is formally defined in Table 46.

Table 46 – ExclusiveLevelAlarmType Definition

Attribute	Value				
BrowseName	ExclusiveLevelAlarmType				
IsAbstract	False				
References	NodeClass	BrowseName	Data Type	Type Definition	Modelling Rule
Inherits the Properties of the ExclusiveLimitAlarmType defined in 5.8.5.3.					

No additional *Properties* to the *ExclusiveLimitAlarmType* are defined.

5.8.8 Deviation Alarm

5.8.8.1 Overview

A deviation *Alarm* is commonly used to report an excess deviation between a desired set point level of a process value and an actual measurement of that value. The deviation *Alarm* becomes active when the deviation exceeds or drops below a defined limit.

For example if a set point had a value of 10 and the high deviation *Alarm* limit were set for 2 and the low deviation *Alarm* limit had a value of -1 then the low sub state is entered if the process value dropped to below 9; the high sub state is entered if the process value became larger than 12. If the set point were changed to 11 then the new deviation values would be 10 and 13 respectively.