

BS EN 62541-9:2012



BSI Standards Publication

OPC unified architecture

Part 9: Alarms and conditions

NO COPYING WITHOUT BSI PERMISSION EXCEPT AS PERMITTED BY COPYRIGHT LAW

raising standards worldwide™

Full document details: BS EN 62541-9:2012



National foreword

This British Standard is the UK implementation of EN 62541-9:2012. It is identical to IEC 62541-9:2012.

The UK participation in its preparation was entrusted to Technical Committee AMT/7, Industrial communications: process measurement and control, including fieldbus.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© The British Standards Institution 2012

Published by BSI Standards Limited 2012

ISBN 978 0 580 65225 7

ICS 25.040.40; 25.100.01

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 October 2012.

Amendments issued since publication

Amd. No.	Date	Text affected
----------	------	---------------

EUROPEAN STANDARD
NORME EUROPÉENNE
EUROPÄISCHE NORM

EN 62541-9

September 2012

ICS 25.040.40; 25.100.01

English version

**OPC unified architecture -
Part 9: Alarms and conditions
(IEC 62541-9:2012)**

Architecture unifiée OPC -
Partie 9: Alarmes et conditions
(CEI 62541-9:2012)

OPC Unified Architecture -
Teil 9: Alarme und Zustände
(IEC 62541-9:2012)

This European Standard was approved by CENELEC on 2012-08-31. CENELEC members are bound to comply with the CEN/CENELEC Internal Regulations which stipulate the conditions for giving this European Standard the status of a national standard without any alteration.

Up-to-date lists and bibliographical references concerning such national standards may be obtained on application to the CEN-CENELEC Management Centre or to any CENELEC member.

This European Standard exists in three official versions (English, French, German). A version in any other language made by translation under the responsibility of a CENELEC member into its own language and notified to the CEN-CENELEC Management Centre has the same status as the official versions.

CENELEC members are the national electrotechnical committees of Austria, Belgium, Bulgaria, Croatia, Cyprus, the Czech Republic, Denmark, Estonia, Finland, Former Yugoslav Republic of Macedonia, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, the Netherlands, Norway, Poland, Portugal, Romania, Slovakia, Slovenia, Spain, Sweden, Switzerland, Turkey and the United Kingdom.

CENELEC

European Committee for Electrotechnical Standardization
Comité Européen de Normalisation Electrotechnique
Europäisches Komitee für Elektrotechnische Normung

Management Centre: Avenue Marnix 17, B - 1000 Brussels

© 2012 CENELEC - All rights of exploitation in any form and by any means reserved worldwide for CENELEC members.

Ref. No. EN 62541-9:2012 E

AVA Documents
Document Sharing Hub

Foreword

The text of document 65E/243/FDIS, future edition 1 of IEC 62541-9, prepared by SC 65E "Devices and integration in enterprise systems" of IEC/TC 65 "Industrial-process measurement, control and automation" was submitted to the IEC-CENELEC parallel vote and approved by CENELEC as EN 62541-9:2012.

The following dates are fixed:

- latest date by which the document has to be implemented at national level by publication of an identical national standard or by endorsement (dop) 2013-05-31
- latest date by which the national standards conflicting with the document have to be withdrawn (dow) 2015-08-31

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. CENELEC [and/or CEN] shall not be held responsible for identifying any or all such patent rights.

Endorsement notice

The text of the International Standard IEC 62541-9:2012 was approved by CENELEC as a European Standard without any modification.

In the official version, for Bibliography, the following note has to be added for the standard indicated :

IEC 62541-7 NOTE Harmonised as EN 62541-7.

Annex ZA (normative)

Normative references to international publications with their corresponding European publications

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.

NOTE When an international publication has been modified by common modifications, indicated by (mod), the relevant EN/HD applies.

<u>Publication</u>	<u>Year</u>	<u>Title</u>	<u>EN/HD</u>	<u>Year</u>
IEC/TR 62541-1	-	OPC unified architecture - Part 1: Overview and concepts	CLC/TR 62541-1	-
IEC 62541-3	-	OPC unified architecture - Part 3: Address space model	EN 62541-3	-
IEC 62541-4	-	OPC unified architecture - Part 4: Services	EN 62541-4	-
IEC 62541-5	-	OPC unified architecture - Part 5: Information model	EN 62541-5	-
IEC 62541-6	-	OPC unified architecture - Part 6: Mappings	EN 62541-6	-
IEC 62541-8	-	OPC unified architecture - Part 8: Data access	EN 62541-8	-
EEMUA 191	2007	Alarm systems - A guide to design, management and procurement	-	-

CONTENTS

INTRODUCTION	9
1 Scope	10
2 Normative references	10
3 Terms, definitions, abbreviations and data types	10
3.1 Terms and definitions	10
3.2 Abbreviations	12
3.3 Used data types	12
4 Concepts	12
4.1 General	12
4.2 Conditions	13
4.3 Acknowledgeable Conditions	14
4.4 Previous States of Conditions	16
4.5 Condition State Synchronization	16
4.6 Severity, Quality and Comment	17
4.7 Dialogs	17
4.8 Alarms	17
4.9 Multiple Active States	18
4.10 Condition Instances in the Address Space	19
4.11 Alarm and Condition Auditing	20
5 Model	20
5.1 General	20
5.2 Two-State State Machines	21
5.3 Condition Variables	23
5.4 Substate Reference Types	23
5.4.1 General	23
5.4.2 HasTrueSubState ReferenceType	23
5.4.3 HasFalseSubState ReferenceType	24
5.5 Condition Model	24
5.5.1 General	24
5.5.2 ConditionType	25
5.5.3 Condition and Branch Instances	28
5.5.4 Disable Method	28
5.5.5 Enable Method	29
5.5.6 AddComment Method	29
5.5.7 ConditionRefresh Method	30
5.6 Dialog Model	32
5.6.1 General	32
5.6.2 DialogConditionType	32
5.6.3 Respond Method	34
5.7 Acknowledgeable Condition Model	34
5.7.1 General	34
5.7.2 AcknowledgeableConditionType	34
5.7.3 Acknowledge Method	36
5.7.4 Confirm Method	37

5.8	Alarm Model	38
5.8.1	General	38
5.8.2	AlarmConditionType	38
5.8.3	ShelvedStateMachineType	40
5.8.4	Unshelve Method	43
5.8.5	TimedShelve Method	44
5.8.6	OneShotShelve Method	44
5.8.7	LimitAlarmType	45
5.8.8	ExclusiveLimit Types	46
5.8.9	NonExclusiveLimitAlarmType	49
5.8.10	Level Alarm	51
5.8.11	Deviation Alarm	52
5.8.12	Rate of Change	53
5.8.13	Discrete Alarms	54
5.9	ConditionClasses	56
5.9.1	Overview	56
5.9.2	Base ConditionClassType	56
5.9.3	ProcessConditionClassType	56
5.9.4	MaintenanceConditionClassType	57
5.9.5	SystemConditionClassType	57
5.10	Audit Events	57
5.10.1	Overview	57
5.10.2	AuditConditionEventType	58
5.10.3	AuditConditionEnableEventType	58
5.10.4	AuditConditionCommentEventType	59
5.10.5	AuditConditionRespondEventType	59
5.10.6	AuditConditionAcknowledgeEventType	59
5.10.7	AuditConditionConfirmEventType	59
5.10.8	AuditConditionShelvingEventType	59
5.11	Condition Refresh Related Events	60
5.11.1	Overview	60
5.11.2	RefreshStartEventType	60
5.11.3	RefreshEndEventType	60
5.11.4	RefreshRequiredEventType	61
5.12	HasCondition Reference Type	61
5.13	Alarm and Condition Status Codes	62
6	AddressSpace Organisation	62
6.1	General	62
6.2	Event Notifier and Source Hierarchy	62
6.3	Adding Conditions to the Hierarchy	63
6.4	Conditions in InstanceDeclarations	64
6.5	Conditions in a VariableType	65
Annex A (informative)	Recommended Localized Names	66
Annex B (informative)	Examples	69
Annex C (informative)	Mapping to EEMUA	74
Annex D (informative)	Mapping from OPC A&E to OPC UA A&C	75
Bibliography		89

Figure 1 – Base Condition State Model	14
Figure 2 – AcknowledgeableConditions State Model	14
Figure 3 – Acknowledge State Model	15
Figure 4 – Confirmed Acknowledge State Model	16
Figure 5 – Alarm State Machine Model.....	18
Figure 6 – Multiple Active States Example	19
Figure 7 – ConditionType Hierarchy	21
Figure 8 – Condition Model	25
Figure 9 – DialogConditionType Overview.....	32
Figure 10 – AcknowledgeableConditionType Overview	35
Figure 11 – AlarmConditionType Hierarchy Model.....	38
Figure 12 – Alarm Model	39
Figure 13 – Shelve state transitions	41
Figure 14 – Shelved State Machine Model	41
Figure 15 – LimitAlarmType	45
Figure 16 – ExclusiveLimitStateMachine	47
Figure 17 – ExclusiveLimitAlarmType	49
Figure 18 – NonExclusiveLimitAlarmType	50
Figure 19 – DiscreteAlarmType Hierarchy.....	54
Figure 20 – ConditionClass Type Hierarchy	56
Figure 21 – AuditEvent Hierarchy.....	58
Figure 22 – Refresh Related Event Hierarchy	60
Figure 23 – Typical Event Hierarchy	63
Figure 24 – Use of HasCondition in an Event Hierarchy	64
Figure 25 – Use of HasCondition in an InstanceDeclaration	65
Figure 26 – Use of HasCondition in a VariableType	65
Figure B.1 – Single State Example.....	69
Figure B.2 – Previous State Example.....	70
Figure B.3 – HasCondition used with Condition instances	72
Figure B.4 – HasCondition reference to a Condition Type	73
Figure B.5 – HasCondition used with an instance declaration	73
Figure D.1 – The Type Model of a Wrapped COM AE Server	77
Figure D.2 – Mapping UA Event Types to COM A&E Event Types.....	81
Figure D.3 – Example Mapping of UA Event Types to COM A&E Categories.....	82
Figure D.4 – Example Mapping of UA Event Types to A&E Categories with Attributes	85
Table 1 – Parameter Types defined in IEC 62541-3	12
Table 2 – Parameter Types defined in IEC 62541-4	12
Table 3 – TwoStateVariableType Definition.....	22
Table 4 – ConditionVariableType Definition.....	23
Table 5 – HasTrueSubState ReferenceType	24
Table 6 – HasFalseSubState ReferenceType	24
Table 7 – ConditionType Definition	26

Table 8 – SimpleAttributeOperand to select ConditionId.....	28
Table 9 – Disable Method AddressSpace Definition	29
Table 10 – Enable Method AddressSpace Definition	29
Table 11 – AddComment Method AddressSpace Definition	30
Table 12 – ConditionRefresh Method AddressSpace Definition	32
Table 13 – DialogConditionType Definition.....	33
Table 14 – Respond Method AddressSpace Definition	34
Table 15 – AcknowledgeableConditionType Definition	35
Table 16 – Acknowledge Method AddressSpace Definition	37
Table 17 – Confirm Method Parameters	37
Table 18 – Confirm Method AddressSpace Definition.....	38
Table 19 – AlarmConditionType Definition	39
Table 20 – ShelvedStateMachine Definition	42
Table 21 – ShelvedStateMachine Transitions.....	43
Table 22 – Unshelve Method AddressSpace Definition	44
Table 23 – TimedShelve Method AddressSpace Definition.....	44
Table 24 – OneShotShelve Method AddressSpace Definition.....	45
Table 25 – LimitAlarmType Definition.....	46
Table 26 – ExclusiveLimitStateMachineType Definition	47
Table 27 – ExclusiveLimitStateMachineType Transitions	48
Table 28 – ExclusiveLimitAlarmType Definition	49
Table 29 – NonExclusiveLimitAlarmType Definition	51
Table 30 – NonExclusiveLevelAlarmType Definition	52
Table 31 – ExclusiveLevelAlarmType Definition	52
Table 32 – NonExclusiveDeviationAlarmType Definition.....	53
Table 33 – ExclusiveDeviationAlarmType Definition	53
Table 34 – NonExclusiveRateOfChangeAlarmType Definition	54
Table 35 – ExclusiveRateOfChangeAlarmType Definition	54
Table 36 – DiscreteAlarmType Definition	55
Table 37 – OffNormalAlarmType Definition	55
Table 38 – TripAlarmType Definition	55
Table 39 – BaseConditionClassType Definition	56
Table 40 – ProcessConditionClassType Definition	57
Table 41 – MaintenanceConditionClassType Definition	57
Table 42 – SystemConditionClassType Definition	57
Table 43 – AuditConditionEventType Definition.....	58
Table 44 – AuditConditionEnableEventType Definition	58
Table 45 – AuditConditionCommentEventType Definition.....	59
Table 46 – AuditConditionRespondEventType Definition	59
Table 47 – AuditConditionAcknowledgeEventType Definition	59
Table 48 – AuditConditionConfirmEventType Definition	59
Table 49 – AuditConditionShelvingEventType Definition	59
Table 50 – RefreshStartEventType Definition.....	60

Table 51 – RefreshEndEventType Definition	60
Table 52 – RefreshRequiredEventType Definition	61
Table 53 – HasCondition ReferenceType	61
Table 54 – Alarm and Condition Result Codes	62
Table A.1 – Recommended state names for LocaleId “en”	66
Table A.2 – Recommended display names for LocaleId “en”	66
Table A.3 – Recommended state names for LocaleId “de”	67
Table A.4 – Recommended display names for LocaleId “de”	67
Table A.5 – Recommended state names for LocaleId “fr”	67
Table A.6 – Recommended display names for LocaleId “fr”	68
Table A.7 – Recommended Dialog Response Options.....	68
Table B.1 – Example of a Condition that only keeps the latest state.....	69
Table B.2 – Example of a <i>Condition</i> that maintains previous states via branches	71
Table C.1 – EEMUA Terms	74
Table D.1 – Mapping from ONEVENTSTRUCT fields to UA BaseEventType Variables.....	78
Table D.2 – Mapping from ONEVENTSTRUCT fields to UA AuditEventType Variables.....	78
Table D.3 – Mapping from ONEVENTSTRUCT fields to UA AlarmType Variables	79
Table D.4 – Event Category Attribute Mapping Table	83

INTRODUCTION

This International Standard is a specification intended for developers of OPC UA applications. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of applications by multiple vendors that inter-operate seamlessly together.

OPC UNIFIED ARCHITECTURE –

Part 9: Alarms and conditions

1 Scope

This part of the IEC 62541 series 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 191:2007, *Alarm Systems – A guide to design, management and procurement*, available at <<http://www.eemua.co.uk/>>

3 Terms, definitions, abbreviations and data types

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in IEC 62541-1, IEC 62541-3 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: As defined in EEMUA, the term “Accept” is another common term used to describe Acknowledge. They can be used interchangeably. This document will use Acknowledge.

3.1.2

Active

state for an *Alarm* that indicates that the situation the *Alarm* represents 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

a *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

a 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: As defined in EEMUA.

3.1.8

Operator

special user who is assigned to monitor and control a portion of the 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 specification an Operator is a special user. All descriptions that apply to general users also apply to Operators.

3.1.9

Refresh

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" as defined in EEMUA.

3.1.12

Suppress

logical criterion to determine that the *Alarm* does 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

A&C Alarms and Conditions

A&E Alarms and Events

DA Data Access

UA Unified Architecture

3.3 Used data types

The following tables describe the data types that are used throughout this document. These types are separated into two tables. Base data types defined in IEC 62541-3 are in Table 1. The base types and data types defined in IEC 62541-4 are 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 specification defines an *Information Model* for *Conditions*, *Dialog Conditions*, and *Alarms* including acknowledgement capabilities. It is built upon and extends base eventing 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.

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 subtypes 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 specification 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.

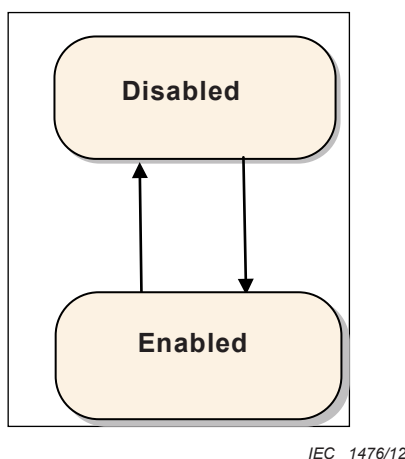


Figure 1 – Base Condition State Model

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*.

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.

4.3 Acknowledgeable Conditions

AcknowledgeableConditions are subtypes 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 Enabled state 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.

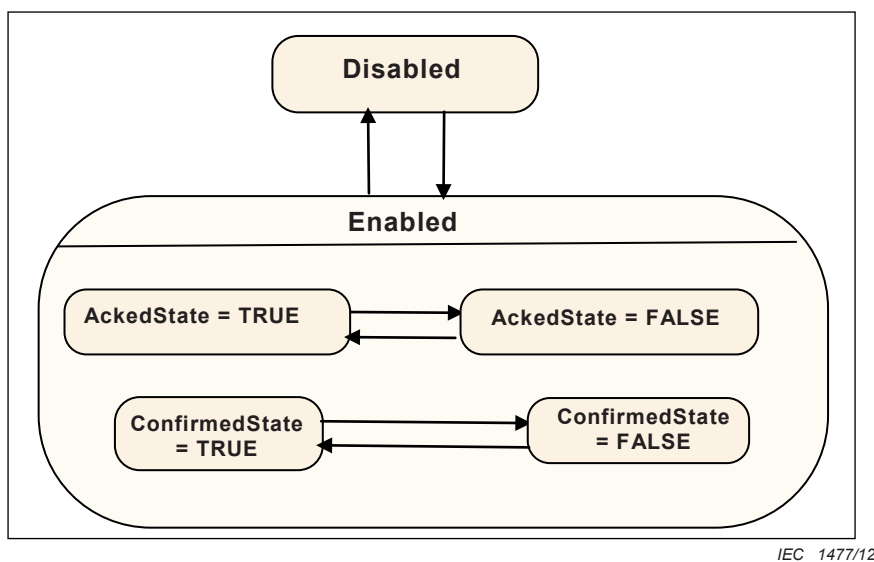
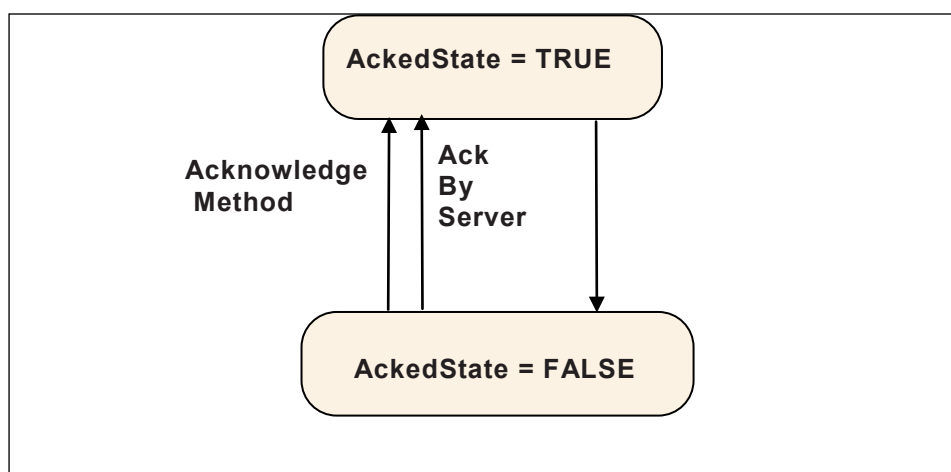


Figure 2 – AcknowledgeableConditions State Model

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 specification. 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 inactive.



IEC 1478/12

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.

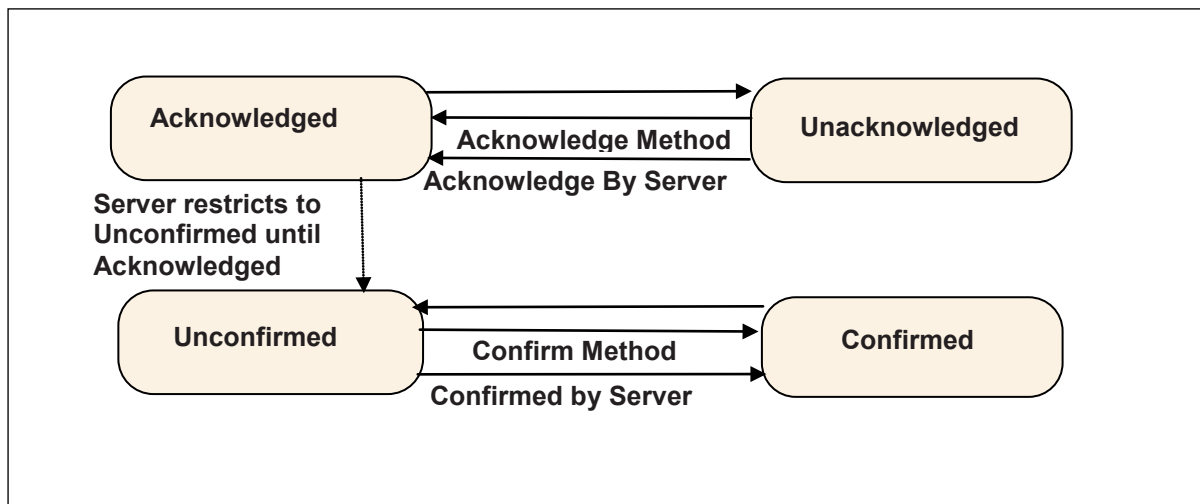


Figure 4 – Confirmed Acknowledge State Model

IEC 1479/12

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 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.

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* (*Refresh-EndEventType*, *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*.

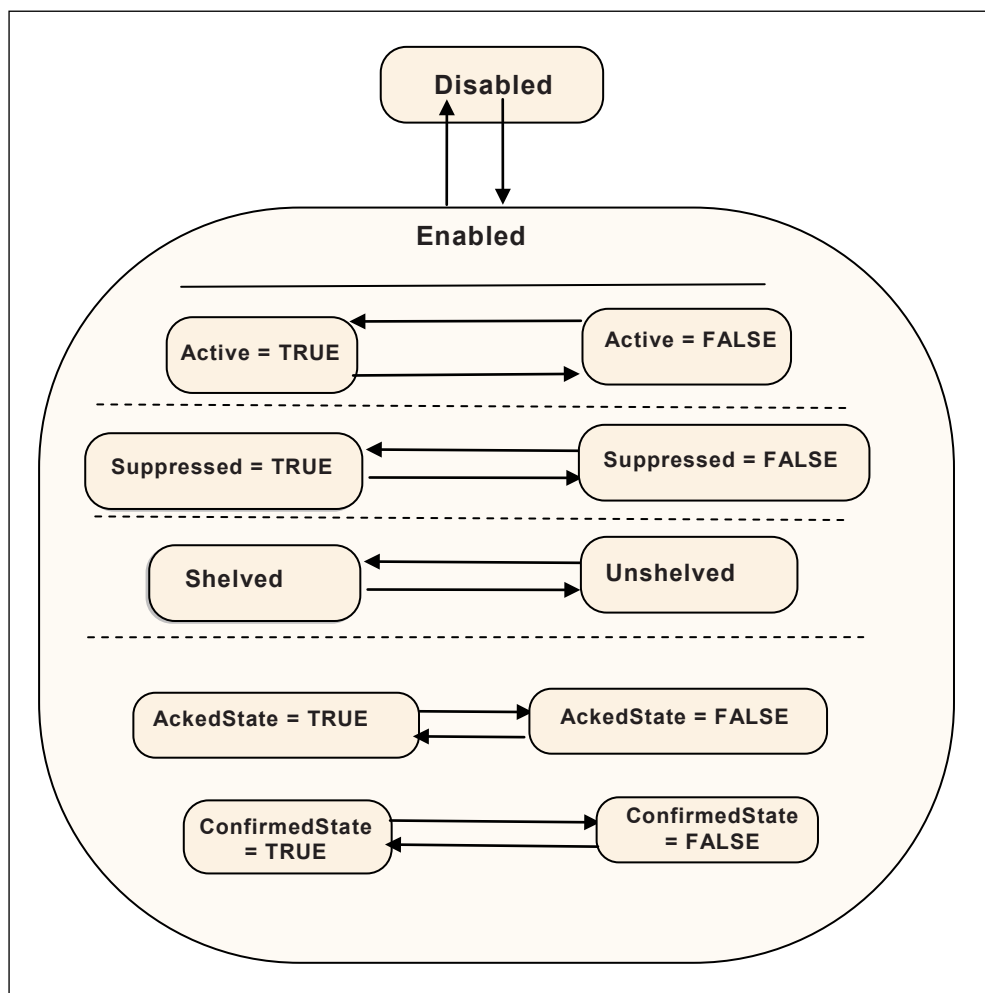
Quality refers to the quality of the data value(s) upon which this *Condition* is based. Since a *Condition* is usually based on one or more *Variables*, the *Condition* inherits the quality of these *Variables*. E.g., if the process value is "Uncertain", the "LevelAlarm" *Condition* is also questionable.

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.



IEC 1480/12

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 inactive, it is representing a situation that has returned to a normal state.

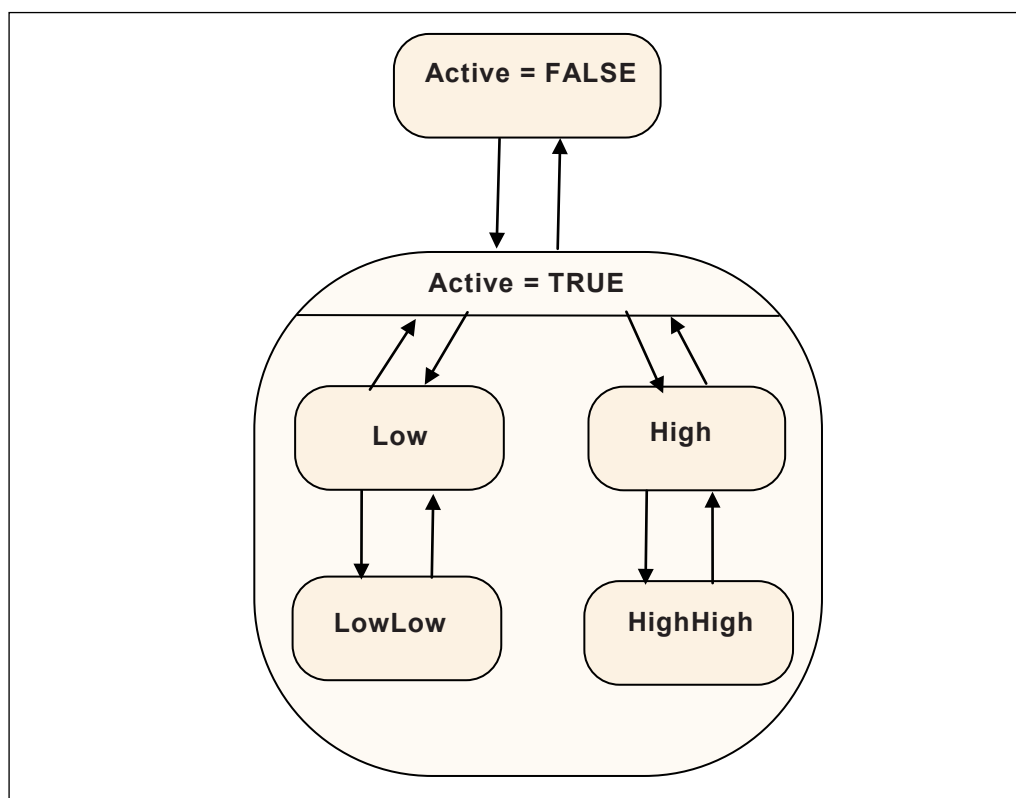
Some *Alarm* subtypes introduce sub-states of the active state. For example an *Alarm* representing a temperature may provide a high level state as well as a critically high state (see following clause).

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 Shelf 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 active may be in one of the following sub-states: LowLow, Low, High or HighHigh. The state model is illustrated in Figure 6.



IEC 1481/12

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 Specifications 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 specification defines a number of subtypes of the *AuditEventTypes* but without adding new fields to them.

This specification describes the *AuditEventType* that each *Method* is required to generate. For example, the *Disable Method* has an *AlwaysGeneratesEvent Reference* to an *AuditCondition-EnableEventType*. 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 specification 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.

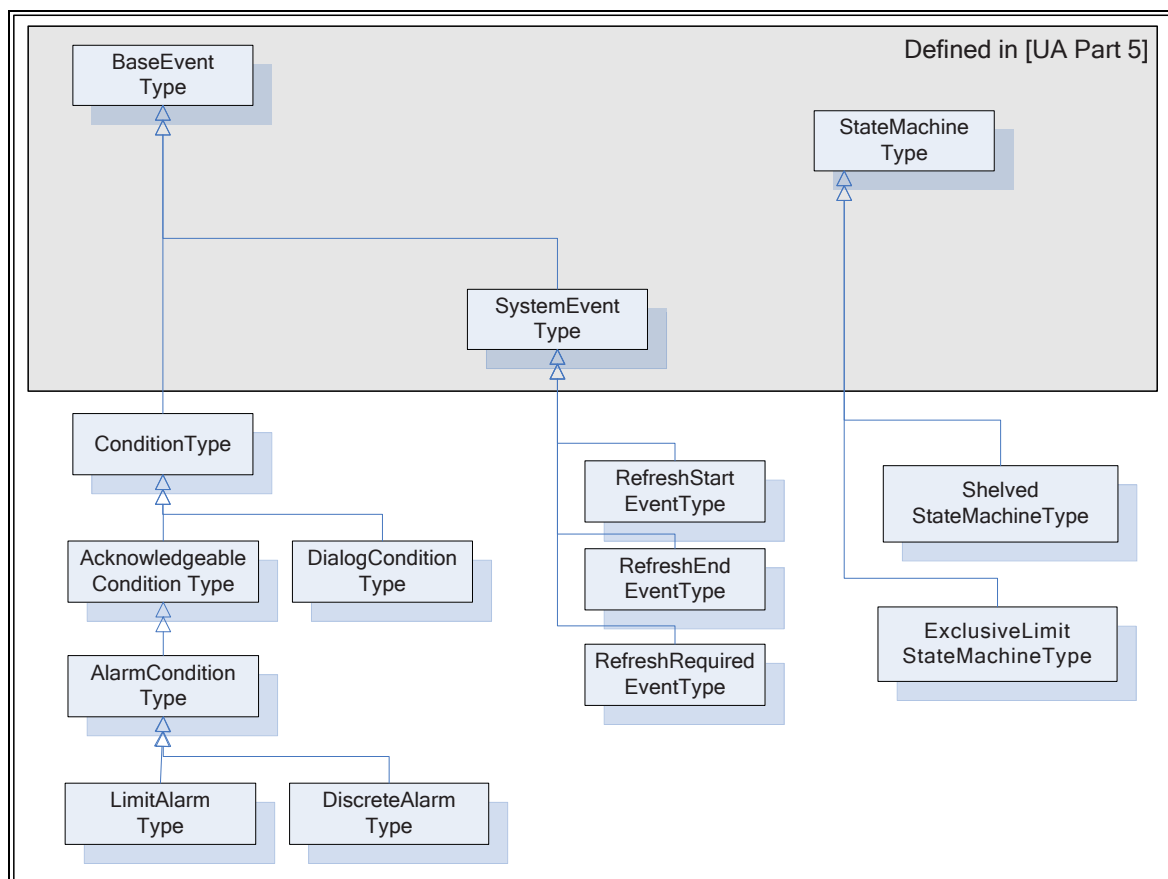


Figure 7 – ConditionType Hierarchy

IEC 1482/12

5.2 Two-State State Machines

Most states defined in this specification 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 substates 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 substate.

The optional *Property EffectiveDisplayName* from the *StateVariableType* is used if a state has substates. 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 *Two-StateVariables*.

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 substates.

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

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. 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 substate (the target *Node*) is a subordinate state of the TRUE super state. If more than one state within a *Condition* is a substate 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 substate, but it is required that the substate 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 substate (the target *Node*) is a subordinate state of the FALSE super state. If more than one state within a *Condition* is a substate 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 substate, but it is required that the substate provides the inverse *Reference* (*IsFalseSubStateOf*) to its super state.

Table 6 – HasFalseSubState ReferenceType

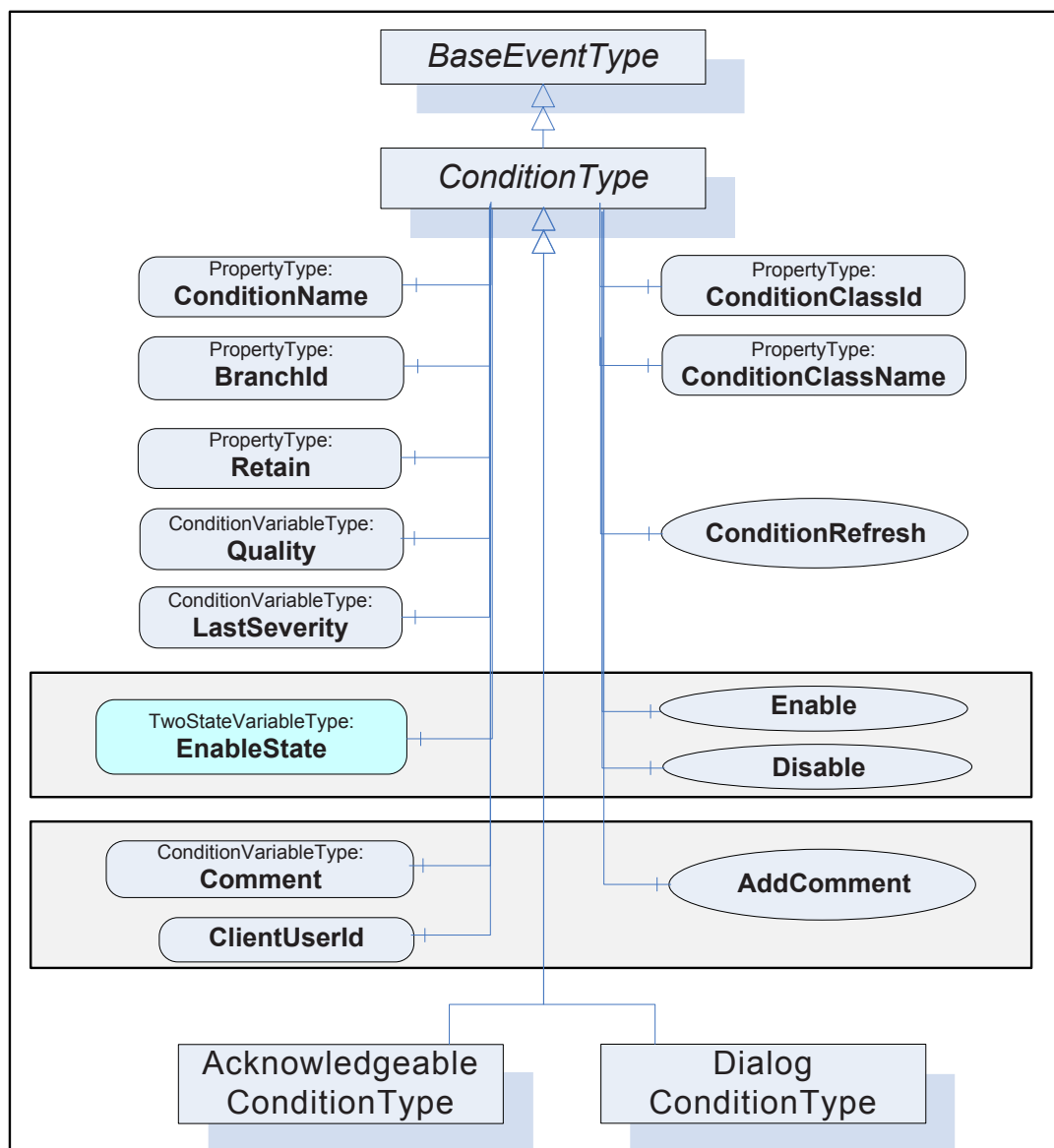
Attributes	Value		
BrowseName	HasFalseSubState		
InverseName	IsFalseSubStateOf		
Symmetric	False		
IsAbstract	False		
References	NodeClass	BrowseName	Comment

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 document, is not intended to be complete. Rather, the figures only illustrate information provided by the formal definitions.



IEC 1483/12

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 substate machine.

Table 7 – ConditionType Definition

Attribute	Value				
BrowseName	ConditionType				
IsAbstract	True				
References	NodeClass	BrowseName	Data Type	TypeDefinition	Modelling Rule
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	Localized Text	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	Localized Text	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.

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 specification. 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.

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 be 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. See 4.4 for more information about the need for creating and maintaining previous *ConditionBranches* and B.1 for an example using branches. Note that a *NodeId* *DataType* is used as the identifier 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* (a *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*, *EventType*, *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 list may not be exhaustive. SubTypes 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 (DataAccess) as well as *Good*, *Uncertain* and *Bad* as defined in IEC 62541-9.