

DIN EN 62541-9

ICS 35.240.50; 35.200

Einsprüche bis 2014-12-24
Vorgesehen als Ersatz für
DIN EN 62541-9:2013-06

Entwurf

**OPC Unified Architecture –
Teil 9: Alarme und Zustände
(IEC 65E/382/CDV:2014);
Englische Fassung FprEN 62541-9:2014**

OPC Unified Architecture –
Part 9: Alarms and conditions
(IEC 65E/382/CDV:2014);
English version FprEN 62541-9:2014

Architecture unifiée OPC –
Partie 9: Alarmes et conditions
(CEI 65E/382/CDV:2014);
Version anglaise FprEN 62541-9:2014

Anwendungswarnvermerk

Dieser Norm-Entwurf mit Erscheinungsdatum 2014-10-24 wird der Öffentlichkeit zur Prüfung und Stellungnahme vorgelegt.

Weil die beabsichtigte Norm von der vorliegenden Fassung abweichen kann, ist die Anwendung dieses Entwurfes besonders zu vereinbaren.

Stellungnahmen werden erbeten

- vorzugsweise online im Norm-Entwurfs-Portal des DIN unter www.entwuerfe.din.de bzw. für Norm-Entwürfe der DKE auch im Norm-Entwurfs-Portal der DKE unter www.entwuerfe.normenbibliothek.de, sofern dort wiedergegeben;
- oder als Datei per E-Mail an dke@vde.com möglichst in Form einer Tabelle. Die Vorlage dieser Tabelle kann im Internet unter www.din.de/stellungnahme oder für Stellungnahmen zu Norm-Entwürfen der DKE unter www.dke.de/stellungnahme abgerufen werden;
- oder in Papierform an die DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik im DIN und VDE, Stresemannallee 15, 60596 Frankfurt am Main.

Die Empfänger dieses Norm-Entwurfs werden gebeten, mit ihren Kommentaren jegliche relevanten Patentrechte, die sie kennen, mitzuteilen und unterstützende Dokumentationen zur Verfügung zu stellen.

Gesamtumfang 94 Seiten

DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik im DIN und VDE

Nationales Vorwort

Die Englische Fassung des europäischen Dokuments FprEN 62541-9:2014 „OPC Unified Architecture – Teil 9: Alarime und Zustände“ (Entwurf in der Umfrage) ist unverändert in diesen Norm-Entwurf übernommen worden.

Die Internationale Elektrotechnische Kommission (IEC) und das Europäische Komitee für Elektrotechnische Normung (CENELEC) haben vereinbart, dass ein auf IEC-Ebene erarbeiteter Entwurf für eine Internationale Norm zeitgleich (parallel) bei IEC und CENELEC zur Umfrage (CDV-Stadium) und Abstimmung als FDIS (en: Final Draft International Standard) bzw. Schluss-Entwurf für eine Europäische Norm gestellt wird, um eine Beschleunigung und Straffung der Normungsarbeit zu erreichen. Dem entsprechend ist das internationale Dokument IEC 65E/382/CDV:2014 „OPC Unified Architecture – Part 9: Alarms and conditions“ unverändert in den Entwurf FprEN 62541-9:2014 übernommen worden.

Das internationale Dokument wurde vom SC 65E „Devices and integration in enterprise systems“ der Internationalen Elektrotechnischen Kommission (IEC) erarbeitet.

Bei der Abstimmung zu dem Europäischen Schluss-Entwurf bei CENELEC und dem Internationalen Schluss-Entwurf bei IEC [Final Draft International Standard (FDIS)] sind jeweils nur „JA/NEIN“-Entscheidungen möglich, wobei „NEIN“-Entscheidungen fundiert begründet werden müssen. Dokumente, die bei CENELEC als Europäische Norm angenommen und ratifiziert werden, sind unverändert als Deutsche Normen zu übernehmen.

Für dieses Dokument ist das nationale Arbeitsgremium K 931 „Systemaspekte“ der DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik im DIN und VDE (www.dke.de) zuständig.

Für den Fall einer undatierten Verweisung im normativen Text (Verweisung auf eine Norm ohne Angabe des Ausgabedatums und ohne Hinweis auf eine Abschnittsnummer, eine Tabelle, ein Bild usw.) bezieht sich die Verweisung auf die jeweils neueste gültige Ausgabe der in Bezug genommenen Norm.

Für den Fall einer datierten Verweisung im normativen Text bezieht sich die Verweisung immer auf die in Bezug genommene Ausgabe der Norm.

Der Zusammenhang der zitierten Normen mit den entsprechenden Deutschen Normen ergibt sich, soweit ein Zusammenhang besteht, grundsätzlich über die Nummer der entsprechenden IEC-Publikation. Beispiel: IEC 60068 ist als EN 60068 als Europäische Norm durch CENELEC übernommen und als DIN EN 60068 ins Deutsche Normenwerk aufgenommen.

Das Präsidium des DIN hat mit Präsidialbeschluss 1/2004 festgelegt, dass DIN-Normen, deren Inhalt sich auf internationale Arbeitsergebnisse der Informationsverarbeitung gründet, unter bestimmten Bedingungen allein in englischer Sprache veröffentlicht werden dürfen. Diese Bedingungen sind für den vorliegenden Normentwurf erfüllt.

Da sich die Benutzer des vorliegenden Normentwurfs der englischen Sprache als Fachsprache bedienen, wird die Englische Fassung der EN 62541-9 veröffentlicht. Zu deren Abschnitt 3, Begriffe, wurde eine Übersetzung angefertigt und als informativer Nationaler Anhang NA des vorliegenden Normentwurfs hinzugefügt. Dieser Normentwurf steht nicht in unmittelbarem Zusammenhang mit Rechtsvorschriften und ist nicht als Sicherheitsnorm anzusehen.

Es ist vorgesehen, auch bei der entsprechenden zukünftigen Deutschen Norm auf die deutsche Sprachfassung zu verzichten und diese in der englischsprachigen Fassung zu veröffentlichen.

Es wird auf die Möglichkeit hingewiesen, dass einige Elemente dieses Dokuments Patentrechte berühren können. ISO [und/oder] IEC sind nicht dafür verantwortlich, einige oder alle diesbezüglichen Patentrechte zu identifizieren.

Das Original-Dokument enthält Bilder in Farbe, die in der Papierversion in einer Graustufen-Darstellung wiedergegeben werden. Elektronische Versionen dieses Dokuments enthalten die Bilder in der originalen Farbdarstellung.

Änderungen

Gegenüber DIN EN 62541-9:2013-06 wurden folgende Änderungen vorgenommen:

- a) Bereich hinzugefügt, der das erwartete Verhalten von A&C-Servern und des zugehörigen Informationsmodells im Falle von Redundanz oder Kommunikationsfehlern beschreibt. Siehe 5.14 für weitere Einzelheiten.
- b) Den DialogConditionType so geändert, dass er nicht mehr abstrakt ist, da das Auftreten von Instanzen von ihm im System erwartet werden. Siehe Tabelle 19 für weitere Einzelheiten.
- c) Die ConditionRefresh-Methode auf den neuesten Stand gebracht, sodass die Verwendung von wohl-bekannten NodeIds in Zusammenhang mit Typen MethodId und ConditionId zu ermöglichen anstatt den Aufruf nur derjenigen MethodId und ConditionId zu verlangen, die Teil der Instanz sind. Ohne diese Änderung würden Server, die die Instanz nicht aufzeigen, Probleme mit dem ConditionRefresh haben, siehe 5.5.7 für weitere Einzelheiten.
- d) ExclusiveLimitStateMachineType und ShelvedStateMachineType sind jetzt Untertypen von FiniteStateMachineType und nicht mehr von StateMachineType. Siehe 5.8.3 und 5.8.5.2 für weitere Einzelheiten.

Nationaler Anhang NA (informativ)

3.1 Begriffe

Für die Anwendung dieses Dokuments gelten die Begriffe nach IEC 62541-1, IEC 62541-3 und IEC 62541-5 und die folgenden Begriffe.

3.1.1

Quittierung

(en: Acknowledge)

Aktion eines Operators, die das Anerkennen eines neuen Alarms anzeigt

ANMERKUNG Wie in EEMUA definiert, ist der Begriff „Akzeptieren“ (en: accept) ein weiterer gängiger Begriff für Quittierung. Sie können synonym verwendet werden. Dieses Dokument verwendet Quittierung.

3.1.2

aktiv

(en: Active)

Zustand eines Alarms, der anzeigt, dass die Situation, die der Alarm repräsentiert, aktuell existiert

ANMERKUNG Andere gängige Begriffe, definiert bei EEMUA, sind „steht an“ (en: standing) für einen aktiven Alarm und „nullgestellt“ (en: cleared), wenn die Bedingung wieder normal und nicht länger aktiv ist.

3.1.3

Bedingungsklasse

(en: ConditionClass)

eine Gruppierung von Bedingungen, die anzeigt, in welcher Domäne oder für welchen Zweck eine bestimmte Bedingung verwendet wird

ANMERKUNG Einige oberste (en: top-level) Bedingungsklassen sind in dieser Spezifikation definiert. Hersteller oder Organisationen können davon konkretere Klassen ableiten oder andere oberste Klassen definieren.

3.1.4

Bedingungsweig

(en: ConditionBranch)

ein bestimmter Zustand einer Bedingung

ANMERKUNG Der Server kann Bedingungsweige sowohl für aktuelle Zustände als auch für vorherige Zustände verwalten.

3.1.5

Bedingungsquelle

(en: **ConditionSource**)

Element, auf dem eine bestimmte Bedingung basiert oder verbunden ist

ANMERKUNG Typischerweise wird dies eine Variable sein, die einen Prozesswert (z. B. FIC101) repräsentiert, oder ein Objekt, das ein Gerät oder Teilsystem repräsentiert. Bei Ereignissen, die für Bedingungen erzeugt werden, wird die Eigenschaft Quellknoten (en: **SourceNode** – geerbt vom **BaseEventType**) die Knotenidentität der Bedingungsquelle enthalten.

3.1.6

Bestätigung

(en: **Confirm**)

Aktion eines Operators, die den Server informiert, dass eine korrigierende Aktion ausgeführt wurde, welche die Ursache des Alarms betrifft

3.1.7

deaktivieren

(en: **Disable**)

System ist so konfiguriert, dass der Alarm nicht erzeugt wird, auch wenn die Alarmbedingung erfüllt ist

ANMERKUNG Wie von EEMUA definiert.

3.1.8

Operator

(en: **Operator**)

Spezieller Benutzer, der beauftragt ist, einen Teil des Prozesses zu beobachten und zu bedienen

ANMERKUNG Ein Mitglied des Operator-Teams, das beauftragt wurde, einen Teil des Prozesses zu bedienen und zu beobachten, und der an einer Leitsystemkonsole arbeitet, wie bei EEMUA definiert. In dieser Spezifikation ist ein Operator ein spezieller Benutzer. Alle Beschreibungen, die auf einen allgemeinen Benutzer angewandt werden, gelten auch für einen Operator.

3.1.9

aktualisieren

(en: **Refresh**)

eine Erneuerung eines Ereignisabonnements, die alle Alarme liefert, die als erhaltenswert betrachtet werden

ANMERKUNG Dieses Konzept ist genauer bei EEMUA beschrieben.

3.1.10

erhalten

(en: **Retain**)

Alarm, der in einem Zustand ist, der für einen Client interessant ist, der seinen Zustand der Bedingungen mit dem Zustand im Servers synchronisieren will

3.1.11

unterdrücken

(en: **Shelving**)

Einrichtung, mit der der Operator in der Lage ist, temporär einen Alarm daran zu hindern, dem Operator angezeigt zu werden, wenn dies den Operator ablenkt

ANMERKUNG Ein unterdrückter Alarm wird von der Liste entfernt und wird nicht wieder angezeigt, bis die Unterdrückung aufgehoben wird, wie bei der EEMUA definiert.

3.1.12

ausblenden

(en: **Suppress**)

logisches Kriterium, um zu bestimmen, dass ein Alarm nicht auftritt

ANMERKUNG Ein Alarm ist ausgeblendet, wenn logische Kriterien angewandt werden, die bestimmen, dass der Alarm nicht auftreten soll, selbst wenn die Alarmbedingung (z. B. Alarmeinstellungen überschritten) erfüllt ist, wie bei der EEMUA definiert.

CONTENTS

1		
2		
3		
4	FOREWORD.....	8
5	1 Scope.....	1
6	2 Normative References	1
7	3 Terms, definitions, and abbreviations	1
8	3.1 Terms and definitions	1
9	3.2 Abbreviations and symbols	3
10	3.3 Used data types.....	3
11	4 Concepts.....	4
12	4.1 General	4
13	4.2 Conditions	4
14	4.3 Acknowledgeable Conditions	5
15	4.4 Previous States of Conditions	7
16	4.5 Condition State Synchronization	7
17	4.6 Severity, Quality, and Comment.....	8
18	4.7 Dialogs	8
19	4.8 Alarms	8
20	4.9 Multiple Active States	9
21	4.10 Condition Instances in the Address Space	10
22	4.11 Alarm and Condition Auditing.....	11
23	5 Model	12
24	5.1 General	12
25	5.2 Two-State State Machines	12
26	5.3 Condition Variables.....	13
27	5.4 Substate Reference Types.....	14
28	5.4.1 General	14
29	5.4.2 HasTrueSubState ReferenceType	14
30	5.4.3 HasFalseSubState ReferenceType	14
31	5.5 Condition Model.....	15
32	5.5.1 General	15
33	5.5.2 ConditionType	16
34	5.5.3 Condition and Branch Instances	19
35	5.5.4 Disable Method	19
36	5.5.5 Enable Method	20
37	5.5.6 AddComment Method	20
38	5.5.7 ConditionRefresh Method	21
39	5.6 Dialog Model.....	23
40	5.6.1 General	23
41	5.6.2 DialogConditionType	23
42	5.6.3 Respond Method	24
43	5.7 Acknowledgeable Condition Model.....	26
44	5.7.1 General	26
45	5.7.2 AcknowledgeableConditionType	26
46	5.7.3 Acknowledge Method.....	27
47	5.7.4 Confirm Method.....	28
48	5.8 Alarm Model	29
49	5.8.1 General	29

50	5.8.2 AlarmConditionType	29
51	5.8.3 LimitAlarmType	35
52	5.8.4 ExclusiveLimit Types	36
53	5.8.5 NonExclusiveLimitAlarmType	40
54	5.8.6 Level Alarm	41
55	5.8.7 Deviation Alarm	41
56	5.8.8 Rate of Change	42
57	5.8.9 Discrete Alarms	44
58	5.9 ConditionClasses	46
59	5.9.1 Overview	46
60	5.10 Audit Events	48
61	5.10.1 Overview	48
62	5.10.2 AuditConditionEventType	48
63	5.10.3 AuditConditionEnableEventType	49
64	5.10.4 AuditConditionCommentEventType	49
65	5.10.5 AuditConditionRespondEventType	49
66	5.10.6 AuditConditionAcknowledgeEventType	50
67	5.10.7 AuditConditionConfirmEventType	50
68	5.10.8 AuditConditionShelvingEventType	50
69	5.11 Condition Refresh Related Events	50
70	5.11.1 Overview	50
71	5.11.2 RefreshStartEventType	51
72	5.11.3 RefreshEndEventType	51
73	5.11.4 RefreshRequiredEventType	51
74	5.12 HasCondition Reference Type	52
75	5.13 Alarm & Condition Status Codes	52
76	5.14 Expected A&C Server Behaviours	53
77	5.14.1 Communication problems	53
78	5.14.2 Redundant A&C Servers	53
79	6 AddressSpace Organisation	54
80	6.1 General	54
81	6.2 Event Notifier and Source Hierarchy	54
82	6.3 Adding Conditions to the Hierarchy	54
83	6.4 Conditions in InstanceDeclarations	55
84	6.5 Conditions in a VariableType	55
85	Annex A (informative): Recommended Localized Names	57
86	A.1 Recommended State Names for TwoState Variables	57
87	A.1.1 LocaleId “en”	57
88	A.1.2 LocaleId “de”	57
89	A.1.3 LocaleId “fr”	58
90	A.2 Recommended Dialog Response Options	59
91	Annex B (informative): Examples	60
92	B.1 Examples for Event sequences from Condition instances	60
93	B.1.1 Overview	60
94	B.1.2 Server Maintains Current State Only	60
95	B.1.3 Server Maintains Previous States	60
96	B.2 Address Space Examples	62
97	Annex C (informative): Mapping to EEMUA	

98 Annex D (informative): Mapping from OPC A&E to OPC UA A&C 65

99 D.1 Overview 65

100 D.1.1 Alarms and Events COM UA Wrapper 65

101 D.1.2 Alarms and Events COM UA Proxy 70

102 Annex E (informative): Revision 1.02 Highlights 80

103

104

105

106

Figures

107	Figure 1 – Base Condition State Model	5
108	Figure 2 - AcknowledgeableConditions State Model.....	5
109	Figure 3 – Acknowledge State Model	6
110	Figure 4 – Confirmed Acknowledge State Model	6
111	Figure 5 – Alarm State Machine Model	9
112	Figure 6 – Multiple Active States Example	10
113	Figure 7 - ConditionType Hierarchy.....	12
114	Figure 8 - Condition Model.....	16
115	Figure 9 - DialogConditionType Overview	23
116	Figure 10 - AcknowledgeableConditionType Overview	26
117	Figure 11 – AlarmConditionType Hierarchy Model.....	29
118	Figure 12 - Alarm Model	30
119	Figure 13 - Shelve state transitions.....	32
120	Figure 14 - Shelved State Machine Model.....	32
121	Figure 15 - LimitAlarmType	36
122	Figure 16 - ExclusiveLimitStateMachine.....	37
123	Figure 17 - ExclusiveLimitAlarmType	39
124	Figure 18 - NonExclusiveLimitAlarmType	40
125	Figure 19 - DiscreteAlarmType Hierarchy.....	44
126	Figure 20 - ConditionClass Type Hierarchy	46
127	Figure 21 – AuditEvent Hierarchy	48
128	Figure 22 – Refresh Related Event Hierarchy	50
129	Figure 23 – Typical Event Hierarchy	54
130	Figure 24 – Use of HasCondition in an Event Hierarchy	55
131	Figure 25 – Use of HasCondition in an InstanceDeclaration	55
132	Figure 26 – Use of HasCondition in a VariableType	56
133	Figure 27 – Single State Example	60
134	Figure 28 – Previous State Example	61
135	Figure 29 – HasCondition used with Condition instances	62
136	Figure 30 – HasCondition reference to a Condition Type	63
137	Figure 31 – HasCondition used with an instance declaration.....	63
138	Figure 32 – The Type Model of a Wrapped COM AE Server.....	67
139	Figure 33 – Mapping UA Event Types to COM A&E Event Types	71
140	Figure 34 – Example Mapping of UA Event Types to COM A&E Categories	72
141	Figure 35 – Example Mapping of UA Event Types to A&E Categories with Attributes	76

142

143

144	Tables	
145	Table 1 – Parameter Types defined in IEC 62541-3	3
146	Table 2 – Parameter Types defined in IEC 62541-4	4
147	Table 3 – TwoStateVariableType Definition.....	13
148	Table 4 – ConditionVariableType Definition	14
149	Table 5 – HasTrueSubState ReferenceType	14
150	Table 6 – HasFalseSubState ReferenceType	15
151	Table 7 – ConditionType Definition	17
152	Table 8 - Simple Attribute Operand	19
153	Table 9 - Disable Result Codes.....	20
154	Table 10 – Disable Method AddressSpace Definition	20
155	Table 11 - Enable Result Codes.....	20
156	Table 12 – Enable Method AddressSpace Definition	20
157	Table 13 - AddComment Arguments.....	21
158	Table 14 - AddComment result Codes.....	21
159	Table 15 – AddComment Method AddressSpace Definition	21
160	Table 16 _ ConditionRefresh Parameters.....	22
161	Table 17 - ConditionRefresh ReturnCodes	22
162	Table 18 – ConditionRefresh Method AddressSpace Definition	22
163	Table 19 – DialogConditionType Definition	23
164	Table 20 - Repond Parameters	24
165	Table 21 - Respond ResultCodes.....	24
166	Table 22 – Respond Method AddressSpace Definition	25
167	Table 23 – AcknowledgeableConditionType Definition	26
168	Table 24 - Acknowledge Parameters.....	27
169	Table 25 - Acknowledge result codes.....	27
170	Table 26 – Acknowledge Method AddressSpace Definition	28
171	Table 27 – Confirm Method Parameters	28
172	Table 28 - Confirm Result Codes	28
173	Table 29 – Confirm Method AddressSpace Definition.....	29
174	Table 30 – AlarmConditionType Definition	30
175	Table 31 –ShelvedStateMachine Definition	33
176	Table 32 – ShelvedStateMachine Transitions.....	33
177	Table 33 - Unshelve Result Codes	34
178	Table 34 – Unshelve Method AddressSpace Definition	34
179	Table 35 - TimedShelve Parameters	34
180	Table 36 - TimedShelve Result Codes	34
181	Table 37 – TimedShelve Method AddressSpace Definition.....	35
182	Table 38 - OneShotShelve Result Codes	35
183	Table 39 – OneShotShelve Method AddressSpace Definition.....	35
184	Table 40 –LimitAlarmType Definition.....	36
185	Table 41 – ExclusiveLimitStateMachineType Definition.....	37
186	Table 42 – ExclusiveLimitStateMachineType Transitions	38

187	Table 43 – ExclusiveLimitAlarmType Definition	39
188	Table 44 – NonExclusiveLimitAlarmType Definition	40
189	Table 45 – NonExclusiveLevelAlarmType Definition	41
190	Table 46 – ExclusiveLevelAlarmType Definition	41
191	Table 47 – NonExclusiveDeviationAlarmType Definition	42
192	Table 48 – ExclusiveDeviationAlarmType Definition	42
193	Table 49 – NonExclusiveRateOfChangeAlarmType Definition	43
194	Table 50 – ExclusiveRateOfChangeAlarmType Definition	43
195	Table 51 – DiscreteAlarmType Definition	44
196	Table 52 – OffNormalAlarmType Definition	44
197	Table 53 – SystemOffNormalAlarmType Definition	45
198	Table 54 – TripAlarmType Definition	45
199	Table 55 – BaseConditionClassType Definition	46
200	Table 56 – ProcessConditionClassType Definition	47
201	Table 57 – MaintenanceConditionClassType Definition	47
202	Table 58 – SystemConditionClassType Definition	48
203	Table 59 – AuditConditionEventType Definition	48
204	Table 60 – AuditConditionEnableEventType Definition	49
205	Table 61 – AuditConditionCommentEventType Definition	49
206	Table 62 – AuditConditionRespondEventType Definition	49
207	Table 63 – AuditConditionAcknowledgeEventType Definition	50
208	Table 64 – AuditConditionConfirmEventType Definition	50
209	Table 65 – AuditConditionShelvingEventType Definition	50
210	Table 66 – RefreshStartEventType Definition	51
211	Table 67 – RefreshEndEventType Definition	51
212	Table 68 – RefreshRequiredEventType Definition	51
213	Table 69 – HasCondition ReferenceType	52
214	Table 70 – Alarm & Condition Result Codes	52
215	Table 71 – Recommended state names for LocaleId “en”	57
216	Table 72 – Recommended display names for LocaleId “en”	57
217	Table 73 – Recommended state names for LocaleId “de”	57
218	Table 74 – Recommended display names for LocaleId “de”	57
219	Table 75 – Recommended state names for LocaleId “fr”	58
220	Table 76 – Recommended display names for LocaleId “fr”	58
221	Table 77 – Recommended Dialog Response Options	59
222	Table 78 – Example of a Condition that only keeps the latest state	60
223	Table 79 – Example of a <i>Condition</i> that maintains previous states via branches	61
224	Table 80 – EEMUA Terms	64
225	Table 81 – Mapping from Standard Event Categories to OPC UA Event Types	66
226	Table 82 – Mapping from ONEVENTSTRUCT fields to UA BaseEventType Variables	68
227	Table 83 – Mapping from ONEVENTSTRUCT fields to UA AuditEventType Variables	68
228	Table 84 – Mapping from ONEVENTSTRUCT fields to UA AlarmType Variables	69
229	Table 85 – Event Category Attribute Mapping Table	73

INTERNATIONAL ELECTROTECHNICAL COMMISSION

OPC Unified Architecture –

Part 9: Alarms & 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.

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

FDIS	Report on voting
65E/XX/FDIS	65E/XX/RVD

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

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.

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

- 286 • reconfirmed,
287 • withdrawn,
288 • replaced by a revised edition, or
289 • amended.

290

291 The National Committees are requested to note that for this publication the stability date
292 is 2017.

293 THIS TEXT IS INCLUDED FOR THE INFORMATION OF THE NATIONAL COMMITTEES AND WILL BE
294 DELETED AT THE PUBLICATION STAGE.

295

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.

296

OPC Unified Architecture

Part 9: Alarms & Conditions

1 Scope

This standard 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 62541-7, *OPC Unified Architecture – Part 7: Profiles*

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

IEC 62541-11, *OPC Unified Architecture – Part 11: Historical Access*

Additional external reference used to provide *Information Model* suggestions for this document:

EEMUA : 2nd Edition EEMUA 191 - Alarm System - A guide to design, management and procurement (Appendixes 6, 7, 8, 9).

<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.

330 **3.1.1**331 **Acknowledge**332 “a *Operator* action that indicates recognition of a new *Alarm*”

333 Note: as defined in EEMUA. The term “Accept” is another common term used to describe *Acknowledge*. They can be used
334 interchangeably. This document will use *Acknowledge*.

335 **3.1.2**336 **Active**337 a *state for an Alarm* that indicates that the situation the *Alarm* is representing currently exists.

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

340 **3.1.3**341 **ConditionClass**342 a *Condition* grouping that indicates in which domain or for what purpose a certain *Condition* is used.

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

345 **3.1.4**346 **ConditionBranch**347 a specific state of a *Condition*.348 Note: The *Server* can maintain *ConditionBranches* for the current state as well as for previous states.349 **3.1.5**350 **ConditionSource**351 an element which a specific *Condition* is based upon or related to.

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

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

356 **3.1.6**357 **Confirm**

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

360 **3.1.7**361 **Disable**

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

364 Note: This concept is further described in EEMUA

365 **3.1.8**366 **Operator**

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

368 Note: “A Member of the operations team who is assigned to monitor and control a portion of the process and is working at
369 the control system’s Console” as defined in EEMUA. In this standard an *Operator* is a special user. All descriptions that
370 apply to general users also apply to *Operators*.

371 **3.1.9**
372 **Refresh**

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

375 Note: This concept is further described in EEMUA.

376 **3.1.10**
377 **Retain**

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

380 **3.1.11**
381 **Shelving**

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

384 Note: A Shelved *Alarm* will be removed from the list and will not re-annunciate until un-shelved” as defined in EEMUA.

385 **3.1.12**
386 **Suppress**

387 the act of determining whether an *Alarm* should not occur

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

390 **3.2 Abbreviations and symbols**

391 A&E Alarm&Event (as used for OPC COM)
392 COM (Microsoft Windows) Component Object Model
393 DA Data Access
394 UA Unified Architecture
395

396 **3.3 Used data types**

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

400 **Table 1 – Parameter Types defined in IEC 62541-3**

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

401

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*. Clause 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. Clause 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.

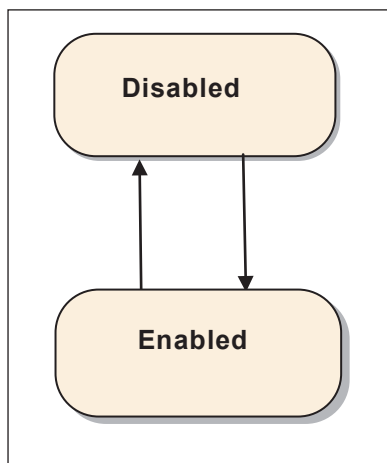


Figure 1 – Base Condition State Model

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 Clause 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*.

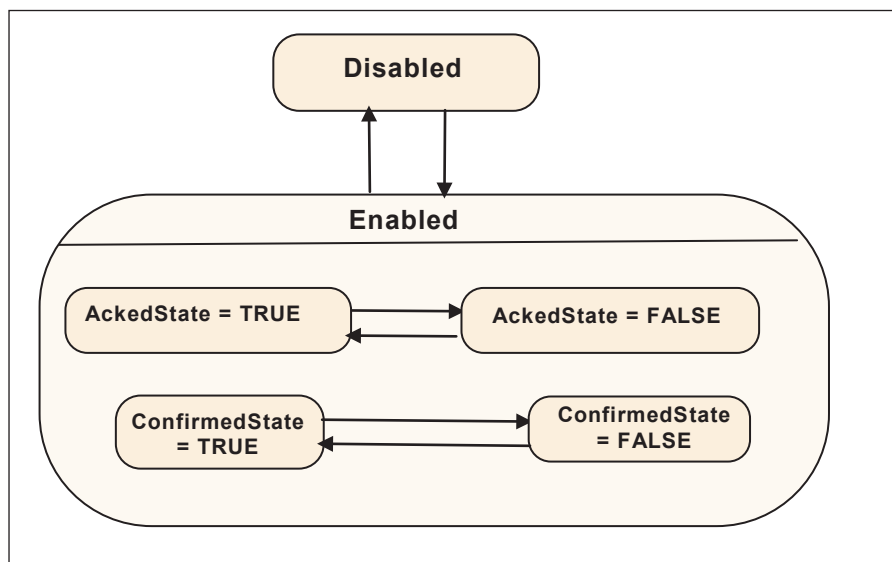


Figure 2 - AcknowledgeableConditions State Model

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.

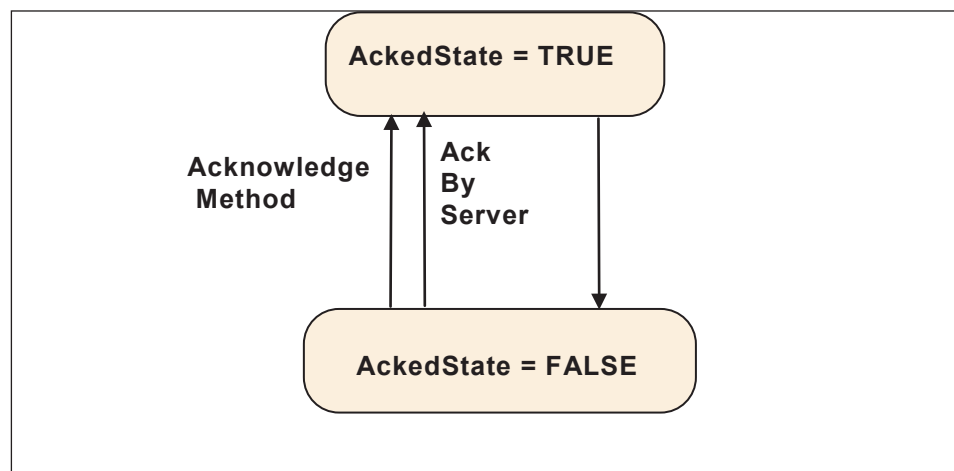


Figure 3 – Acknowledge State Model

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.

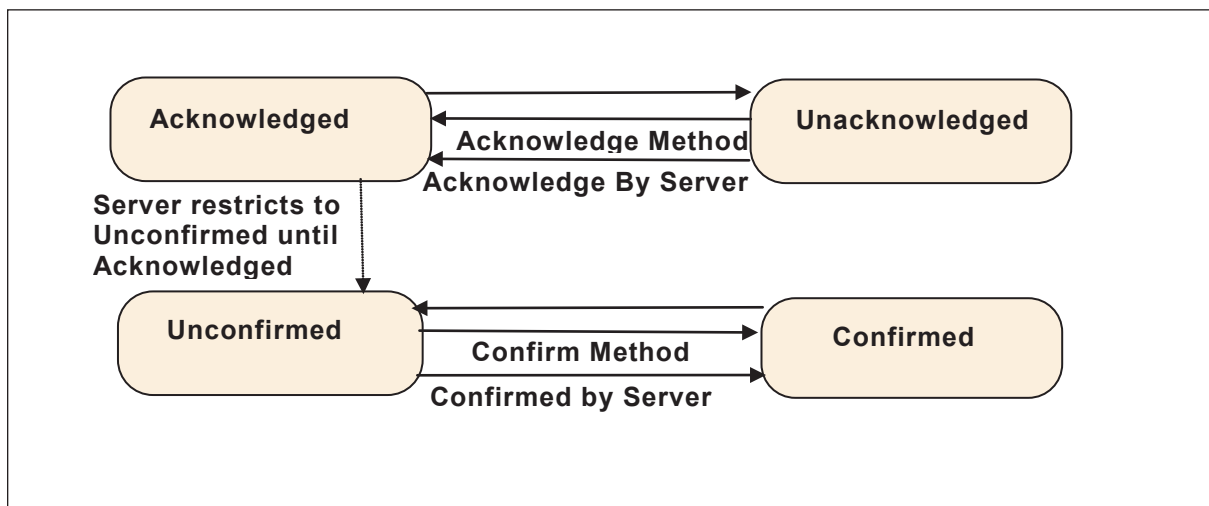


Figure 4 – Confirmed Acknowledge State Model

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 Clause 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.

- 536 • To ensure a *Client* is always informed, the three special *EventTypes* (*RefreshEndEventType*,
537 *RefreshStartEventType* and *RefreshRequiredEventType*) ignore the *Event* content filtering
538 associated with a *Subscription* and will always be delivered to the *Client*.

539 4.6 Severity, Quality, and Comment

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

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

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

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

552 4.7 Dialogs

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

559 4.8 Alarms

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

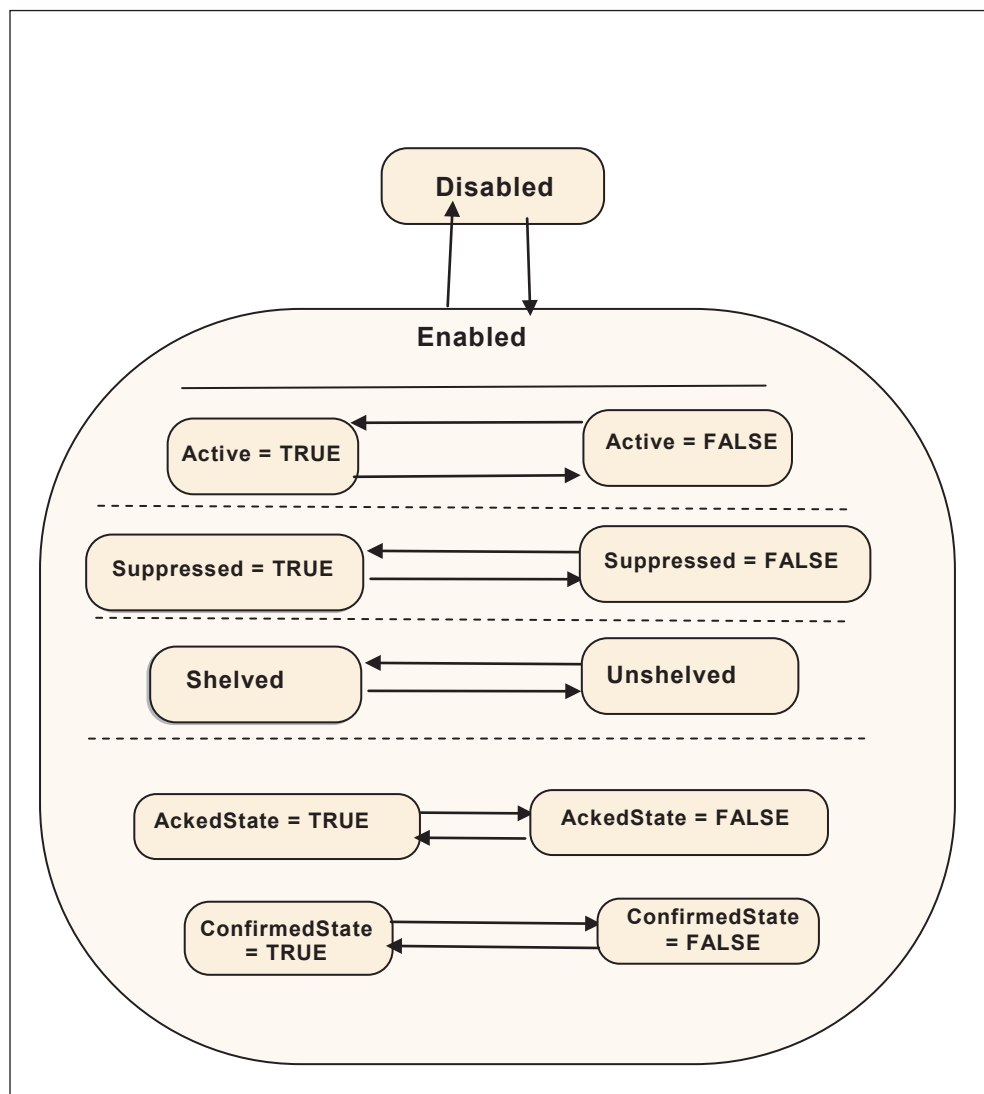


Figure 5 – Alarm State Machine Model

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).

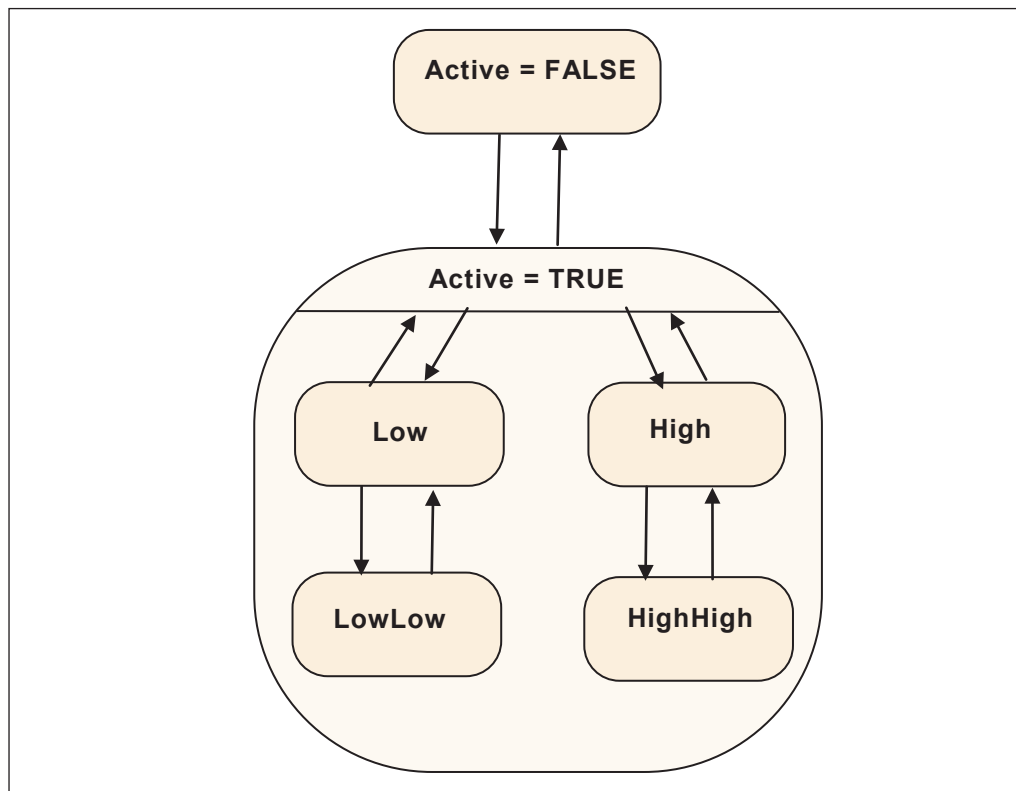
The *Shelving* state can be set by an *Operator* via OPC UA *Methods*. The *Suppressed* state is set internally by the *Server* due to system specific reasons. *Alarm* systems typically implement the *Suppress* and *Shelve* features to help keep *Operators* from being overwhelmed during *Alarm* “storms” 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

579 in one of the following sub-states: LowLow, Low, High or HighHigh. The state model is illustrated in
580 Figure 6.



582 **Figure 6 – Multiple Active States Example**

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

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

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

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

593 4.10 Condition Instances in the Address Space

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

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

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

606 4.11 Alarm and Condition Auditing

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

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

617 This standard describes the *AuditEventType* that each *Method* is required to generate. For example,
618 the *Disable Method* has an *AlwaysGeneratesEvent Reference* to an
619 *AuditConditionEnableEventType*. An *Event* of this type shall be generated for every invocation of the
620 *Method*. The audit *Event* describes the user interaction with the system, in some cases this
621 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 sub clauses 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.

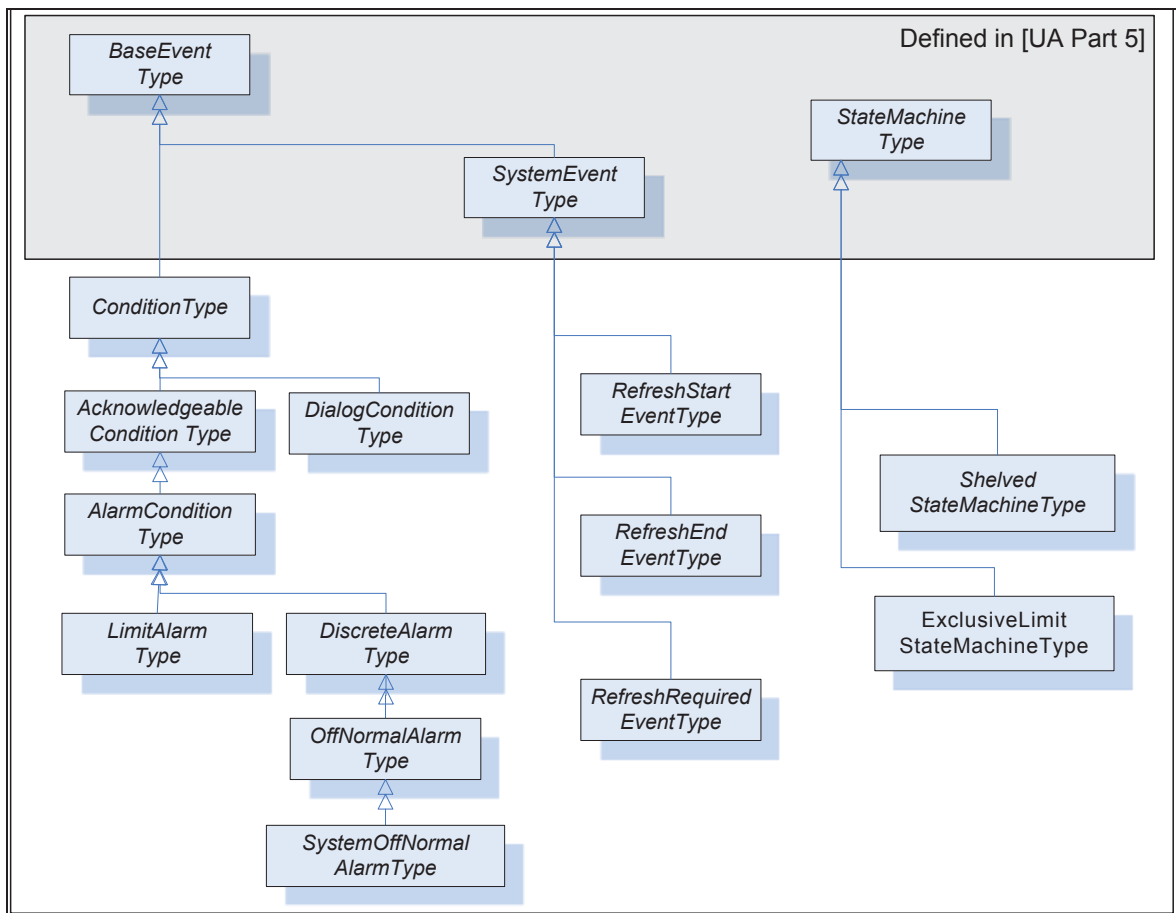


Figure 7 - ConditionType Hierarchy

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.

640

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 Clause 5.4.2		Optional
HasFalseSubState	StateMachine or TwoStateVariableType	<StateIdentifier>	Defined in Clause 5.4.3		Optional

641

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

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

647 *TransitionTime* specifies the time when the current state was entered.

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

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

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

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

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

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

664 5.3 Condition Variables

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

668 *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 than 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*.

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

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

708 It is not required to provide the *HasFalseSubState Reference* from super state to sub state, but it is
 709 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

711

712 5.5 Condition Model

713 5.5.1 General

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

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

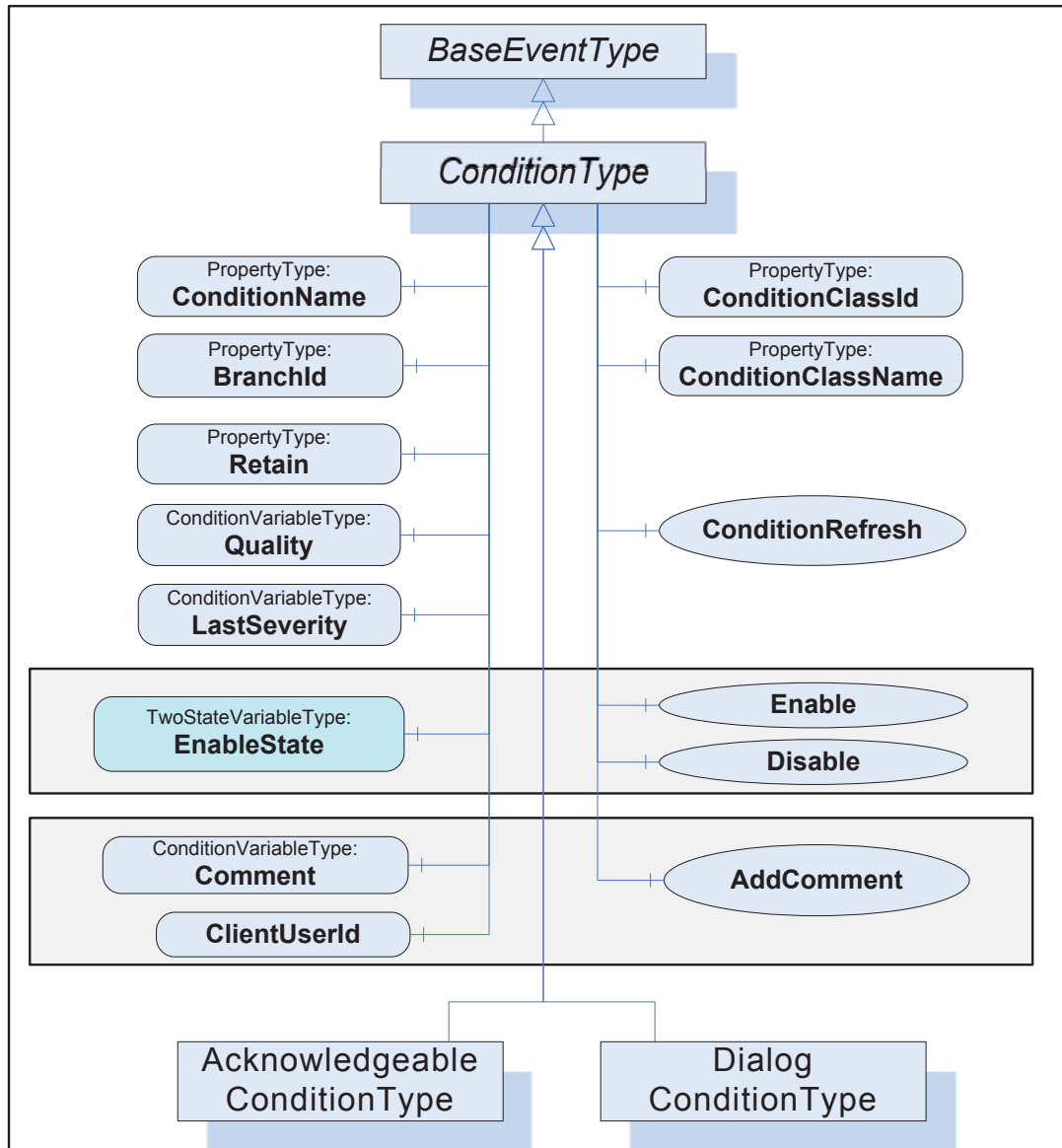


Figure 8 - Condition Model

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.