



DPC: 24 / 30484424 DC

BSI Group Headquarters

389 Chiswick High Road London W4 4AL

Tel: +44 (0)20 8996 9000

Fax: +44 (0)20 8996 7400

www.bsigroup.com

Date: 29 January 2024

Origin: International

Latest date for receipt of comments: 20 March 2024

Project No. 2023/02973

Responsible committee: GEL/65/5 Devices and integration in enterprise systems

Interested committees:

Title: BS EN IEC 62541-9 OPC Unified Architecture

Part 9: Alarms and Conditions

Please notify the secretary if you are aware of any keywords that might assist in classifying or identifying the standard or if the content of this standard

- i) has any issues related to 3rd party IPR, patent or copyright
- ii) affects other national standard(s)
- iii) requires additional national guidance or information

**WARNING: THIS IS A DRAFT AND MUST NOT BE REGARDED OR USED AS A BRITISH STANDARD.
THIS DRAFT IS NOT CURRENT BEYOND 20 March 2024**

This draft is issued to allow comments from interested parties; all comments will be given consideration prior to publication. No acknowledgement will normally be sent. **See overleaf for information on the submission of comments.**

No copying is allowed, in any form, without prior written permission from BSI except as permitted under the Copyright, Designs and Patent Act 1988 or for circulation within a nominating organization for briefing purposes. Electronic circulation is limited to dissemination by e-mail within such an organization by committee members.

Further copies of this draft may be purchased from BSI Shop <http://shop.bsigroup.com> or from BSI Customer Services, Tel: +44(0) 20 8996 9001 or email cservices@bsigroup.com. British, International and foreign standards are also available from BSI Customer Services.

Information on the co-operating organizations represented on the committees referenced above may be obtained from <http://standardsdevelopment.bsigroup.com>

Responsible Committee Secretary: Ms Charmaine Mukherjee (BSI)

Direct tel:

E-mail: charmaine.mukherjee@bsigroup.com

Introduction

This draft standard is based on national and international discussions. Your comments on this draft are invited and will assist in the preparation of the consequent standard.

For international standards, comments will be reviewed by the relevant UK national committee before sending the consensus UK vote and comments to the international committee, which will then decide appropriate action. If the international standard is approved, it is usual for the text to be published as a British Standard.

For national standards, comments will be reviewed by the relevant UK national committee and the resulting standards published as a British Standard.

UK Vote

Please indicate whether you consider the UK should submit a negative (with supporting technical reasons) or positive vote on this draft. Please indicate if you are aware of any reason why this draft standard should not be published as a British Standard.

Submission of Comments

- **Annotated drafts are not acceptable and will be rejected.**
- All comments should be submitted online at <https://standardsdevelopment.bsigroup.com/>. You will need to register in order to comment.

Template for comments and secretariat observations

Date: xx/xx/20xx	Document: ISO/DIS xxxx
------------------	------------------------

1	2	(3)	4	5	(6)	7
MB	Clause No./ Subclause No./Annex (e.g. 3.1)	Paragraph/Figure/ Table/Note	Type of comment	Commend (justification for change) by the MB	Proposed change by the MB	Secretariat observations on each comment submitted
	3.1	Definition 1	ed	Definition is ambiguous and needs clarifying.	Amend to read '...so that the mains connector to which no connection...'	
	6.4	Paragraph 2	te	The use of the UV photometer as an alternative cannot be supported as serious problems have been encountered in its use in the UK.	Delete reference to UV photometer.	

COMMITTEE DRAFT FOR VOTE (CDV)

PROJECT NUMBER:

IEC 62541-9 ED4

DATE OF CIRCULATION:

2024-01-26

CLOSING DATE FOR VOTING:

2024-04-19

SUPERSEDES DOCUMENTS:

65E/981/RR

IEC SC 65E : DEVICES AND INTEGRATION IN ENTERPRISE SYSTEMS

SECRETARIAT:

United States of America

SECRETARY:

Mr Donald (Bob) Lattimer

OF INTEREST TO THE FOLLOWING COMMITTEES:

PROPOSED HORIZONTAL STANDARD:

☐

Other TC/SCs are requested to indicate their interest, if any, in this CDV to the secretary.

FUNCTIONS CONCERNED:

☐ EMC☐ ENVIRONMENT☐ QUALITY ASSURANCE☐ SAFETY☒ SUBMITTED FOR CENELEC PARALLEL VOTING☐ NOT SUBMITTED FOR CENELEC PARALLEL VOTING**Attention IEC-CENELEC parallel voting**

The attention of IEC National Committees, members of CENELEC, is drawn to the fact that this Committee Draft for Vote (CDV) is submitted for parallel voting.

The CENELEC members are invited to vote through the CENELEC online voting system.

This document is still under study and subject to change. It should not be used for reference purposes.

Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Recipients of this document are invited to submit, with their comments, notification of any relevant "In Some Countries" clauses to be included should this proposal proceed. Recipients are reminded that the CDV stage is the final stage for submitting ISC clauses. (SEE [AC/22/2007](#) OR [NEW GUIDANCE DOC](#)).

TITLE:

OPC Unified Architecture - Part 9: Alarms and Conditions

PROPOSED STABILITY DATE: 2026

NOTE FROM TC/SC OFFICERS:

Copyright © 2024 International Electrotechnical Commission, IEC. All rights reserved. It is permitted to download this electronic file, to make a copy and to print out the content for the sole purpose of preparing National Committee positions. You may not copy or "mirror" the file or printed version of the document, or any part of it, for any other purpose without permission in writing from IEC.

CONTENTS

FIGURES	iv
TABLES	v
FOREWORD	x
1 Scope	1
2 Normative references	1
3 Terms, definitions, and abbreviations	2
3.1 Terms and definitions	2
3.2 Abbreviations and symbols	4
3.3 Used data types	4
4 Concepts	4
4.1 General	4
4.2 Conditions	4
4.3 Acknowledgeable Conditions	6
4.4 Previous states of Conditions	8
4.5 Condition state synchronization	8
4.6 Severity, quality, and comment	9
4.7 Dialogs	9
4.8 Alarms	9
4.9 Multiple active states	11
4.10 Condition instances in the AddressSpace	12
4.11 Alarm and Condition auditing	13
4.12 Alarms in a system	13
5 Model	13
5.1 General	13
5.2 Two-state state machines	14
5.3 ConditionVariable	16
5.4 ReferenceTypes	17
5.4.1 General	17
5.4.2 HasTrueSubState ReferenceType	17
5.4.3 HasFalseSubState ReferenceType	17
5.4.4 HasAlarmSuppressionGroup ReferenceType	18
5.4.5 AlarmGroupMember ReferenceType	18
5.4.6 AlarmSuppressionGroupMember ReferenceType	18
5.5 Condition Model	19
5.5.1 General	19
5.5.2 ConditionType	20
5.5.3 Condition and branch instances	24
5.5.4 Disable Method	24
5.5.5 Enable Method	25
5.5.6 AddComment Method	25
5.5.7 ConditionRefresh Method	26
5.5.8 ConditionRefresh2 Method	28
5.6 Dialog Model	30

46	5.6.1	General.....	30
47	5.6.2	DialogConditionType	30
48	5.6.3	Respond Method	31
49	5.6.4	Respond2 Method	32
50	5.7	Acknowledgeable Condition Model.....	33
51	5.7.1	General.....	33
52	5.7.2	AcknowledgeableConditionType	33
53	5.7.3	Acknowledge Method	34
54	5.7.4	Confirm Method.....	35
55	5.8	Alarm model	36
56	5.8.1	General.....	36
57	5.8.2	AlarmConditionType	37
58	5.8.3	AlarmGroupType	42
59	5.8.4	AlarmSuppressionGroupType	42
60	5.8.5	Reset Method.....	42
61	5.8.6	Reset2 Method.....	43
62	5.8.7	Silence Method	44
63	5.8.8	Suppress Method	45
64	5.8.9	Suppress2 Method	46
65	5.8.10	Unsuppress Method.....	46
66	5.8.11	Unsuppress2 Method.....	47
67	5.8.12	RemoveFromService Method	48
68	5.8.13	RemoveFromService2 Method	48
69	5.8.14	PlaceInService Method.....	49
70	5.8.15	PlaceInService2 Method	50
71	5.8.16	GetGroupMemberships Method	51
72	5.8.17	ShelvedStateMachineType	51
73	5.8.18	LimitAlarmType	59
74	5.8.19	Exclusive Limit Types	62
75	5.8.20	NonExclusiveLimitAlarmType.....	65
76	5.8.21	Level Alarm	67
77	5.8.22	Deviation Alarm.....	68
78	5.8.23	Rate of change Alarms	69
79	5.8.24	Discrete Alarms.....	70
80	5.8.25	DiscrepancyAlarmType.....	73
81	5.9	ConditionClasses	74
82	5.9.1	Overview	74
83	5.9.2	BaseConditionClassType.....	74
84	5.9.3	ProcessConditionClassType	75
85	5.9.4	MaintenanceConditionClassType	75
86	5.9.5	SystemConditionClassType	75
87	5.9.6	SafetyConditionClassType.....	76
88	5.9.7	HighlyManagedAlarmConditionClassType	76
89	5.9.8	TrainingConditionClassType	76
90	5.9.9	StatisticalConditionClassType.....	77
91	5.9.10	TestingConditionClassType	77
92	5.10	Audit Events.....	77
93	5.10.1	Overview.....	77

94	5.10.2	AuditConditionEventType	78
95	5.10.3	AuditConditionEnableEventType	79
96	5.10.4	AuditConditionCommentEventType	79
97	5.10.5	AuditConditionRespondEventType	79
98	5.10.6	AuditConditionAcknowledgeEventType	79
99	5.10.7	AuditConditionConfirmEventType	80
100	5.10.8	AuditConditionShelvingEventType	80
101	5.10.9	AuditConditionSuppressionEventType	80
102	5.10.10	AuditConditionSilenceEventType	81
103	5.10.11	AuditConditionResetEventType	81
104	5.10.12	AuditConditionOutOfServiceEventType	81
105	5.11	Condition Refresh related Events	82
106	5.11.1	Overview	82
107	5.11.2	RefreshStartEventType	82
108	5.11.3	RefreshEndEventType	82
109	5.11.4	RefreshRequiredEventType	83
110	5.12	HasCondition Reference type	83
111	5.13	Alarm & Condition status codes	83
112	5.14	Expected A & C server behaviours	84
113	5.14.1	General	84
114	5.14.2	Communication problems	84
115	5.14.3	Redundant A & C servers	84
116	6	AddressSpace organisation	85
117	6.1	General	85
118	6.2	EventNotifier and source hierarchy	85
119	6.3	Adding Conditions to the hierarchy	85
120	6.4	Conditions in InstanceDeclarations	86
121	6.5	Conditions in a VariableType	86
122	7	System State & Alarms	88
123	7.1	Overview	88
124	7.2	HasEffectDisable	88
125	7.3	HasEffectEnable	88
126	7.4	HasEffectSuppressed	89
127	7.5	HasEffectUnsuppressed	89
128	8	Alarm Summary and Objects	90
129	8.1	Overview	90
130	8.2	AlarmState Variable	91
131	8.3	AlarmMask	92
132	9	Alarm Metrics	92
133	9.1	Overview	92
134	9.2	AlarmMetricsType	93
135	9.3	AlarmRateVariableType	94
136	9.4	Reset Method	94
137	Annex A (informative)	Recommended localized names	95
138	A.1	Recommended state names for TwoState Variables	95
139	A.1.1	LocaleId “en”	95
140	A.1.2	LocaleId “de”	95

141	A.1.3	LocaleId “fr”	96
142	A.2	Recommended dialog response options	96
143	Annex B (informative)	Examples	98
144	B.1	Examples for Event sequences from Condition instances	98
145	B.1.1	Overview	98
146	B.1.2	Server maintains current state only	98
147	B.1.3	Server maintains previous states	98
148	B.1.4	Server current-State Model with Suppression	100
149	B.1.5	Example for On-Delay, Off Delay and ReAlarmTime	101
150	B.2	AddressSpace examples	102
151	Annex C (informative)	Mapping to EEMUA	105
152	Annex D (informative)	Mapping from OPC A&E to OPC UA A & C	106
153	D.1	Overview	106
154	D.2	Alarms and Events COM UA wrapper	106
155	D.2.1	Event areas	106
156	D.2.2	Event sources	106
157	D.2.3	Event categories	107
158	D.2.4	Event attributes	108
159	D.2.5	Event subscriptions	108
160	D.2.6	Condition instances	110
161	D.2.7	Condition Refresh	111
162	D.3	Alarms and Events COM UA proxy	111
163	D.3.1	General	111
164	D.3.2	Server status mapping	111
165	D.3.3	Event Type mapping	111
166	D.3.4	Event category mapping	112
167	D.3.5	Event Category attribute mapping	113
168	D.3.6	Event Condition mapping	116
169	D.3.7	Browse mapping	116
170	D.3.8	Qualified names	117
171	D.3.9	Subscription filters	118
172	Annex E – IEC 62682 Mapping		120
173	E.1	Overview	120
174	E.2	Terms	120
175	E.3	Alarm records & State Indications	124
176	Annex F System State (Informative)		125
177	F.1	Overview	125
178	F.2	SystemStateStateMachineType	126

179

180 FIGURES

181

182	Figure 1 – Base Condition state model	5
183	Figure 2 – AcknowledgeableConditions state model	6
184	Figure 5 – Alarm state machine model	10
185	Figure 6 – Typical Alarm Timeline example	11
186	Figure 7 – Multiple active states example	12

187	Figure 8 – ConditionType hierarchy	14
188	Figure 10 – Condition model.....	20
189	Figure 12 – DialogConditionType Overview.....	30
190	Figure 13 – AcknowledgeableConditionType overview	33
191	Figure 14 – AlarmConditionType Hierarchy Model.....	37
192	Figure 15 – Alarm Model	38
193	Figure 16 – Shelve state transitions.....	52
194	Figure 17 – ShelvedStateMachineType model.....	53
195	Figure 18 – LimitAlarmType.....	60
196	Figure 19 – ExclusiveLimitStateMachineType	63
197	Figure 20 – ExclusiveLimitAlarmType	65
198	Figure 21 – NonExclusiveLimitAlarmType	66
199	Figure 22 – DiscreteAlarmType Hierarchy.....	71
200	Figure 23 – ConditionClass type hierarchy	74
201	Figure 24 – AuditEvent hierarchy.....	78
202	Figure 25 – Refresh Related Event Hierarchy	82
203	Figure 26 – Typical HasNotifier Hierarchy.....	85
204	Figure 27 – Use of HasCondition in a HasNotifier hierarchy	86
205	Figure 28 – Use of HasCondition in an InstanceDeclaration	86
206	Figure 29 – Use of HasCondition in a VariableType	87
207	Figure B.1 – Single state example	98
208	Figure B.2 – Previous state example	99
209	Figure B.3 – SuppressedState and OutOfServiceState example	100
210	Figure B.5 – HasCondition used with Condition instances	103
211	Figure B.6 – HasCondition reference to a Condition type	104
212	Figure B.7 – HasCondition used with an instance declaration.....	104
213	Figure D.1 – The type model of a wrapped COM AE server.....	108
214	Figure D.2 – Mapping UA Event Types to COM A&E Event Types	112
215	Figure D.3 – Example mapping of UA Event Types to COM A&E categories	113
216	Figure D.4 – Example mapping of UA Event Types to A&E categories with attributes.....	116
217	Figure F.1 – SystemState transitions	125
218	Figure F.2 – SystemStateStateMachineType Model	126

219

220

221

TABLES

222	Table 1 – Parameter types defined in 10000-3	4
223	Table 2 – Parameter types defined in 10000-4	4
224	Table 3 – TwoStateVariableType definition	14
225	Table 4 – ConditionVariableType definition	16
226	Table 5 – HasTrueSubState ReferenceType	17
227	Table 6 – HasFalseSubState ReferenceType	18
228	Table 7 – HasAlarmSuppressionGroup ReferenceType	18

229	Table 8 – AlarmGroupMember ReferenceType.....	18
230	Table 9 – AlarmSuppressionGroupMember ReferenceType.....	19
231	Table 10 – ConditionType definition.....	21
232	Table 11 – ConditionType Additional Subcomponents.....	21
233	Table 12 – ConditionId SimpleAttributeOperand Illustration.....	24
234	Table 13 – Disable result codes.....	24
235	Table 14 – Disable Method AddressSpace definition.....	25
236	Table 15 – Enable result codes.....	25
237	Table 16 – Enable Method AddressSpace definition.....	25
238	Table 17 – AddComment arguments.....	26
239	Table 18 – AddComment result codes.....	26
240	Table 19 – AddComment Method AddressSpace definition.....	26
241	Table 20 – ConditionRefresh parameters.....	27
242	Table 21 – ConditionRefresh result codes.....	27
243	Table 22 – ConditionRefresh Method AddressSpace definition.....	28
244	Table 23 – ConditionRefresh2 parameters.....	28
245	Table 24 – ConditionRefresh2 result codes.....	29
246	Table 25 – ConditionRefresh2 Method AddressSpace definition.....	29
247	Table 26 – DialogConditionType definition.....	30
248	Table 27 – DialogConditionType Additional Subcomponents.....	30
249	Table 28 – Respond parameters.....	31
250	Table 29 – Respond Result Codes.....	31
251	Table 30 – Respond Method AddressSpace definition.....	32
252	Table 31 – Respond2 parameters.....	32
253	Table 32 – Respond2 Result Codes.....	32
254	Table 33 – Respond2 Method AddressSpace definition.....	32
255	Table 34 – AcknowledgeableConditionType definition.....	33
256	Table 35 – AcknowledgeableConditionType Additional Subcomponents.....	33
257	Table 36 – Acknowledge parameters.....	34
258	Table 37 – Acknowledge result codes.....	34
259	Table 38 – Acknowledge Method AddressSpace definition.....	35
260	Table 39 – Confirm Method parameters.....	35
261	Table 40 – Confirm result codes.....	35
262	Table 41 – Confirm Method AddressSpace definition.....	36
263	Table 42 – AlarmConditionType definition.....	38
264	Table 43 – AlarmConditionType Additional Subcomponents.....	39
265	Table 44 – AlarmGroupType definition.....	42
266	Table 45 – AlarmSuppressionGroupType definition.....	42
267	Table 46 – Reset result codes.....	43
268	Table 47 – Reset Method AddressSpace definition.....	43
269	Table 48 – Reset2 Method parameters.....	43
270	Table 49 – Reset2 result codes.....	44

271	Table 50 – Reset2 Method AddressSpace definition.....	44
272	Table 51 – Silence result codes.....	44
273	Table 52 – Silence Method AddressSpace definition	45
274	Table 53 – Suppress result codes.....	45
275	Table 54 – Suppress Method AddressSpace definition	45
276	Table 55 – Suppress2 Method parameters	46
277	Table 56 – Suppress2 Method AddressSpace definition	46
278	Table 57 – Unsuppress result codes	47
279	Table 58 – Unsuppress Method AddressSpace definition	47
280	Table 59 – Unsuppress2 Method parameters	47
281	Table 60 – Unsuppress2 Method AddressSpace definition	48
282	Table 61 – RemoveFromService result codes	48
283	Table 62 – RemoveFromService Method AddressSpace definition.....	48
284	Table 63 – RemoveFromService2 Method parameters	49
285	Table 64 – RemoveFromService2 result codes	49
286	Table 65 – RemoveFromService2 Method AddressSpace definition.....	49
287	Table 66 – PlaceInService result codes	50
288	Table 67 – PlaceInService Method AddressSpace definition.....	50
289	Table 68 – PlaceInService2 Method parameters	50
290	Table 69 – PlaceInService2 result codes	50
291	Table 70 – PlaceInService2 Method AddressSpace definition.....	51
292	Table 71 – GetGroupMemberships result codes.....	51
293	Table 72 – GetGroupMemberships Method AddressSpace definition	51
294	Table 73 – ShelvedStateMachineType definition	53
295	Table 74 – ShelvedStateMachineType Additional References.....	54
296	Table 76 – Unshelve result codes	55
297	Table 77 – Unshelve Method AddressSpace definition	55
298	Table 78 – Unshelve2 Method parameters	55
299	Table 79 – Unshelve2 result codes	56
300	Table 80 – Unshelve2 Method AddressSpace definition	56
301	Table 81 – TimedShelve parameters.....	56
302	Table 82 – TimedShelve result codes	56
303	Table 83 – TimedShelve Method AddressSpace definition.....	57
304	Table 84 – TimedShelve2 parameters.....	57
305	Table 85 – TimedShelve2 result codes	57
306	Table 86 – TimedShelve2 Method AddressSpace definition.....	58
307	Table 87 – OneShotShelve result codes	58
308	Table 88 – OneShotShelve Method AddressSpace definition.....	58
309	Table 89 – OneShotShelve2 Method parameters	59
310	Table 90 – OneShotShelve2 result codes	59
311	Table 91 – OneShotShelve2 Method AddressSpace definition.....	59
312	Table 92 – LimitAlarmType definition	61

313	Table 93 – ExclusiveLimitStateMachineType definition.....	63
314	Table 94 – ExclusiveLimitStateMachineType Additional References	64
315	Table 96 – ExclusiveLimitAlarmType definition.....	65
316	Table 97 – NonExclusiveLimitAlarmType definition	66
317	Table 98 – NonExclusiveLimitAlarmType Additional Subcomponents.....	67
318	Table 99 – NonExclusiveLevelAlarmType definition.....	67
319	Table 100 – ExclusiveLevelAlarmType definition.....	68
320	Table 101 – NonExclusiveDeviationAlarmType definition.....	68
321	Table 102 – ExclusiveDeviationAlarmType definition.....	69
322	Table 103 – NonExclusiveRateOfChangeAlarmType definition	70
323	Table 104 – ExclusiveRateOfChangeAlarmType definition	70
324	Table 105 – DiscreteAlarmType definition.....	71
325	Table 106 – OffNormalAlarmType definition	71
326	Table 107 – SystemOffNormalAlarmType definition.....	72
327	Table 108 – TripAlarmType definition	72
328	Table 109 – InstrumentDiagnosticAlarmType definition	72
329	Table 110 – SystemDiagnosticAlarmType definition	73
330	Table 111 – CertificateExpirationAlarmType definition.....	73
331	Table 112 – DiscrepancyAlarmType definition.....	73
332	Table 113 – BaseConditionClassType definition.....	75
333	Table 114 – ProcessConditionClassType definition	75
334	Table 115 – MaintenanceConditionClassType definition	75
335	Table 116 – SystemConditionClassType definition	76
336	Table 117 – SafetyConditionClassType definition.....	76
337	Table 118 – HighlyManagedAlarmConditionClassType definition	76
338	Table 119 – TrainingConditionClassType definition	77
339	Table 120 – StatisticalConditionClassType definition	77
340	Table 121 – TestingConditionClassType definition	77
341	Table 122 – AuditConditionEventType definition	78
342	Table 123 – AuditConditionEnableEventType definition.....	79
343	Table 124 – AuditConditionCommentEventType definition.....	79
344	Table 125 – AuditConditionRespondEventType definition.....	79
345	Table 126 – AuditConditionAcknowledgeEventType definition	80
346	Table 127 – AuditConditionConfirmEventType definition	80
347	Table 128 – AuditConditionShelvingEventType definition	80
348	Table 129 – AuditConditionSuppressionEventType definition	81
349	Table 130 – AuditConditionSilenceEventType definition	81
350	Table 131 – AuditConditionResetEventType definition	81
351	Table 132 – AuditConditionOutOfServiceEventType definition	81
352	Table 133 – RefreshStartEventType definition	82
353	Table 134 – RefreshEndEventType definition.....	82
354	Table 135 – RefreshRequiredEventType definition.....	83

355	Table 136 – HasCondition <i>ReferenceType</i> Definition.....	83
356	Table 137 – Alarm & Condition result codes.....	84
357	Table 138 – HasEffectDisable <i>ReferenceType</i>	88
358	Table 139 – HasEffectEnable <i>ReferenceType</i>	89
359	Table 140 – HasEffectSuppressed <i>ReferenceType</i>	89
360	Table 141 – HasEffectUnsuppressed <i>ReferenceType</i>	90
361	Table 142 – AlarmStateVariableType definition	92
362	Table 143 – AlarmMask values	92
363	Table 144 – AlarmMask definition	92
364	Table 145 – AlarmMetricsType Definition	93
365	Table 146 – AlarmRateVariableType Definition	94
366	Table 147 – Suppress result codes	94
367	Table 148 – Reset Method AddressSpace Definition	94
368	Table A.1 – Recommended state names for LocaleId “en”	95
369	Table A.2 – Recommended DisplayNames for LocaleId “en”	95
370	Table A.3 – Recommended state names for LocaleId “de”	96
371	Table A.4 – Recommended DisplayNames for LocaleId “de”	96
372	Table A.5 – Recommended state names for LocaleId “fr”	96
373	Table A.6 – Recommended DisplayNames for LocaleId “fr”	96
374	Table A.7 – Recommended dialog response options	97
375	Table B.1 – Example of a Condition that only keeps the latest state	98
376	Table B.2 – Example of a <i>Condition</i> that maintains previous states via branches	99
377	Table B.3 – Example of a Condition that is Suppressed / OutOfService	101
378	Table C.1 – EEMUA Terms.....	105
379	Table D.1 – Mapping from standard Event categories to OPC UA Event types	107
380	Table D.2 – Mapping from ONEVENTSTRUCT fields to UA BaseEventType Variables	109
381	Table D.3 – Mapping from ONEVENTSTRUCT fields to UA AuditEventType Variables	109
382	Table D.4 – Mapping from ONEVENTSTRUCT fields to UA AlarmType Variables	110
383	Table D.5 – Event category attribute mapping table	113
384	Table F.1 – SystemStateStateMachineType definition	127
385	Table F.2 – SystemStateStateMachineType additional references	128
386		
387		

INTERNATIONAL ELECTROTECHNICAL COMMISSION

OPC UNIFIED ARCHITECTURE –

Part 9: Alarms and Conditions

FOREWORD

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

2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.

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

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

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

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

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

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

9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC 62541-9 has been prepared by subcommittee 65E: Devices and integration in enterprise systems, of IEC technical committee 65: Industrial-process measurement, control and automation.

This fourth edition cancels and replaces the third edition published in 2020. This edition constitutes a technical revision.

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

- a) Added "Comment" parameter to Alarm shelving methods.
- b) Added method that allows a client to get the members of a group, since the AddressSpace might not expose instances of alarms.
- c) Added Deadband properties for all limits in the limit AlarmType (from which all other types described in this issue are derived).
- d) Added text explaining the disabling of alarms is no longer supported in ISA 18.2 and that it is maintained in this specification for backward compatibility, but that it is recommended that Alarm not be disabled.
- e) Added optional severities for limit alarms.

- f) Added new AlarmState variable type that can be used to collect alarm information for displays on graphics.
- g) Added SupportsFilterRetain property to improve Client filtering.
- h) Removed ConditionSubClassId and ConditionSubClassNames from the conditiontype definition since they are now defined in BaseEventType.
- i)

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

CDV	Report on voting
65E/XX/CDV	65E/XX/RVC

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

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

Throughout this document and the other parts of the IEC 62541 series, certain document conventions are used:

Italics are used to denote a defined term or definition that appears in the "Terms and definition" clause in one of the parts of the IEC 62541 series.

Italics are also used to denote the name of a service input or output parameter or the name of a structure or element of a structure that are usually defined in tables.

The *italicized terms and names* are, with a few exceptions, written in camel-case (the practice of writing compound words or phrases in which the elements are joined without spaces, with each element's initial letter capitalized within the compound). For example, the defined term is *AddressSpace* instead of Address Space. This makes it easier to understand that there is a single definition for *AddressSpace*, not separate definitions for Address and Space.

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

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

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

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

OPC Unified Architecture Specification

Part 9: Alarms & Conditions

1 Scope

This document 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. Other aspects of alarm systems like alarm philosophy, life cycle, alarm response times, alarm types and many other details are captured in standards such as IEC 62682 and ISA 18.2. The *Alarms and Conditions Information Model* in this specification, is designed in accordance with IEC 62682 and ISA 18.2. Annex C specifies how the model described in this document maps to EEMUA. Annex D specifies a recommended mapping between OPC Classic Alarm & Events (A&E) servers to the model described in this document.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application.

IEC 62541-1: OPC UA Specification: Part 1 – Concepts

IEC 62541-3: OPC UA Specification: Part 3 – Address Space Model

IEC 62541-4: OPC UA Specification: Part 4 – Services

IEC 62541-5: OPC UA Specification: Part 5 – Information Model

IEC 62541-6: OPC UA Specification: Part 6 – Mappings

IEC 62541-7: OPC UA Specification: Part 7 – Profiles

IEC 62541-8: OPC UA Specification: Part 8 – Data Access

IEC 62541-11: OPC UA Specification: Part 11 – Historical Access

IEC 62541-16: OPC UA Specification: Part 16 – State Machines

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

<https://www.eemua.org/Products/Publications/Print/EEMUA-Publication-191.aspx>

IEC 62682: Management of alarm systems for the process industries (Edition 1.0 2014-10)

<https://webstore.iec.ch/publication/7363>

ISA 18.2: Management of Alarm Systems for the Process Industries

<https://webstore.ansi.org/Standards/ISA/ansiisa182016>

IETF RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One

<https://www.ietf.org/rfc/rfc2045.txt>

IETF RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two

<https://www.ietf.org/rfc/rfc2046.txt>

IETF RFC2047: Multipurpose Internet Mail Extensions (MIME) Part Three

<https://www.ietf.org/rfc/rfc2047.txt>

3 Terms, definitions, and abbreviations

3.1 Terms and definitions

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

3.1.1

Acknowledge

Operator action that indicates recognition of an *Alarm*

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

3.1.2

Active

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

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

3.1.3

AdaptiveAlarm

Alarm for which the set point or limits are changed by an algorithm.

Note 1 to entry: *AdaptiveAlarms* are alarms that are adjusted automatically by algorithms. These algorithms might detect conditions in a plant and change setpoints or limits to keep alarms from occurring. These changes occur, in many cases, without *Operator* interactions.

3.1.4

AlarmFlood

condition during which the alarm rate is greater than the *Operator* can effectively manage

Note 1 to entry: OPC UA does not define the conditions that would be considered alarm flooding, these conditions are defined in other specification such as IEC 62682 or ISA 18.2

3.1.5

AlarmManager

application that manages alarms in a system

Note 1 An *AlarmManager* usually includes higher level functionality like intelligent alarm suppression or dynamic adjustment of limits or other alarm management functionality. It will still act on the alarm model associated with the actual alarm source.

3.1.6

AlarmSuppressionGroup

group of *Alarms* that is used to suppress other *Alarms*.

Note 1 to entry: An *AlarmSuppressionGroup* is an instance of an *AlarmGroupType* that is used to suppress other *Alarms*. If any *Alarm* in the group is active, then the *AlarmSuppressionGroup* is active. If all *Alarms* in the *AlarmSuppressionGroup* are inactive then the *AlarmSuppressionGroup* is inactive.

Note 2 to entry: The *Alarm* to be affected references *AlarmSuppressionGroups* with a *HasAlarmSuppressionGroupReferenceType*.

3.1.7

ConditionClass

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

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

3.1.8

ConditionBranch

specific state of a *Condition*

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

3.1.9**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*.

3.1.10**Confirm**

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

3.1.11**Disable**

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

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

In IEC 62682 disable has been replaced with "Out of Service".

3.1.12**LatchingAlarm**

alarm that remains in alarm state after the process condition has returned to normal and requires an *Operator* reset before the alarm returns to normal

Note 1 to entry: Latching alarms are typically discrepancy alarms, where an action does not occur within a specific time. Once the action occurs the alarm stays active until it is reset.

3.1.13**Operator**

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

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

3.1.14**Refresh**

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

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

3.1.15**Retain**

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

3.1.16**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.

3.1.17**Suppress**

act of determining whether an *Alarm* should not occur

Note 1 to entry: "An *Alarm* is suppressed when logical criteria are applied to determine that the *Alarm* should not occur, even though the base *Alarm Condition* (e.g. *Alarm setting exceeded*) is present" as defined in EEMUA. In IEC 62682 Suppressed *Alarms* are also described as being "Suppressed by Design", in that the system is designed with logic to Suppress an *Alarm* when certain criteria exist. For example, if a process unit is taken off line then low level alarms are Suppressed for all equipment in the off-line unit.

3.2 Abbreviations and symbols

A&E	Alarm & Event (as used for OPC COM)
COM	(Microsoft Windows) Component Object Model
DA	Data Access
MIME	Multipurpose Internet Mail Extensions
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 given in Table 1. The base types and data types defined in IEC 62541-4 are given in Table 2.

Table 1 – Parameter types defined in IEC 62541-3

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

Table 2 – Parameter types defined in IEC 62541-4

Parameter Type
IntegerId
StatusCode

4 Concepts

4.1 General

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

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 5.5 provides more information and use cases. Maintaining previous states and therefore 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.

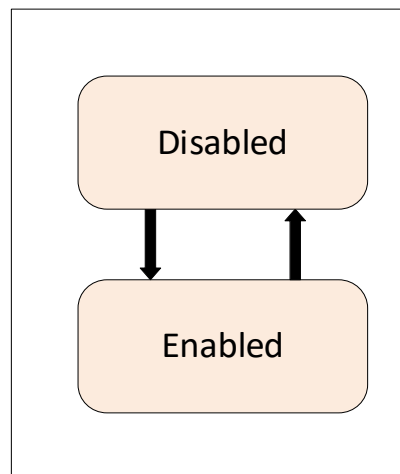


Figure 1 – Base Condition state model

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.

ISA 18.2 and IEC 62682, which are the basis for this information model, no longer support enabling and disabling of alarms. As a result, even though this feature is still supported for backward compatibility it is recommended that alarms never be disabled.

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

If *Auditing* is supported by a *Server*, the following *Auditing* related action shall be performed. The *Server* will generate *AuditEvents* for *Enable* and *Disable* operations (either through a *Method* call or some *Server* / vendor – specific means), rather than generating an *AuditEvent Notification* for each *Condition* instance being enabled or disabled. For more information, see

the definition of *AuditConditionEnableEventType* in 5.10.2. *AuditEvents* are also generated for any other *Operator* action that results in changes to the *Conditions*.

4.3 Acknowledgeable Conditions

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

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

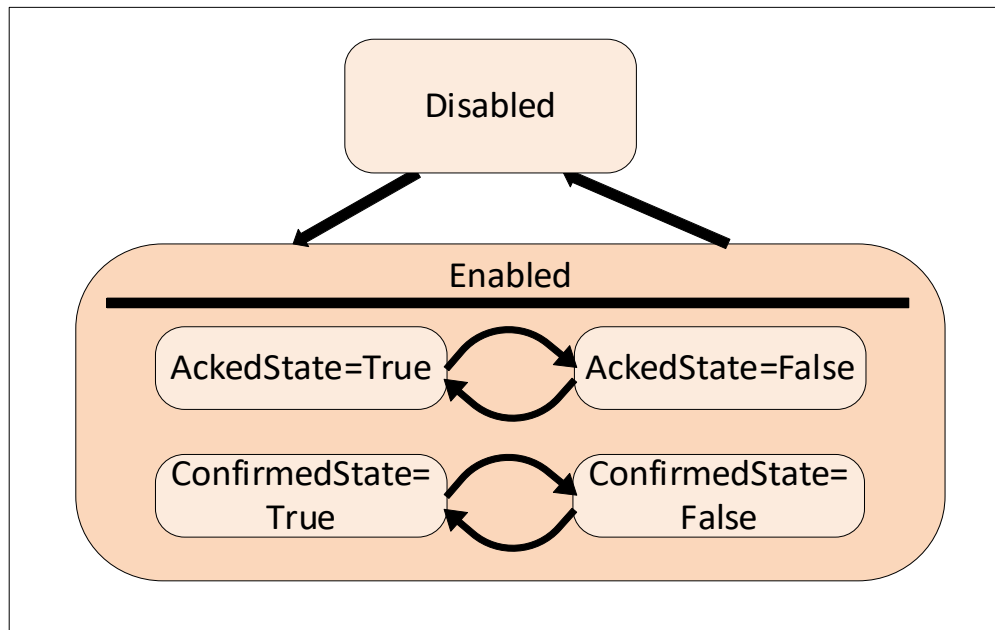


Figure 2 – AcknowledgeableConditions state model

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

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

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

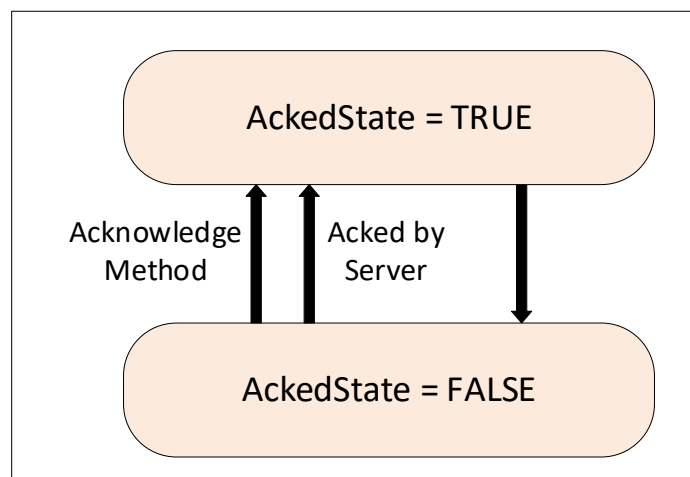


Figure 3 - Acknowledge State Model

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

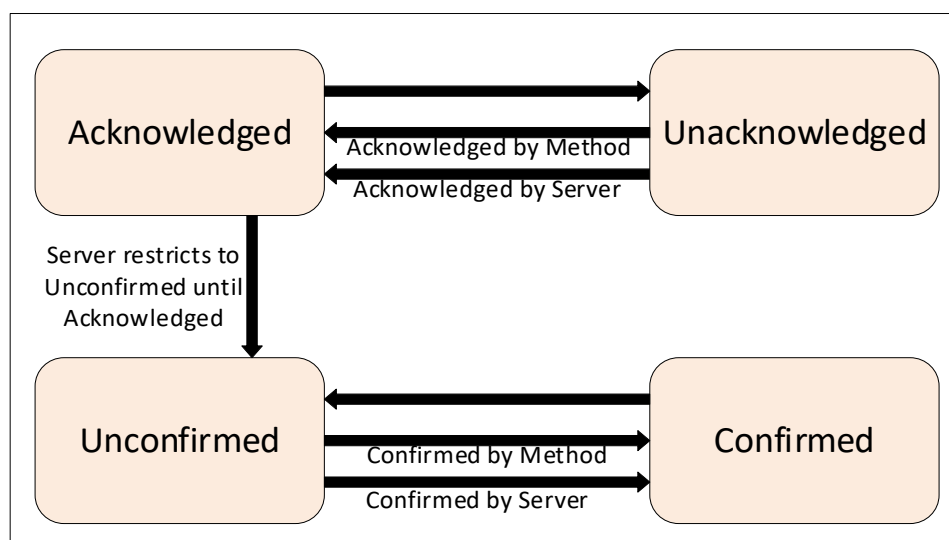


Figure 4 - Confirm acknowledge State model

The *AcknowledgeableConditions* model can also be used for a simple *Condition* that requires a confirmation. For example, a motor start event might need to be acknowledged by an operator, this event is not an alarm, just an action that needs to be acknowledged by the operator. It could also be an event that is generated for an operator change of a process input (grade of paper being made) that needs to be acknowledged. It is not something that requires an answer like a question, for this there are *DialogCondition*. It is just an acknowledge that the operator observed the change.

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* maintains *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 *RefreshStartEventType Event*. 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 *RefreshStartEventType*. The *Client* adds any new *Events* that are received during the *Refresh* without flagging them as suspect. The *Client* also removes the suspect flag from any *Retained Conditions* that are returned as part of the *Refresh*. When the *Client* receives a *RefreshEndEvent*, the *Client* removes any remaining suspect *Events*, since they no longer apply.

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

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

ignore the *Event* content filtering associated with a *Subscription* and will always be delivered to the *Client*.

- ConditionRefresh applies to a *Subscription*. If multiple Event Notifiers are included in the same *Subscription*, all *Event Notifiers* are refreshed.

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 *ProcessConditionClassType*.

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 "Level Alarm" *Condition* is also questionable. If more than one variable is represented by a given condition or if the condition is from an underlining system and no direct mapping to a variable is available, it is up to the application to determine what quality is displayed as part of the condition.

4.7 Dialogs

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

4.8 Alarms

Alarms are specializations of *AcknowledgeableConditions* that add the concepts of an *Active* state and other states like *Shelving* state and *Suppressed* state to a *Condition*. The state model is illustrated in Figure 5. The complete model with all states is defined in 5.8.

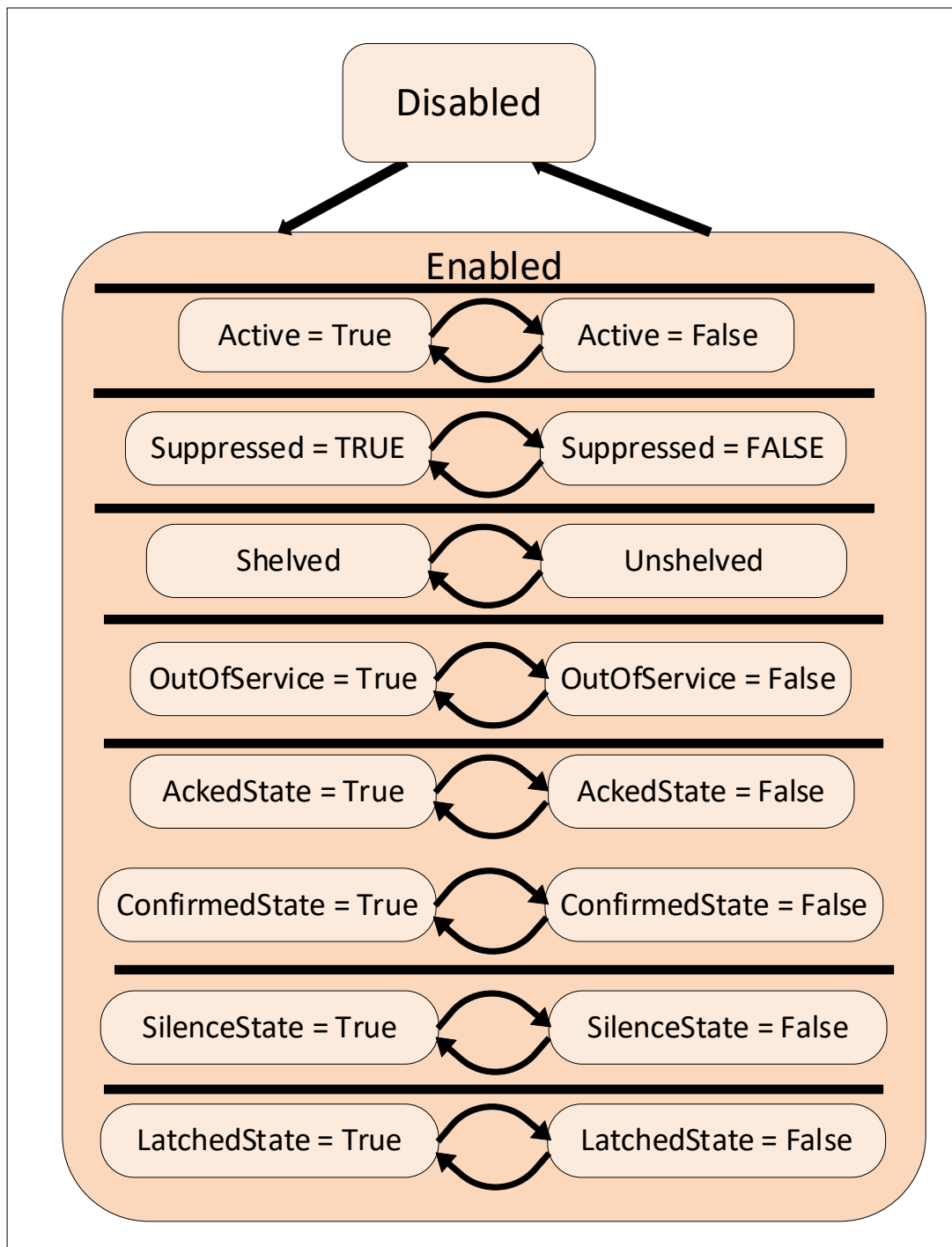


Figure 5 – Alarm state machine model

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, out of service 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 *LatchedState* is a state that is added to any alarm that requires additional processing, once it goes active it does not clear until it is explicitly reset, even if the value that triggered the alarm returns to normal. This might be an alarm that requires a physical inspection once it has occurred to ensure it is no longer in alarm, or a series of tests that might be required to ensure the alarm is no longer active or some other actions. Any *AlarmType* can become a latched alarm by the addition of this optional sub-state.

The shelved, out of service 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*. A disabled Alarm is not processed in any manner, and is not supported as part of the active Alarm model defined in ISA 18.2.

Alarms follow a typical timeline that is illustrated in Figure 6. They have a number of delay times associated with them and a number of states that they might occupy. The goal of an alarming system is to inform *Operators* about conditions in a timely manner and allow the *Operator* to take some action before some consequences occur. The consequences may be economic (product is not usable and must be discard), may be physical (tank overflows), may safety (fire or explosion could occur) or any of a number of other possibilities. Typically, if no action is taken related to an alarm for some period of time the process will cross some threshold at which point consequences will start to occur. The OPC UA *Alarm* model describes these states, delays and actions.

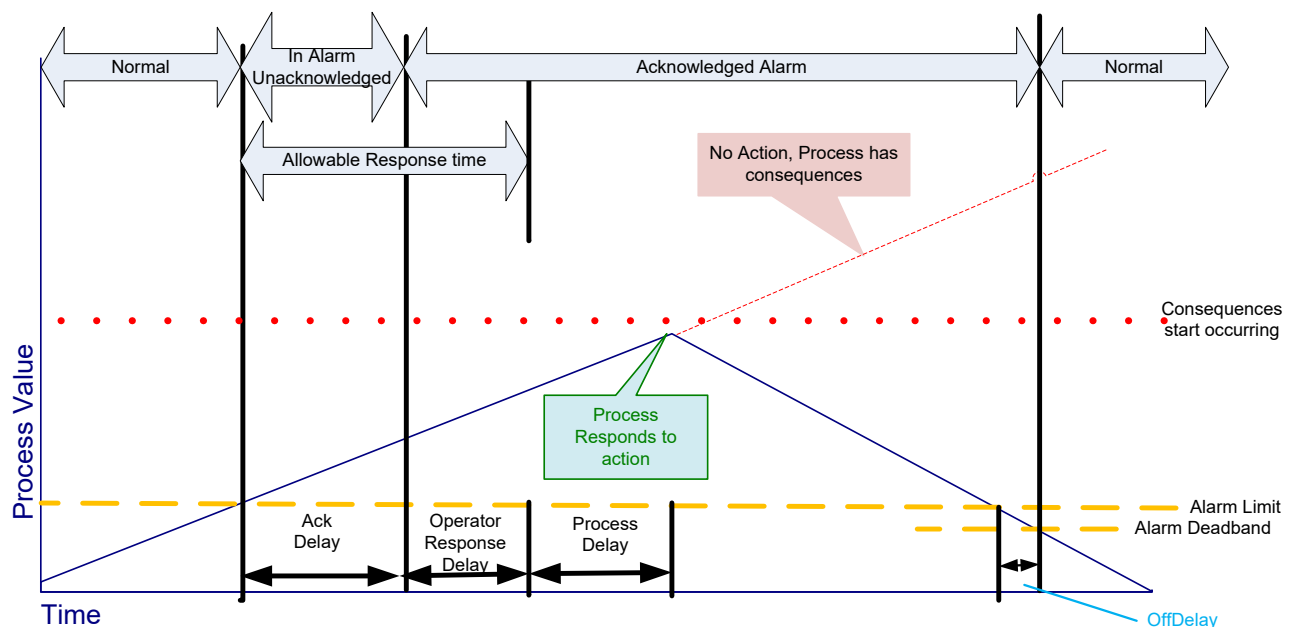


Figure 6 – Typical Alarm Timeline example

4.9 Multiple active states

In some cases, it is desirable to further define the *Active* state of an *Alarm* by providing a sub-state machine for the *Active* State. For example, a multi-state level *Alarm* when in the *Active* state may be in one of the following sub-states: LowLow, Low, High or HighHigh. The state model is illustrated in Figure 7.

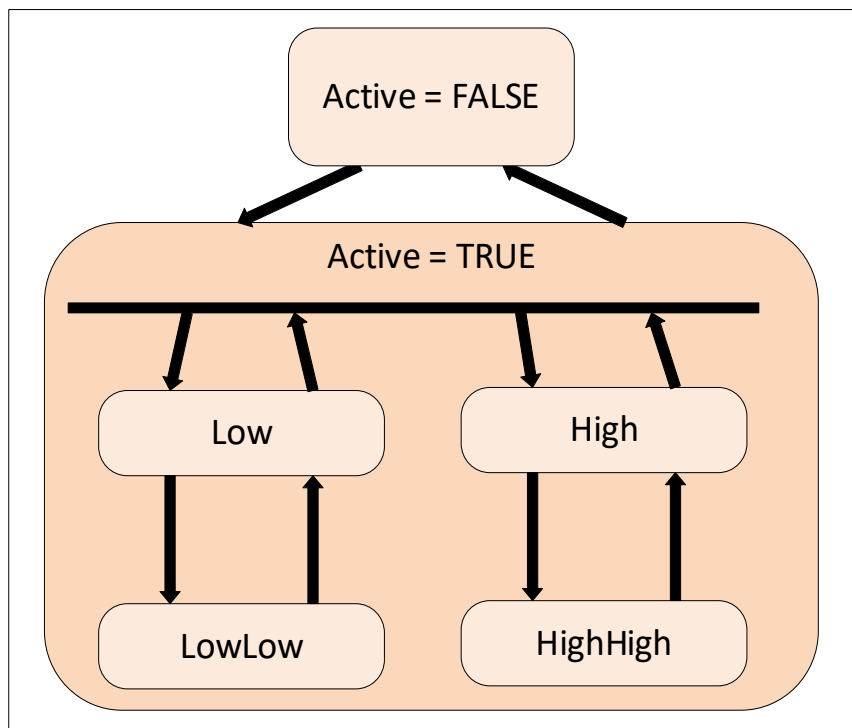


Figure 7 – Multiple active states example

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 level *Alarm* 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 level *Alarm* will be both in the High and the HighHigh sub-state.

4.10 Condition instances in the AddressSpace

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 *LimitAlarmType*.

The availability of instances allows Data Access *Clients* to monitor the current *Condition* state by subscribing to the *Attribute* values of *Variable Nodes*. The values of the nodes may not always correspond with the value that appear in *Events*, they may be more recent than what was in the *Event*.

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

4.11 Alarm and Condition auditing

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

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

This standard describes the *AuditEventType* that each *Method* shall generate if Audit Events are supported by the *Server*. For example, the *Disable Method* has an *AlwaysGeneratesEvent Reference* to an *AuditConditionEnableEventType*. An *Event* of this type shall be generated for every invocation of the *Method* if audit events are supported. 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.

4.12 Alarms in a system

In a system, alarms might be managed at different levels and by different applications. An alarm might be detected in an instrument, but the full alarm model for that alarm instance might be maintained in a higher level *Server*. This can cause issues in cases where the instrument is restarted or the UA Server that is acting as an alarm server is restarted. It is desirable that the state of alarms is maintained and that after any restart of either application all alarms recover their same state as before the restart. But it is acceptable if an alarm might require some management actions again following a restart. It is not acceptable that a device that is in alarm is no longer reported as being in alarm. For example, if the *AlarmManager* is restarted when it recovers it is able to detect that the device has an alarm, but it might not be able to recover that the alarm has a comment that had been entered as part of an alarm acknowledgement or even that it was acknowledged. The details of what states are expected to be recovered are provided in the individual object models.

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*. Instances not exposed in the *AddressSpace* still have a *ConditionId (NodeId)*. This *NodeId* can be the target of references in the *AddressSpace* even though the target node does not exist in the *AddressSpace*. For example, an *AlarmGroup* might reference a number of *Conditions* that do not exist in the *AddressSpace*.

The following sub clauses define the OPC UA *Alarm* and *Condition Types*. Figure 8 informally describes the hierarchy of these *Types*. Subtypes of the *LimitAlarmType* and the *DiscreteAlarmType* are not shown.

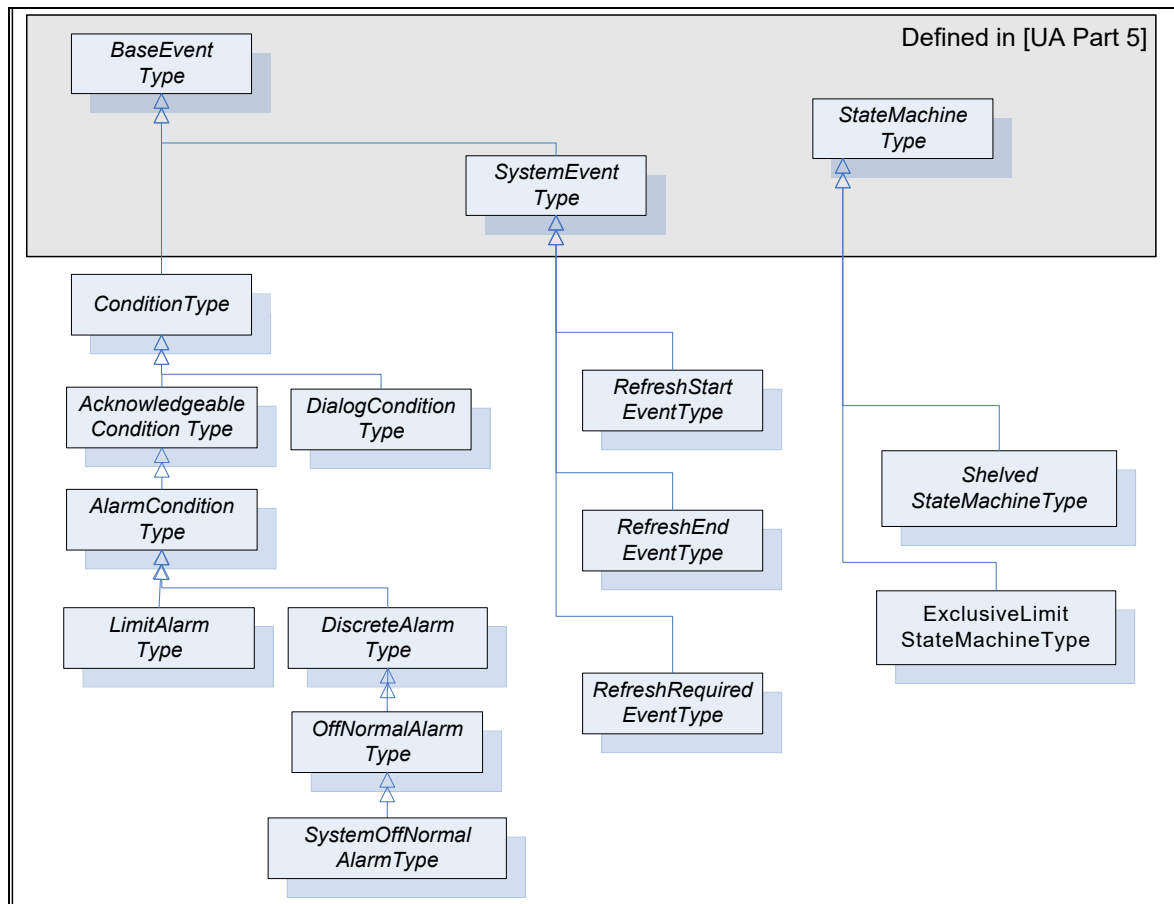


Figure 8 – ConditionType hierarchy

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 62541-16.

The *TwoStateVariableType* is derived from the *StateVariableType* defined in IEC 62541-16. 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 62541-16. 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
ConformanceUnits					
A & C Basic					

933 The *Value Attribute* of an instance of *TwoStateVariableType* contains the current state as a
934 human readable name. The *EnabledState* for example, might contain the name “Enabled” when
935 True and “Disabled” when False.

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

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

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

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

948 Other optional *Properties* of the *StateVariableType* have no defined meaning for
949 *TwoStateVariableType*.

950 *TrueState* and *FalseState* contain the localized string for the *TwoStateVariableType* value when
951 its *Id Property* has the value True or False, respectively. Since the two *Properties* provide meta-
952 data for the *Type*, *Servers* shall not allow these *Properties* to be selected in the *Event* filter for
953 a *MonitoredItem*. The *TrueState Property* and *FalseState Property* shall only exist on
954 *InstanceDeclarations*. See Figure 9 for an illustration. *Clients* can use the Read Service to get
955 the values of the *TrueState* and *FalseState Property*.

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

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

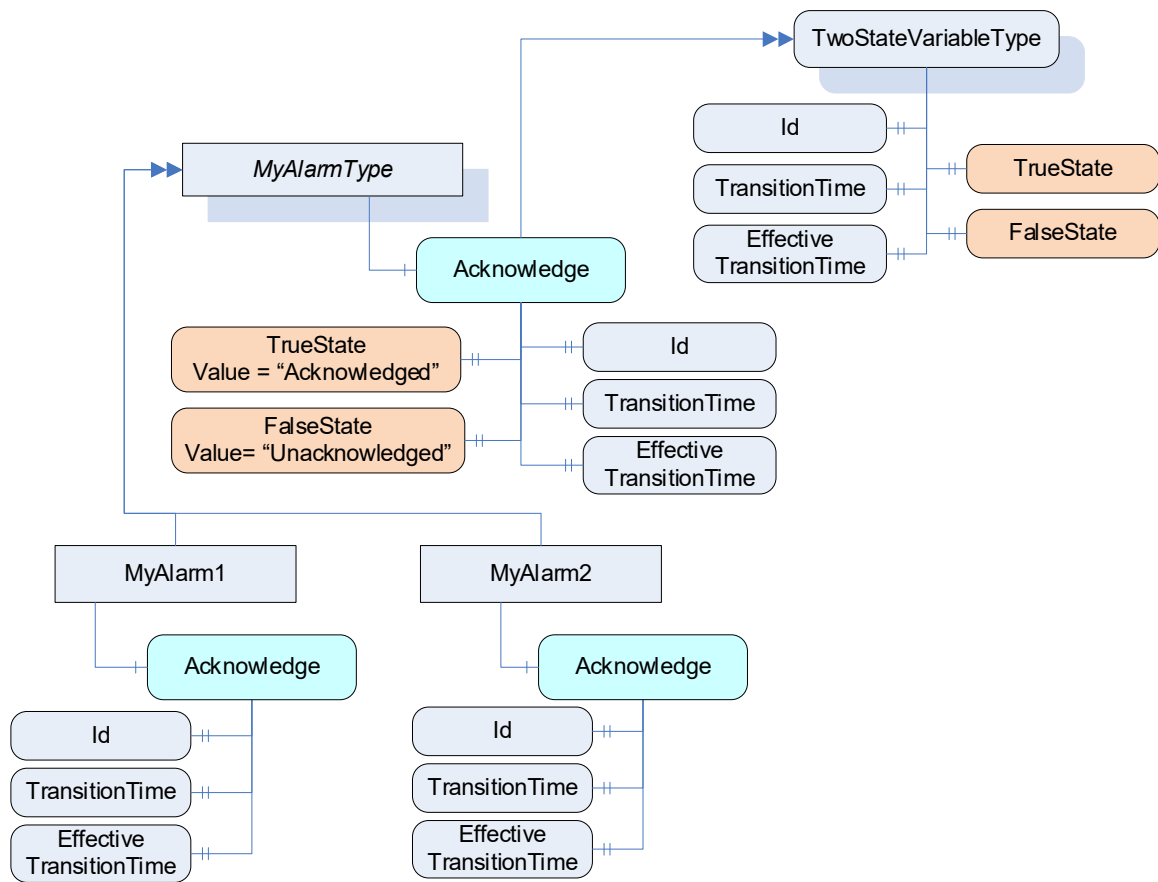


Figure 9 - TwoStateVariable Illustration

5.3 ConditionVariable

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 *ConditionVariable*.

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
ConformanceUnits					
A & C Basic					

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 ReferenceTypes

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. This includes extending *TwoStateVariableType* state machines with sub states and the addition of *Alarm* grouping.

The *TwoStateVariableType References* will only exist when sub states are available. For example, if a *TwoStateVariableType* machine is in a False State, then any sub states 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 sub state is part of the data that is to be reported than this data would be reported as a NULL. With this approach, *TwoStateVariableTypes* can be extended with subordinate state machines in a similar fashion to the *StateMachineType* defined in IEC 62541-16.

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 sub states. 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 be either 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
ConformanceUnits			
A & C Basic			

5.4.3 HasFalseSubState ReferenceType

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

The semantics indicate that the sub state (the target *Node*) is a subordinate state of the False super state. If more than one state within a *Condition* is a sub state of the same super state (i.e. several *HasFalseSubState References* exist for the same super state) they are all treated as independent sub states. 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 be either an instance of a *TwoStateVariableType* or an instance of a subtype of a *StateMachineType*.

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

Table 6 – HasFalseSubState ReferenceType

Attributes	Value		
BrowseName	HasFalseSubState		
InverseName	IsFalseSubStateOf		
Symmetric	False		
IsAbstract	False		
References	NodeClass	BrowseName	Comment
ConformanceUnits			
A & C Basic			

5.4.4 HasAlarmSuppressionGroup ReferenceType

The *HasAlarmSuppressionGroup ReferenceType* is a concrete *ReferenceType* that can be used directly. It is a subtype of the *HasComponent ReferenceType*. The representation in the *AddressSpace* is specified in Table 7

This *ReferenceType* binds an *AlarmSuppressionGroup* to an *Alarm*.

The *SourceNode* of the *Reference* shall be an instance of an *AlarmConditionType* or sub type. The *TargetNode* shall be an instance of an *AlarmGroupType*.

Table 7 – HasAlarmSuppressionGroup ReferenceType

Attributes	Value		
BrowseName	HasAlarmSuppressionGroup		
InverseName	IsAlarmSuppressionGroupOf		
Symmetric	False		
IsAbstract	False		
References	NodeClass	BrowseName	Comment
ConformanceUnits			
A & C Suppression Group			

5.4.5 AlarmGroupMember ReferenceType

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

This *ReferenceType* is used to indicate the *Alarm* instances that are part of an *Alarm Group*. The representation in the *AddressSpace* is specified in Table 8

The *SourceNode* of the *Reference* shall be an instance of an *AlarmGroupType* or sub type of it. The *TargetNode* shall be an instance of an *AlarmConditionType* or a subtype of it.

Table 8 – AlarmGroupMember ReferenceType

Attributes	Value		
BrowseName	AlarmGroupMember		
InverseName	MemberOfAlarmGroup		
Symmetric	False		
IsAbstract	False		
References	NodeClass	BrowseName	Comment
ConformanceUnits			
A & C Suppression Group			
A & C First in Group Alarm			

5.4.6 AlarmSuppressionGroupMember ReferenceType

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

1034 This *ReferenceType* is used to indicate the *Alarm* instances or *Boolean Variables* that are part
1035 of an *Alarm Group*. The representation in the *AddressSpace* is specified in Table 9

1036 The *SourceNode* of the *Reference* shall be an instance of an *AlarmGroupType* or sub type of it.
1037 The *TargetNode* shall be an instance of an *AlarmConditionType* or a subtype of it, or an instance
1038 of *BaseDataVariableType* that has a *DataType* of *Boolean*.

1039 **Table 9 – AlarmSuppressionGroupMember ReferenceType**

Attributes	Value		
BrowseName	AlarmSuppressionGroupMember		
InverseName	MemberOfAlarmSuppressionGroup		
Symmetric	False		
IsAbstract	False		
References	NodeClass	BrowseName	Comment
ConformanceUnits			
A & C Suppression Group			

1040

1041 **5.5 Condition Model**

1042 **5.5.1 General**

1043 The *Condition* model extends the *Event* model by defining the *ConditionType*. The
1044 *ConditionType* introduces the concept of states differentiating it from the base *Event* model.
1045 Unlike the *BaseEventType*, *Conditions* are not transient. The *ConditionType* is further extended
1046 into *Dialog* and *AcknowledgeableConditionType*, each of which has their own sub-types.

1047 The *Condition* model is illustrated in Figure 10 and formally defined in the subsequent tables. It
1048 is worth noting that this figure, like all figures in this document, is not intended to be complete.
1049 Rather, the figures only illustrate information provided by the formal definitions.

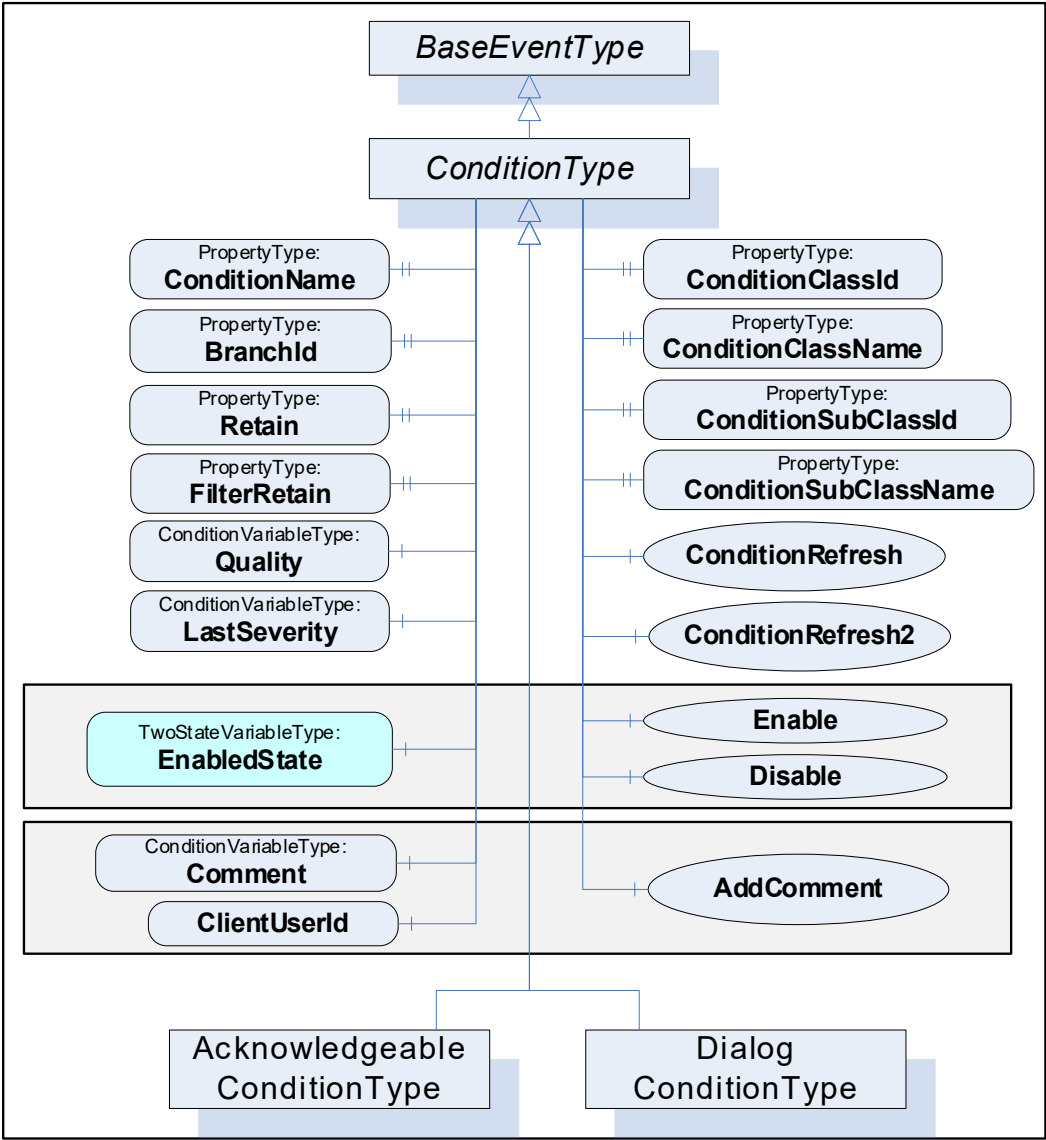


Figure 10 – Condition model

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 10 and Table 11. The False state of the *EnabledState* shall not be extended with a sub state machine.

1056

Table 10 – ConditionType definition

Attribute	Value				
BrowseName	ConditionType				
IsAbstract	True				
References	NodeClass	BrowseName	DataType	TypeDefinition	Modelling Rule
Subtype of the <i>BaseEventType</i> defined in IEC 62541-5					
HasSubtype	ObjectType	DialogConditionType	Defined in Clause 5.6.2		
HasSubtype	ObjectType	AcknowledgeableConditionType	Defined in Clause 5.7.2		
HasProperty	Variable	ConditionClassId	NodeId	PropertyType	Mandatory
HasProperty	Variable	ConditionClassName	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	ConditionName	String	PropertyType	Mandatory
HasProperty	Variable	BranchId	NodeId	PropertyType	Mandatory
HasProperty	Variable	Retain	Boolean	PropertyType	Mandatory
HasProperty	Variable	SupportsFilteredRetain	Boolean	PropertyType	
HasComponent	Variable	EnabledState	LocalizedText	TwoStateVariableType	Mandatory
HasComponent	Variable	Quality	StatusCode	ConditionVariableType	Mandatory
HasComponent	Variable	LastSeverity	UInt16	ConditionVariableType	Mandatory
HasComponent	Variable	Comment	LocalizedText	ConditionVariableType	Mandatory
HasProperty	Variable	ClientUserId	String	PropertyType	Mandatory
HasComponent	Method	Disable	Defined in Clause 5.5.4		Mandatory
HasComponent	Method	Enable	Defined in Clause 5.5.5		Mandatory
HasComponent	Method	AddComment	Defined in Clause 5.5.6		Mandatory
HasComponent	Method	ConditionRefresh	Defined in Clause 5.5.7		
HasComponent	Method	ConditionRefresh2	Defined in Clause 5.5.8		
ConformanceUnits					
A & C Basic					

1057

1058

Table 11 – ConditionType Additional Subcomponents

BrowsePath	References	NodeClass	BrowseName	DataType	TypeDefinition	Others
EnabledState	HasProperty	Variable	TrueState	LocalizedText	PropertyType	
EnabledState	HasProperty	Variable	FalseState	LocalizedText	PropertyType	

1059

1060

The empty “Others” column indicates that no *ModellingRule* applies.

1061

1062

1063

1064

1065

1066

1067

1068

1069

1070

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

1071

1072

1073

1074

1075

ConditionClassId, *ConditionClassName*, *ConditionSubClassId* and *ConditionSubClassName* originally defined in *ConditionType* are now defined in the *BaseEventType* (from which this type is derived). They are optional in the *BaseEventType*, but *ConditionClassId*, and *ConditionClassName* are *Mandatory* in *ConditionType* and thus listed (to update the modelling rule).

1076

1077

1078

1079

1080

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 name element of the *BrowseName* of the *ConditionType*.

1081

1082

BranchId is NULL for all *Event Notifications* that relate to the current state of the *Condition* instance. If *BranchId* is not NULL, it identifies a previous state of this *Condition* instance that

still needs attention by an *Operator*. If the current *ConditionBranch* is transformed into a previous *ConditionBranch* then the *Server* needs to assign a non-NULL *BranchId*. An initial *Event* for the branch will 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. The retain bit on the current *Event* is True, as long as any *ConditionBranches* require *Operator* input. See 4.2 for more information about the need for creating and maintaining previous *ConditionBranches* and Clause B.1 for an example using branches. The *BranchId* *DataType* is *NodeId* although the *Server* is not required to have *ConditionBranches* in the *AddressSpace*. The use of a *NodeId* allows the *Server* to use simple numeric identifiers, strings or arrays of bytes.

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

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

SupportsFilteredRetain *Property* is only provided on the *ConditionType*. When this *Property* is set to True on the *Type*, then the *Server* provides a *Client* specific *Retain* flag value taking into account any *Client* provided filter. When the property is False on the *ConditionType* then the *Server* does not provide a *Client* specific the value of the *Retain* flag. For example, if a *Client* applies a filter to exclude *Alarms* that are shelved, and the *SupportsFilteredRetain* is set to True, the *Client* receives an *Alarm* (it is not shelved, *Retain* is true). The *Client* (or another *Client*) shelves the *Alarm*. At this point the *Alarm* no longer passes the filter, but since the previous event was sent, this event is transmitted with *Retain* = False. For an example see B.1.4

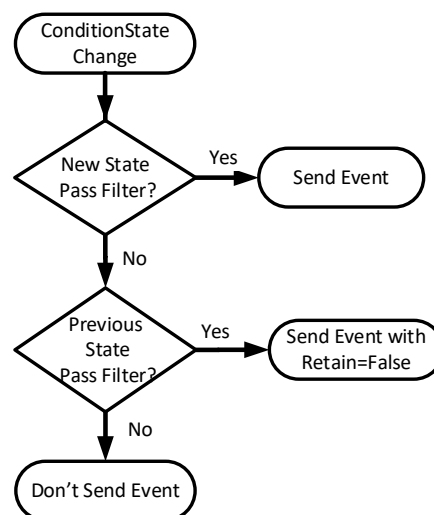


Figure 11 - SupportsFilteredRetain process

Figure 11 provides an illustration of the processing a *Server* shall follow for processing *Retain* flag when *SupportsFilteredRetain* flag is set to True.

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 A.1.

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

- 1120 • When the *Condition* instance enters the *Disabled* state, the *Retain Property* of this
1121 *Condition* shall be set to False by the *Server* to indicate to the *Client* that the *Condition*
1122 instance is currently not of interest to *Clients*. This includes all *ConditionBranches* if any
1123 branches exist.
- 1124 • When the *Condition* instance enters the enabled state, the *Condition* shall be evaluated
1125 and all of its *Properties* updated to reflect the current values. If this evaluation causes
1126 the *Retain Property* to transition to True for any *ConditionBranch*, then an *Event*
1127 *Notification* shall be generated for that *ConditionBranch*.
- 1128 • The *Server* may choose to continue to test for a *Condition* instance while it is *Disabled*.
1129 However, no *Event Notifications* will be generated while the *Condition* instance is
1130 disabled.
- 1131 • For any *Condition* that exists in the *AddressSpace* the *Attributes* and the following
1132 *Variables* will continue to have valid values even in the *Disabled* state; *EventId*, *Event*
1133 *Type*, *Source Node*, *Source Name*, *Time*, and *EnabledState*. Other *Properties* may no
1134 longer provide current valid values. All *Variables* that are no longer provided shall return
1135 a status of *Bad_ConditionDisabled*. The *Event* that reports the *Disabled* state should
1136 report the *Properties* as NULL or with a status of *Bad_ConditionDisabled*.

1137 When enabled, changes to the following *Variables* shall cause a *ConditionType Event*
1138 *Notification*:

- 1139 • Quality
- 1140 • Severity (inherited from *BaseEventType*)
- 1141 • Comment

1142 This may not be the complete list. Subtypes of *ConditionType* may define additional *Variables*
1143 that trigger *Event Notifications*. In general, changes to *Variables* of the types
1144 *TwoStateVariableType*, *ConditionVariableType*, *StateMachineType* or any of their subtypes
1145 trigger *Event Notifications* and are not explicitly described in subtypes.

1146 In the event of a restart of an *AlarmManager*, the *AlarmManager* shall recover the
1147 *Enabled/Disabled* states of all current conditions. If the system can not determine if the
1148 *Condition* is *Enabled* or *Disabled*, it shall be *Enabled*.

1149 *Quality* reveals the status of process values or other resources that this *Condition* instance is
1150 based upon. If, for example, a process value is "Uncertain", the associated "LevelAlarm"
1151 *Condition* is also questionable. Values for the *Quality* can be any of the OPC *StatusCodes*
1152 defined in IEC 62541-8 as well as *Good*, *Uncertain* and *Bad* as defined in IEC 62541-4. These
1153 *StatusCodes* are similar to but slightly more generic than the description of data quality in the
1154 various field bus specifications. It is the responsibility of the *Server* to map internal status
1155 information to these codes. A *Server* that supports no quality information shall return *Good*.
1156 This quality can also reflect the communication status associated with the system that this value
1157 or resource is based on and from which this *Alarm* was received. For communication errors to
1158 the underlying system, especially those that result in some unavailable *Event* fields, the quality
1159 shall be *Bad_NoCommunication* error.

1160 *Events* are only generated for *Conditions* that have their *Retain* field set to True and for the
1161 initial transition of the *Retain* field from True to False.

1162 *LastSeverity* provides the previous severity of the *ConditionBranch*. Initially this *Variable*
1163 contains a zero value; it will return a value only after a severity change. The new severity is
1164 supplied via the *Severity Property*, which is inherited from the *BaseEventType*.

1165 *Comment* contains the last comment provided for a certain state (*ConditionBranch*) of a
1166 *Condition*. It may have been provided by an *AddComment Method*, some other *Method* or in
1167 some other manner. Any change in this field, shall trigger a new event to be generated. The
1168 initial value of this *Variable* is NULL, unless it is provided in some other manner.

1169 *ClientUserId* is related to the *Comment* field and contains the identity of the user who inserted
1170 the most recent *Comment*. The logic to obtain the *ClientUserId* is defined in IEC 62541-5.

The *NodeId* of the *Condition* instance is used as *ConditionId*. It is not explicitly modelled as a component of the *ConditionType*. However, it can be requested with the following *SimpleAttributeOperand* (see Table 12) in the *SelectClause* of the *EventFilter*: See IEC 62541-4 for a detailed definition of the *SelectClause* in an *Event Subscription*.

Table 12 – ConditionId SimpleAttributeOperand Illustration

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

5.5.3 Condition and branch instances

Conditions are *Objects* which have a state which changes over time. Each *Condition* instance has a *ConditionId* as an identifier which uniquely identifies it within the *Server*. This *Condition* (and its representative *ConditionId*) follows the same rules associated with *Nodes* (and their representative *NodeIds*) in the *AddressSpace* (see IEC 62541-3). Therefore, once a *Condition* is created in a system that does not expose instances, any time that *Condition* is active it shall always have the same *ConditionId*. This allows higher level alarm management systems to operate on *Conditions* and to perform analysis of them.

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

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

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

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

5.5.4 Disable Method

The *Disable Method* is used to change a *Condition* instance to the *Disabled* state. Normally, the *NodeId* of the object instance as the *ObjectId* is passed to the *Call Service*. However, some *Servers* do not expose *Condition* instances in the *AddressSpace*. Therefore, all *Servers* shall allow *Clients* to call the *Disable Method* by specifying *ConditionId* as the *ObjectId*. The *Method* cannot be called with an *ObjectId* of the *ConditionType* *Node*. Since *Condition* instances may not be defined in the *AddressSpace*, the *MethodId* that is passed in the *Call Service* shall be the *NodeId* of the *Disable Method* on the *ConditionType*.

Signature

Disable () ;

Method Result Codes in Table 13 (defined in *Call Service*)

Table 13 – Disable result codes

Result Code	Description
Bad_ConditionAlreadyDisabled	See Table 137 for the description of this result code.

Table 14 specifies the *AddressSpace* representation for the *Disable Method*.

Table 14 – Disable Method AddressSpace definition

Attribute	Value				
BrowseName	Disable				
References	NodeClass	BrowseName	DataType	TypeDefinition	Modelling Rule
AlwaysGeneratesEvent	ObjectType	AuditConditionEnable EventType	Defined in 5.10.2		
ConformanceUnits					
A & C Enable					

If *Auditing* is supported, this *Method* shall generate an *Event* of *AuditConditionEnableEventType* for all invocations of the *Method*.

5.5.5 Enable Method

The *Enable Method* is used to change a *Condition* instance to the enabled state. Normally, the *NodeId* of the object instance as the *ObjectId* is passed to the *Call Service*. However, some *Servers* do not expose *Condition* instances in the *AddressSpace*. Therefore, all *Servers* shall allow *Clients* to call the *Enable Method* by specifying *ConditionId* as the *ObjectId*. The *Method* cannot be called with an *ObjectId* of the *ConditionType Node*. If the *Condition* instance is not exposed, then it may be difficult for a *Client* to determine the *ConditionId* for a disabled *Condition*. Since *Condition* instances may not be defined in the *AddressSpace*, the *MethodId* that is passed in the *Call Service* shall be the *NodeId* of the *Enable Method* on the *ConditionType*.

Signature

Enable () ;

Method result codes in Table 15 (defined in *Call Service*)

Table 15 – Enable result codes

Result Code	Description
Bad_ConditionAlreadyEnabled	See Table 137 for the description of this result code.

Table 16 specifies the *AddressSpace* representation for the *Enable Method*.

Table 16 – Enable Method AddressSpace definition

Attribute	Value				
BrowseName	Enable				
References	NodeClass	BrowseName	DataType	TypeDefinition	Modelling Rule
AlwaysGeneratesEvent	ObjectType	AuditConditionEnableEventType	Defined in 5.10.2		
ConformanceUnits					
A & C Enable					

If *Auditing* is supported, this *Method* shall generate an *Event* of *AuditConditionEnableEventType* for all invocations of the *Method*.

5.5.6 AddComment Method

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

Signature

AddComment (
[in] ByteString EventId

1245 [in] LocalizedText Comment
 1246);
 1247

1248 The parameters are defined in Table 17

1249 **Table 17 – AddComment arguments**

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

1250

1251 *Method* result codes in Table 18 (defined in Call Service)

1252 **Table 18 – AddComment result codes**

Result Code	Description
Bad_MethodInvalid	The <i>MethodId</i> provided does not correspond to the <i>ObjectId</i> provided. See IEC 62541-4 for the general description of this result code.
Bad_EventIdUnknown	See Table 137 for the description of this result code.
Bad_NodeIdInvalid	Used to indicate that the specified <i>ObjectId</i> is not valid or that the <i>Method</i> was called on the <i>ConditionType Node</i> . See IEC 62541-4 for the general description of this result code.

1253

1254 Comments

1255 *Comments* are added to *Event* occurrences identified via an *EventId*. *EventIds* where the related
 1256 *EventType* is not a *ConditionType* (or subtype of it) and thus does not support *Comments* are
 1257 rejected.

1258 A *ConditionEvent* – where the *Comment Variable* contains this text – will be sent for the
 1259 identified state. If a comment is added to a previous state (i.e. a state for which the Server has
 1260 created a branch), the *BranchId* and all *Condition* values of this branch will be reported. If the
 1261 comment field is NULL (both locale and text are empty) it will be ignored and any existing
 1262 comments will remain unchanged. If the comment is to be reset, an empty text with a locale
 1263 shall be provided.

1264 Table 19 specifies the *AddressSpace* representation for the *AddComment Method*.

1265 **Table 19 – AddComment Method AddressSpace definition**

Attribute	Value				
BrowseName	AddComment				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGeneratesEvent	ObjectType	AuditConditionComment EventType	Defined in 5.10.4		
ConformanceUnits					
A & C Comment					

1266

1267 If *Auditing* is supported, this *Method* shall generate an *Event* of
 1268 *AuditConditionCommentEventType* for all invocations of the *Method*.

1269 5.5.7 ConditionRefresh Method

1270 *ConditionRefresh* allows a *Client* to request a *Refresh* of all *Condition* instances that currently
 1271 are in an interesting state (they have the *Retain* flag set). This includes previous states of a
 1272 *Condition* instance for which the *Server* maintains *Branches*. A *Client* would typically invoke this
 1273 *Method* when it initially connects to a *Server* and following any situations, such as
 1274 communication disruptions, in which it would require resynchronization with the *Server*. This
 1275 *Method* is only available on the *ConditionType*. To invoke this *Method*, the call shall pass the
 1276 well-known *MethodId* of the *Method* on the *ConditionType* and the *ObjectId* shall be the well-
 1277 known *NodeId* of the *ConditionType ObjectType*.

Signature

```

ConditionRefresh (
    [in] IntegerId SubscriptionId
);

```

The parameters are defined in Table 20

Table 20 – ConditionRefresh parameters

Argument	Description
SubscriptionId	A valid <i>Subscription</i> Id of the <i>Subscription</i> to be refreshed. The <i>Server</i> shall verify that the <i>SubscriptionId</i> provided is part of the <i>Session</i> that is invoking the <i>Method</i> .

Method result codes in Table 21 (defined in Call Service)

Table 21 – ConditionRefresh result codes

Result Code	Description
Bad_SubscriptionIdInvalid	See IEC 62541-4 for the description of this result code
Bad_NothingToDo	The <i>ConditionRefresh Method</i> was called on a <i>SubscriptionId</i> that has no event <i>MonitoredItems</i> .
Bad_RefreshInProgress	See Table 137 for the description of this result code
Bad_UserAccessDenied	The <i>Method</i> was not called in the context of the <i>Session</i> that owns the <i>Subscription</i> See IEC 62541-4 for the general description of this result code.

Comments

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

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

- 1) The *Server* issues an event of *RefreshStartEventType* (defined in 5.11.2) marking the start of *Refresh*. A copy of the instance of *RefreshStartEventType* is queued into the *Event* stream for every *Notifier MonitoredItem* in the *Subscription*. Each of the *Event* copies shall contain the same *EventId*.
- 2) The *Server* issues *Event Notifications* of any *Retained Conditions* and *Retained Branches* of *Conditions* that meet the *Subscriptions* content filter criteria. Note that the *EventId* for such a refreshed *Notification* shall be identical to the one for the original *Notification*, the values of the other *Properties* are *Server* specific, in that some *Servers* may be able to replay the exact *Events* with all *Properties/Variables* maintaining the same values as originally sent, but other *Servers* might only be able to regenerate the *Event*. The regenerated *Event* might contain some updated *Property/Variable* values. For example, if the *Alarm* limits associated with a *Variable* were changed after the generation of the *Event* without generating a change in the *Alarm* state, the new limit might be reported. In another example, if the *HighLimit* was 100 and the *Variable* is 120. If the limit were changed to 90 no new *Event* would be generated since no change to the *StateMachine*, but the limit on a *Refresh* would indicate 90, when the original *Event* had indicated 100.
- 3) The *Server* may intersperse new *Event Notifications* that have not been previously issued to the *Notifier* along with those being sent as part of the *Refresh* request. *Clients* shall check for multiple *Event Notifications* for a *ConditionBranch* to avoid overwriting a new state delivered together with an older state from the *Refresh* process.
- 4) The *Server* issues an instance of *RefreshEndEventType* (defined in 5.11.3) to signal the end of the *Refresh*. A copy of the instance of *RefreshEndEventType* is queued into the *Event* stream for every *Notifier MonitoredItem* in the *Subscription*. Each of the *Events* copies shall contain the same *EventId*.

It is important to note that if multiple *Event Notifiers* are in a *Subscription* all *Event Notifiers* are processed. If a *Client* does not want all *MonitoredItems* refreshed, then the *Client* should place

each *MonitoredItem* in a separate *Subscription* or call *ConditionRefresh2* if the *Server* supports it.

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

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

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

Table 22 – ConditionRefresh Method AddressSpace definition

Attribute	Value				
BrowseName	ConditionRefresh				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGeneratesEvent	ObjectType	RefreshStartEventType	Defined in 5.11.2		
AlwaysGeneratesEvent	ObjectType	RefreshEndEventType	Defined in 5.11.3		
ConformanceUnits					
A & C Refresh					

5.5.8 ConditionRefresh2 Method

ConditionRefresh2 allows a *Client* to request a *Refresh* of all *Condition* instances that currently are in an interesting state (they have the *Retain* flag set) that are associated with the given Monitored item. In all other respects it functions as *ConditionRefresh*. A *Client* would typically invoke this *Method* when it initially connects to a *Server* and following any situations, such as communication disruptions where only a single *MonitoredItem* is to be resynchronized with the *Server*. This *Method* is only available on the *ConditionType*. To invoke this *Method*, the call shall pass the well-known *MethodId* of the *Method* on the *ConditionType* and the *ObjectId* shall be the well-known *NodeId* of the *ConditionType ObjectType*.

This *Method* is optional and as such *Clients* must be prepared to handle *Servers* which do not provide the *Method*. If the *Method* returns *Bad_MethodInvalid*, the *Client* shall revert to *ConditionRefresh*.

Signature

```
ConditionRefresh2 (
    [in] IntegerId SubscriptionId
    [in] IntegerId MonitoredItemId
);
```

The parameters are defined in Table 23

Table 23 – ConditionRefresh2 parameters

Argument	Description
SubscriptionId	The identifier of the <i>Subscription</i> containing the <i>MonitoredItem</i> to be refreshed. The <i>Server</i> shall verify that the <i>SubscriptionId</i> provided is part of the <i>Session</i> that is invoking the <i>Method</i> .
MonitoredItemId	The identifier of the <i>MonitoredItem</i> to be refreshed. The <i>MonitoredItemId</i> shall be in the provided <i>Subscription</i> .

Method result codes in Table 24(defined in Call Service)

Table 24 – ConditionRefresh2 result codes

Result Code	Description
Bad_SubscriptionIdInvalid	See IEC 62541-4 for the description of this result code
Bad_MonitoredItemIdInvalid	See IEC 62541-4 for the description of this result code
Bad_RefreshInProgress	See Table 137 for the description of this result code
Bad_UserAccessDenied	The <i>Method</i> was not called in the context of the Session that owns the <i>Subscription</i> See IEC 62541-4 for the general description of this result code.
Bad_MethodInvalid	See IEC 62541-4 for the description of this result code

Comments

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

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

- 1) The *Server* issues a *RefreshStartEvent* (defined in 5.11.2) marking the start of *Refresh*. The *RefreshStartEvent* is queued into the *Event* stream for the *Notifier MonitoredItem* in the *Subscription*.
- 2) The *Server* issues *Event Notifications* of any *Retained Conditions* and *Retained Branches* of *Conditions* that meet the *Subscriptions* content filter criteria. Note that the *EventId* for such a refreshed *Notification* shall be identical to the one for the original *Notification*, the values of the other *Properties* are *Server* specific, in that some *Servers* may be able to replay the exact *Events* with all *Properties/Variables* maintaining the same values as originally sent, but other *Servers* might only be able to regenerate the *Event*. The regenerated *Event* might contain some updated *Property/Variable* values. For example, if the *Alarm* limits associated with a *Variable* were changed after the generation of the *Event* without generating a change in the *Alarm* state, the new limit might be reported. In another example, if the *HighLimit* was 100 and the *Variable* is 120. If the limit were changed to 90 no new *Event* would be generated since no change to the *StateMachine*, but the limit on a *Refresh* would indicate 90, when the original *Event* had indicated 100.
- 3) The *Server* may intersperse new *Event Notifications* that have not been previously issued to the notifier along with those being sent as part of the *Refresh* request. *Clients* shall check for multiple *Event Notifications* for a *ConditionBranch* to avoid overwriting a new state delivered together with an older state from the *Refresh* process.
- 4) The *Server* issues a *RefreshEndEvent* (defined in 5.11.3) to signal the end of the *Refresh*. The *RefreshEndEvent* is queued into the *Event* stream for the *Notifier MonitoredItem* in the *Subscription*.

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

As mentioned above, *ConditionRefresh2* shall also issue *Event Notifications* for prior states if those states still need attention. In particular, this is True for *Condition* instances where previous states still need acknowledgement or confirmation.

Table 25 specifies the *AddressSpace* representation for the *ConditionRefresh2 Method*.

Table 25 – ConditionRefresh2 Method AddressSpace definition

Attribute	Value				
BrowseName	ConditionRefresh2				
References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGeneratesEvent	ObjectType	RefreshStartEventType	Defined in 5.11.2		
AlwaysGeneratesEvent	ObjectType	RefreshEndEventType	Defined in 5.11.3		
ConformanceUnits					
A & C Refresh2					