IEC 62541-9

Edition 3.0 2020-06
REDLINE VERSION

INTERNATIONAL STANDARD

colour inside

OPC unified architecture –
Part 9: Alarms and Conditions

IEC 62541-9:2020-06 RLV(en)

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

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

About the IEC
The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications
The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigendum or an amendment might have been published.

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 It also gives information on projects, replaced and withdrawn publications.

IEC Just Published - webstore.iec.ch/justpublished
Stay up to date on all new IEC publications. Just Published details all new publications released. Available online and once a month by email.

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

Electropedia - www.electropedia.org
The world's leading online dictionary on electrotechnology, containing more than 22 000 terminological entries in English and French, with equivalent terms in 16 additional languages. Also known as the International Electrotechnical Vocabulary (IEV) online.

IEC Glossary - std.iec.ch/glossary
67 000 electrotechnical terminology entries in English and French extracted from the Terms and Definitions clause of IEC publications issued since 2002. Some entries have been collected from earlier publications of IEC TC 37, 77, 86 and CISPR.

IEC 62541-9

Edition 3.0 2020-06
REDLINE VERSION

INTERNATIONAL STANDARD

colour inside

OPC unified architecture – Part 9: Alarms and Conditions

INTERNATIONAL ELECTROTECHNICAL COMMISSION

ICS 25.040.40; 35.100.05

ISBN 978-2-8322-8554-1

CONTENTS

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.

This redline version of the official IEC Standard allows the user to identify the changes made to the previous edition. A vertical bar appears in the margin wherever a change has been made. Additions are in green text, deletions are in strikethrough red text.

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 third edition cancels and replaces the second edition published in 2015. This edition constitutes a technical revision.

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

a) added optional engineering units to the definition of RateOfChange alarms;
b) to fulfill the IEC 62682 model, the following elements have been added:
   - AlarmConditionType States: Suppression, Silence, OutOfService, Latched;
   - AlarmConditionType Properties: OnDelay, OffDelay, FirstInGroup, ReAlarmTime;
   - New alarm types: DiscrepancyAlarm, DeviationAlarm, InstrumentDiagnosticAlarm, SystemDiagnosticAlarm.
c) added Annex that specifies how the concepts of this OPC UA part maps to IEC 62682 and ISA 18.2;
d) added new ConditionClasses: Safety, HighlyManaged, Statistical, Testing, Training;
e) added CertificateExpiration AlarmType;
f) added Alarm Metrics model.

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

FDIS	Report on voting
65E/709/FDIS	65E/727/RVD

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 –

Part 9: Alarms and Conditions

1 Scope

This part of IEC 62541 specifies the representation of *Alarms* and *Conditions* in the OPC Unified Architecture. Included is the *Information Model* representation of *Alarms* and *Conditions* in the OPC UA address space. Other aspects of alarm systems such as alarm philosophy, life cycle, alarm response times, alarm types and many other details are captured in documents 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.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC TR 62541-1, *OPC unified architecture – Part 1: Overview and concepts*

IEC 62541-3, *OPC unified architecture – Part 3: Address Space Model*

IEC 62541-4, *OPC unified architecture – Part 4: Services*

IEC 62541-5, *OPC unified architecture – Part 5: Information Model*

IEC 62541-6, *OPC unified architecture – Part 6: Mappings*

IEC 62541-7, *OPC unified architecture – Part 7: Profiles*

IEC 62541-8, *OPC unified architecture – Part 8: Data Access*

IEC 62541-11, *OPC unified architecture – Part 11: Historical Access*

IEC 62682, *Management of alarms systems for the process industries*

EEMUA: 2nd Edition EEMUA 191 – *Alarm System – A guide to design, management and procurement (Appendixes 6, 7, 8, 9)*, available at https://www.eemua.org/Products/Publications/Print/EEMUA-Publication-191.aspx

3 Terms, definitions, abbreviated terms and data types used

3.1 Terms and definitions

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

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- IEC Electropedia: available at http://www.electropedia.org/
- ISO Online browsing platform: available at http://www.iso.org/obp

3.1.1
Acknowledge
Operator action that indicates recognition of ~~a new~~ 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 uses *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 can 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 specifications such as IEC 62682 or ISA 18.2.

3.1.5
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 *HasAlarmSuppressionGroup ReferenceType*.

3.1.6
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~~ can derive more concrete classes or define different top-level classes.

3.1.7
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.

3.1.8
ConditionSource
element which a specific *Condition* is based upon or related to

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

Note 2 to entry: In *Events* generated for *Conditions*, the *SourceNode Property* (inherited from the *BaseEventType*) will contain the *NodeId* of the *ConditionSource*.

3.1.9
confirm
Operator action informing the *Server* that a corrective action has been taken to address the cause of the *Alarm*

3.1.10
disable
action configurating a system ~~is configured~~ such that the *Alarm* will not be generated even though the base *Alarm Condition* is present

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

Note 2 to entry: In IEC 62682, "disable" is referenced as "Out of Service".

3.1.11
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.

3.1.12
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 document, an Operator is a special user. All descriptions that apply to general users also apply to Operators.

3.1.13
Refresh
act of providing an update to an *Event Subscription* that provides all *Alarms* which are considered to be *Retained*

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

3.1.14
Retain
Alarm in a state that is interesting for a *Client* wishing to synchronize its state of *Conditions* with the *Server*'s state

3.1.15
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.

3.1.16
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 IEC62682 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 offline then low-level alarms are Suppressed for all equipment in the off-line unit.

3.2 Abbreviated terms

A&E Alarm & Event (as used for OPC COM)

COM (Microsoft Windows) Component Object Model

DA data access

UA Unified Architecture

3.3 Data types used

Table 1 and Table 2 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 document defines an *Information Model* for *Conditions*, Dialog *Conditions*, and *Alarms* including acknowledgement capabilities. It is built upon and extends base Event handling which is defined in IEC 62541-3, IEC 62541-4 and IEC 62541-5. This *Information Model* can also be extended to support the additional needs of specific domains. ~~The details of what aspects of the Information Model are supported are provided via Profiles (see IEC 62541-7). It is expected that systems will provide historical Events and Conditions via the standard Historical Access framework (see IEC 62541-11).~~ The details of which 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 subtypes of the *BaseEventType* (see IEC 62541-3 and IEC 62541-5). This part defines types that are common across many industries. It is expected that vendors or other standardisation groups will define additional *ConditionTypes* deriving from the common base types defined in this part. The *ConditionTypes* supported by a *Server* are exposed in the *AddressSpace* of the *Server*.

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

As mentioned above, *Conditions* represent the state of a system or one of its components. In certain cases, however, previous states that still need attention shall 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. *Server*s can generate separate *Event Notifications* for each branch. When the state represented by a *ConditionBranch* does not need further attention, a final *Event Notification* for this branch will have the *Retain Property* set to False. Subclause 4.4 provides more information and use cases. Maintaining previous states and therefore ~~also~~ the support of multiple branches is optional for *Servers*.

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

The base *Condition* state model is illustrated in Figure 1. It is extended by the various *Condition* subtypes defined in this document 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 substates.

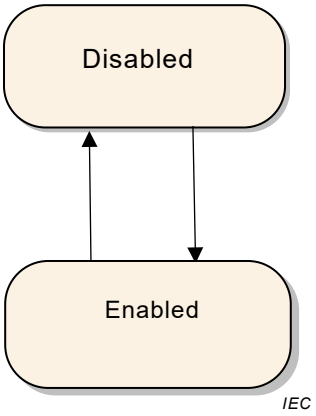


Figure 1 – Base Condition state model

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

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

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

4.3 Acknowledgeable Conditions

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

An *AckedState* and a *ConfirmedState* extend the *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*.

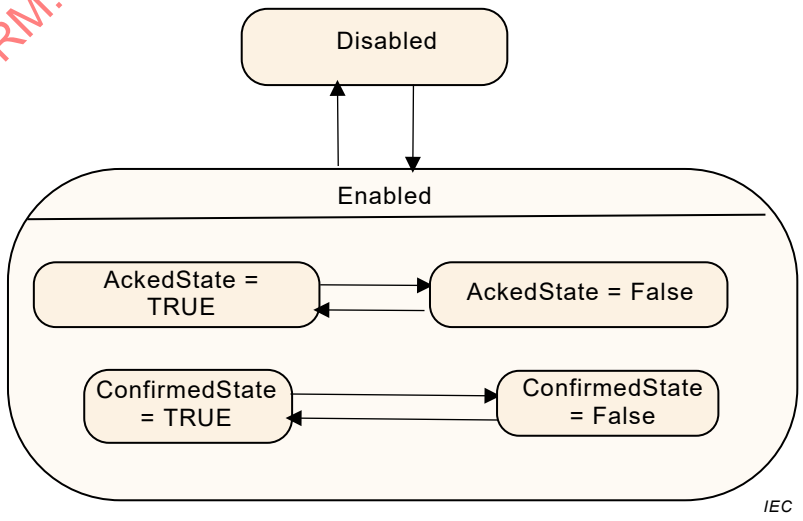


Figure 2 – AcknowledgeableConditions state model

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 document. Either of these state models can be extended to support more complex acknowledgement situations.

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

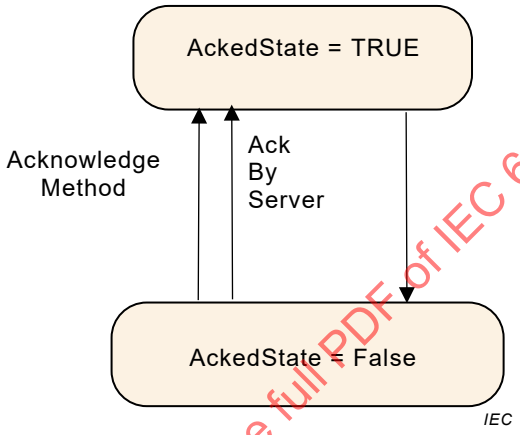


Figure 3 – Acknowledge state model

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

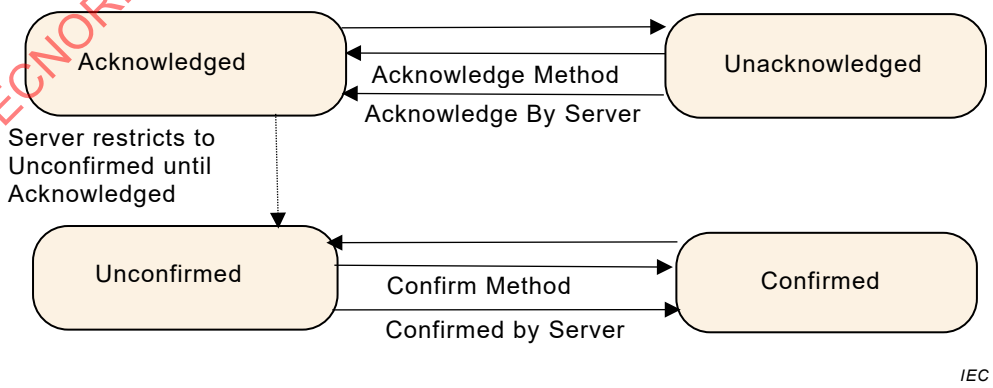


Figure 4 – Confirmed Acknowledge state model

4.4 Previous states of Conditions

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

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

4.5 Condition state synchronization

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

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

Clients request a *Refresh* by calling the ConditionRefresh *Method*. The *Server* will respond with a ~~*RefreshStartEvent*~~ *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 ~~*RefreshStartEvent*~~ *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 subclause.

- 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 Notification*s.

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

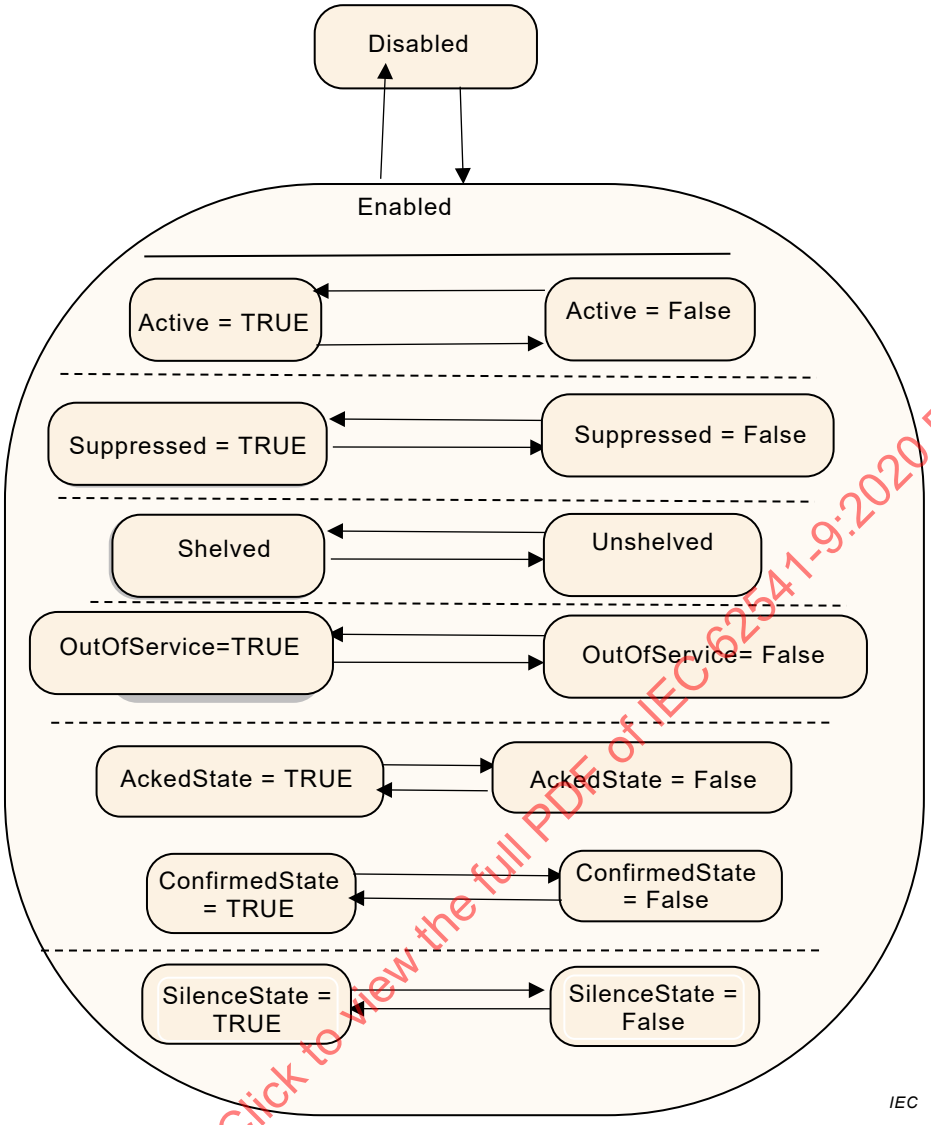


Figure 5 – Alarm state machine model

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

Some *Alarm* subtypes introduce substates 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 shelve features to help keep *Operators* from being overwhelmed during *Alarm* "storms" by limiting the number of *Alarms* an *Operator* sees on a current *Alarm* display. This is accomplished by setting the *SuppressedOrShelved* flag on second order dependent *Alarms* and/or *Alarms* of less severity, leading the *Operator* to concentrate on the most critical issues.

The shelved, 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*.

Alarms follow a typical timeline, which 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 *Operator*s about conditions in a timely manner and allow the *Operator* to take some action before some consequences occur. The consequences can be economic (product is not usable and shall be discarded), can be physical (tank overflows), can be safety related (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.

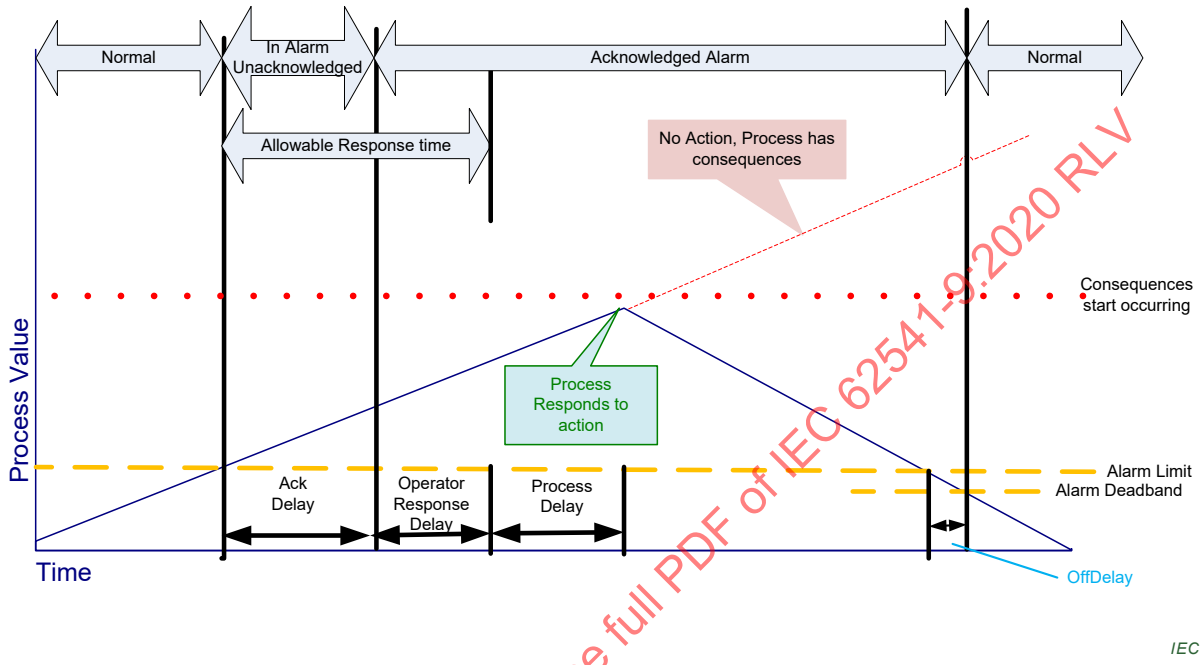


Figure 6 – Typical Alarm Timeline example

4.9 Multiple active states

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

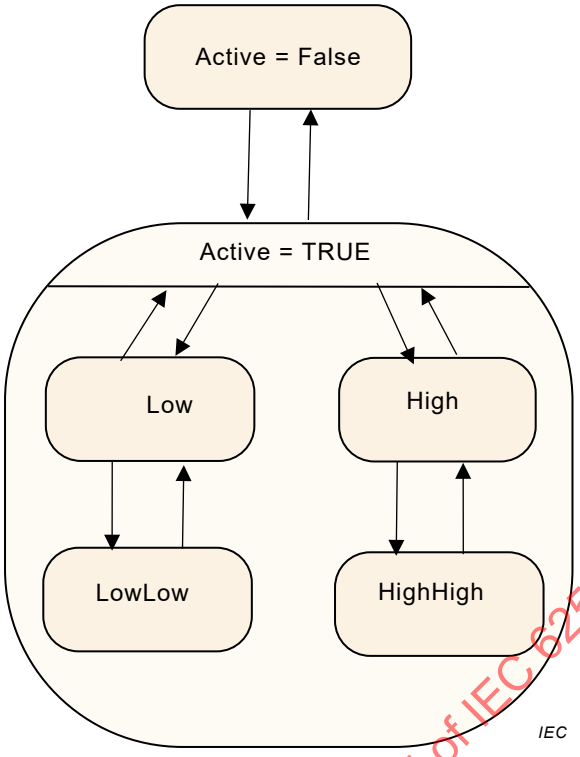


Figure 7 – Multiple active states example

With the multi-state *Alarm* model, state transitions among the substates 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 substate at a time. If for example a temperature exceeds the HighHigh limit the associated exclusive level *Alarm* will be in the HighHigh substate and not in the High substate.

Some *Alarm* systems, however, allow multiple substates 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 substate.

4.10 Condition instances in the AddressSpace

Because *Conditions* always have a state (*Enabled* or *Disabled*) and possibly many substates 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~~ The IEC 62541 series includes provisions for auditing. Auditing is an important security and tracking concept. Audit records provide the "Who", "When" and "What" information regarding user interactions with a system. These audit records are especially important when *Alarm* management is considered. *Alarms* are the typical instrument for providing information to a user that something needs the user's attention. A record of how the user reacts to this information is required in many cases. Audit records are generated for all *Method* calls that affect the state of the system, for example, an *Acknowledge Method* call would generate an ~~*AuditConditionAck*~~ *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 document defines a number of subtypes of the *AuditEventTypes* but without adding new fields to them.

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

5 Model

5.1 General

The *Alarm* and *Condition* model extends the OPC UA base *Event* model by defining various *Event Types* based on the *BaseEventType*. All of the *Event Types* defined in this document 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*.

Subclauses 5.5 to 5.8 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. The full *AlarmConditionType* hierarchy can be found in Figure ~~11~~ 8.

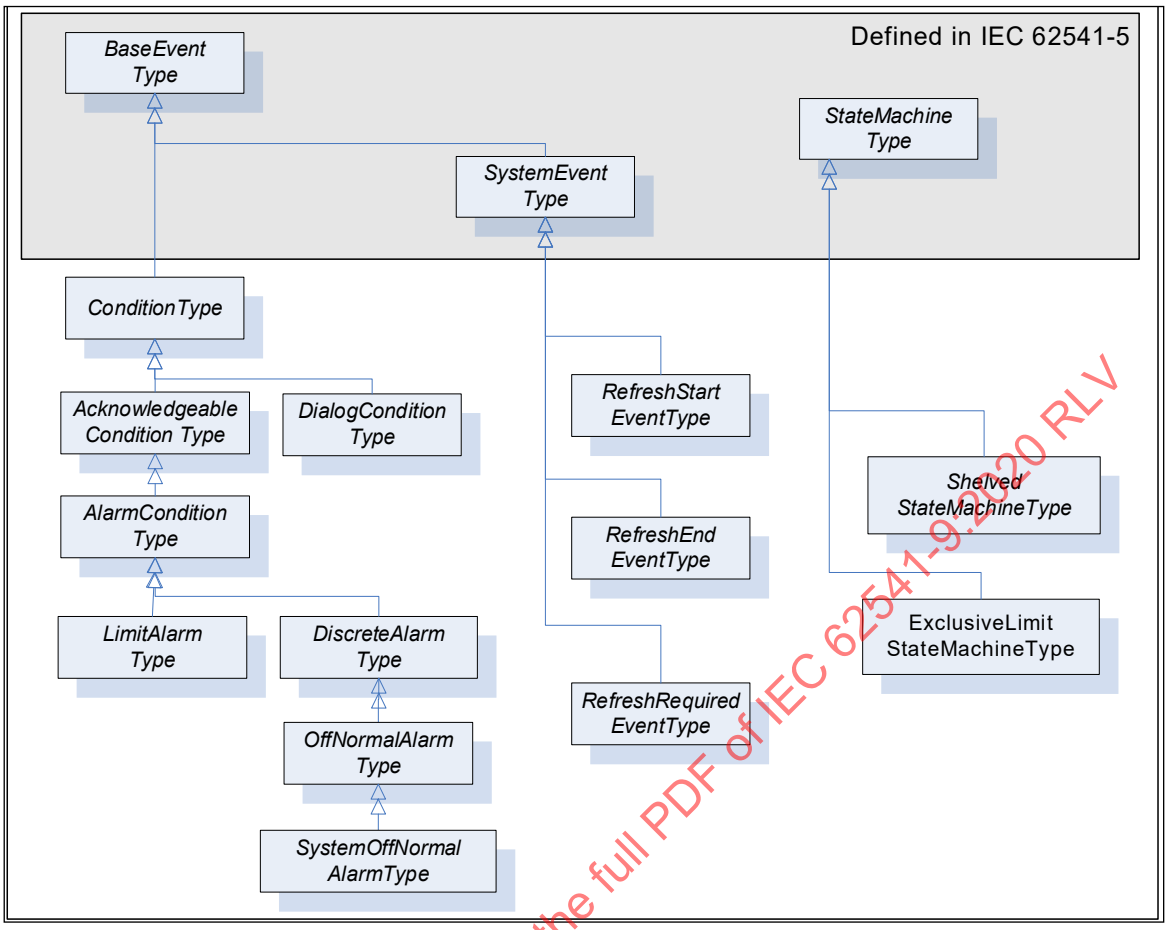


Figure 8 – ConditionType hierarchy

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 and the model described in this document.

5.2 Two-state state machines

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

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

Table 3 – TwoStateVariableType definition

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

The *Value Attribute* of ~~a *TwoStateVariable*~~ an instance of *TwoStateVariableType* contains the current state as a human readable name. The *EnabledState* for example, might contain the name "Enabled" when True and "Disabled" when False.

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

TransitionTime specifies the time when the current state was entered.

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

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

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

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

A *HasTrueSubState Reference* is used to indicate that the True state has substates.

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

5.3 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 *BaseDataVariableType* defined in IEC 62541-5.					
HasProperty	Variable	SourceTimestamp	UtcTime	PropertyType	Mandatory

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

5.4 ~~Substate~~ 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 to extend *TwoState* state machines with substates. These *References* will only exist when substates are available. For example if a TwoState machine is in a False State, then any substates referenced from the True state will not be available. If an Event is generated while in the False state and information from the True state substate is part of the data that is to be reported than this data would be reported as a NULL. With this approach *TwoStateVariables* can be extended with subordinate state machines in a similar fashion to the *StateMachineType* defined in IEC 62541-5.~~

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 substates and the addition of *Alarm* grouping.

The *TwoStateVariableType References* will only exist when substates are available. For example, if a *TwoStateVariableType* machine is in a False State, then any substates referenced from the True state will not be available. If an Event is generated while in the False state and information from the True state substate is part of the data that is to be reported than this data would be reported as a NULL. With this approach, *TwoStateVariableTypes* can be extended with subordinate state machines in a similar fashion to the *StateMachineType* defined in IEC 62541-5.

5.4.2 HasTrueSubState ReferenceType

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

The semantics indicate that the substate (the target Node) is a subordinate state of the True super state. If more than one state within a Condition is a substate of the same super state

(i.e. several HasTrueSubState References exist for the same super state) they are all treated as independent substates. The representation in the AddressSpace is specified in Table 5.

The SourceNode of the Reference shall be an instance of a TwoStateVariableType and the TargetNode shall 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 substate, but it is required that the substate provides the inverse Reference (IsTrueSubStateOf) to its super state.

Table 5 – HasTrueSubState ReferenceType

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

5.4.3 HasFalseSubState ReferenceType

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

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

The SourceNode of the Reference shall be an instance of a TwoStateVariableType and the TargetNode shall 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 substate, but it is required that the substate provides the inverse Reference (IsFalseSubStateOf) to its super state.

Table 6 – HasFalseSubState ReferenceType

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

5.4.4 HasAlarmSuppressionGroup ReferenceType

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

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

The *SourceNode* of the *Reference* shall be an instance of an *AlarmConditionType* or subtype and 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**

5.4.5 AlarmGroupMember ReferenceType

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

This *ReferenceType* is used to indicate the *Alarm* instances that are part of an *Alarm Group*.

The *SourceNode* of the *Reference* shall be an instance of an *AlarmGroupType* or subtype of it and 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**

5.5 Condition Model

5.5.1 General

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

The *Condition* model is illustrated in Figure 9 and formally defined in the subsequent tables. It is worth noting that this figure, like all figures in this document, is not intended to be complete. Rather, the figures only illustrate information provided by the formal definitions.

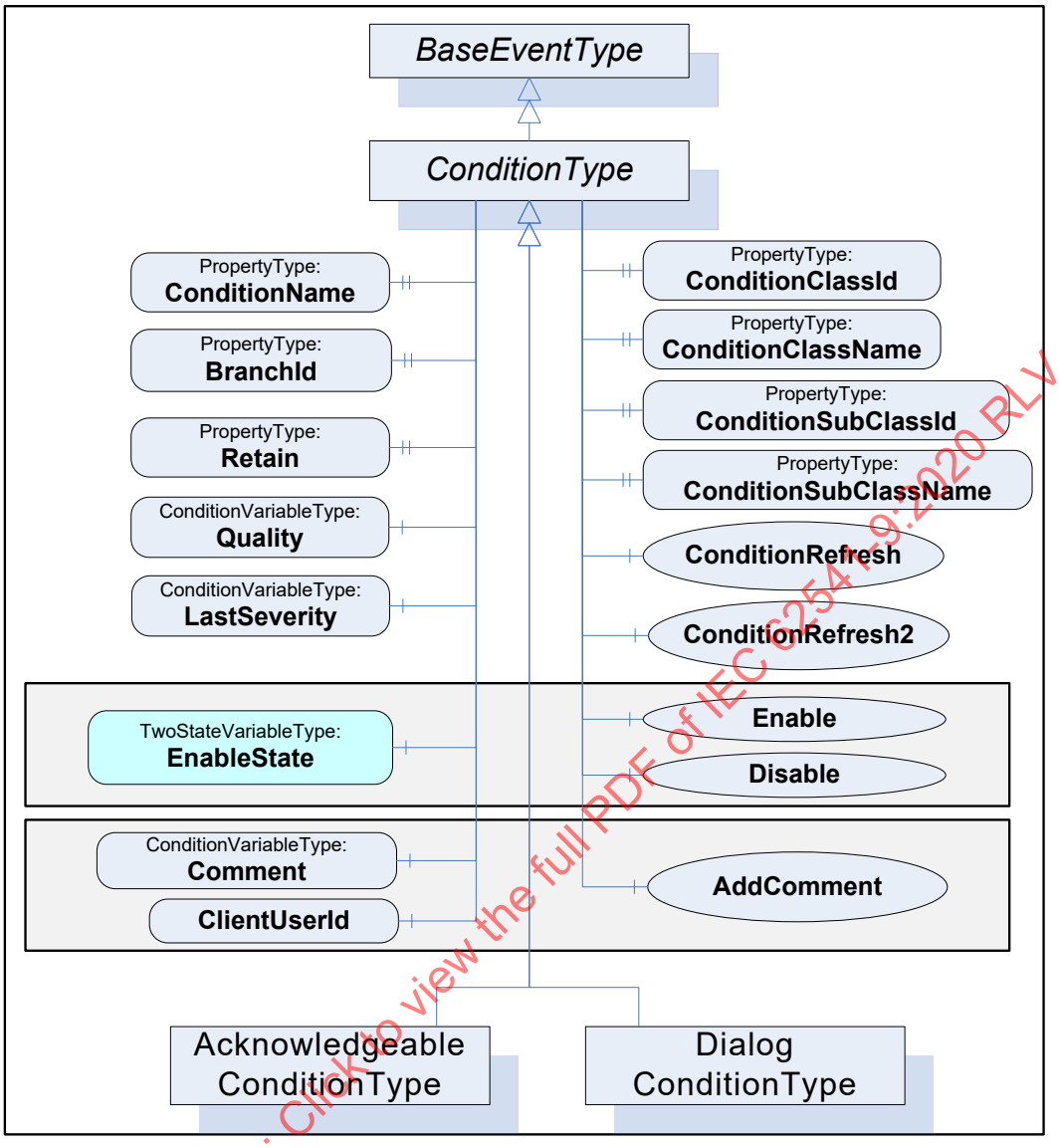


Figure 9 – Condition model

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 9. The False state of the *EnabledState* shall not be extended with a substate machine.

Table 9 – ConditionType definition

Attribute	Value				
BrowseName	ConditionType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseEventType* defined in IEC 62541-5					
HasSubtype	ObjectType	DialogConditionType	Defined in 5.6.2		
HasSubtype	ObjectType	AcknowledgeableConditionType	Defined in 5.7.2		
HasProperty	Variable	ConditionClassId	NodeId	PropertyType	Mandatory
HasProperty	Variable	ConditionClassName	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	ConditionSubClassId	NodeId[]	PropertyType	Optional
HasProperty	Variable	ConditionSubClassName	LocalizedText[]	PropertyType	Optional
HasProperty	Variable	ConditionName	String	PropertyType	Mandatory
HasProperty	Variable	BranchId	NodeId	PropertyType	Mandatory
HasProperty	Variable	Retain	Boolean	PropertyType	Mandatory
HasComponent	Variable	EnabledState	LocalizedText	TwoStateVariableType	Mandatory
HasComponent	Variable	Quality	StatusCode	ConditionVariableType	Mandatory
HasComponent	Variable	LastSeverity	UInt16	ConditionVariableType	Mandatory
HasComponent	Variable	Comment	LocalizedText	ConditionVariableType	Mandatory
HasProperty	Variable	ClientUserId	String	PropertyType	Mandatory
HasComponent	Method	Disable	Defined in 5.5.4		Mandatory
HasComponent	Method	Enable	Defined in 5.5.5		Mandatory
HasComponent	Method	AddComment	Defined in 5.5.6		Mandatory
HasComponent	Method	ConditionRefresh	Defined in 5.5.7		None
HasComponent	Method	ConditionRefresh2	Defined in 5.5.8		None

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* which are related to the motor as well as a temperature sensor associated with the motor. In the former, the *InputNode* for the High RPM *Alarm* is the value of the Motor RPM, while in the later the *InputNode* of the High Alarm would be the value of the temperature sensor that is associated with the motor.

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

ConditionClassName provides the display name of the ~~*ConditionClassType*~~ subtype of *BaseConditionClassType*.

ConditionSubClassId specifies additional class[es] that apply to the *Condition*. It is the *NodeId* of the corresponding subtype of *BaseConditionClassType*. See 5.9.6 for the definition of *ConditionClass* and a set of *ConditionClasses* defined in this document. When using this *Property* for filtering, *Clients* shall specify all individual subtypes of *BaseConditionClassType NodeIds*. The OfType operator cannot be applied. The *Client* specifies a NULL in the filter, to return *Conditions* where no sub class is applied. When returning *Conditions*, if this optional field is not available in a *Condition*, a NULL shall be returned for the field.

ConditionSubClassName provides the display name[s] of the *ConditionClassType[s]* listed in the ConditionSubClassId.

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

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

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

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

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

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

- When the *Condition* instance enters the *Disabled* state, the *Retain Property* of this *Condition* shall be set to False by the *Server* to indicate to the *Client* that the *Condition* instance is currently not of interest to *Client*s. This includes all *ConditionBranches* if any branches exist.
- When the *Condition* instance enters the enabled state, the *Condition* shall be evaluated and all of its *Properties* updated to reflect the current values. If this evaluation causes the *Retain Property* to transition to True for any *ConditionBranch*, then an *Event Notification* shall be generated for that *ConditionBranch*.
- The *Server* may choose to continue to test for a *Condition* instance while it is *Disable*d. However, no *Event Notifications* will be generated while the *Condition* instance is disabled.

- For any *Condition* that exists in the *AddressSpace* the *Attributes* and the following *Variables* will continue to have valid values even in the *Disable*d state; EventId, *Event* Type, Source *Node*, Source Name, Time, and EnabledState. Other *Properties* may no longer provide current valid values. All *Variables* that are no longer provided shall return a status of Bad_ConditionDisabled. The Event that reports the Disabled state should report the *Properties* as NULL or with a status of Bad_ConditionDisabled.

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

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

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

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

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

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

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

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

The *NodeId* of the *Condition* instance is used as *ConditionId*. It is not explicitly modelled as a component of the *ConditionType*. However, it can be requested with the following *SimpleAttributeOperand* (see Table 10) in the *SelectClause* of the *EventFilter*:

Table 10 – SimpleAttributeOperand

Name	Type	Description
SimpleAttributeOperand		
typeId	NodeId	*NodeId* of the *ConditionType Node*
browsePath[]	QualifiedName	empty
attributeId	IntegerId	Id of the *NodeId Attribute*

5.5.3 Condition and branch instances

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

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

The state of a *Condition* instance at any given time is the set values for the *Variable*s 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 document does not define a standard way to make them visible.

5.5.4 Disable Method

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

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*.

Signature

```
Disable();
```

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

Table 11 – Disable result codes

Result Code	Description
Bad_ConditionAlreadyDisabled	See Table 101 for the description of this result code.

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

Table 12 – Disable Method AddressSpace definition

Attribute	Value				
BrowseName	Disable				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionEnableEventType	Defined in 5.10.2		

5.5.5 Enable Method

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

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*.

Signature

```
Enable();
```

Method result codes in Table 13 (defined in Call Service).

Table 13 – Enable result codes

Result Code	Description
Bad_ConditionAlreadyEnabled	See Table 101 for the description of this result code.

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

Table 14 – Enable Method AddressSpace definition

Attribute	Value				
BrowseName	Enable				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionEnableEventType	Defined in 5.10.2		

5.5.6 AddComment Method

~~*AddComment* is used to apply a comment to a specific state of a *Condition* instance. Normally, the *MethodId* passed to the *Call Service* is found by browsing the *Condition* instance in the *AddressSpace*. However, some *Servers* do not expose *Condition* instances in the *AddressSpace*. Therefore all *Servers* shall allow *Clients* to call the *AddComment Method*~~

~~by specifying *ConditionId* as the *ObjectId* and the well known *NodeId* of the *Method* declaration on the *ConditionType* as the *MethodId*. The *Method* cannot be called on the *ConditionType Node*.~~

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
    [in] LocalizedText Comment
    );
```

The parameters are defined in Table 15.

Table 15 – AddComment arguments

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

Method result codes in Table 16 (defined in Call Service).

Table 16 – AddComment result codes

Result Code	Description
Bad_MethodInvalid	The *MethodId* provided does not correspond to the *ObjectId* provided. See IEC 62541-4 for the general description of this result code. ~~The addressed *Condition* does not support adding comments.~~
Bad_EventIdUnknown	See Table 101 for the description of this result code.
~~Bad_NodeIdUnknown~~ Bad_NodeIdInvalid	~~See IEC 62541-4 for the description of this result code.~~ Used to indicate that the specified ~~*Condition*~~ *ObjectId* is not valid or that the *Method* was called on the ConditionType *Node*. See IEC 62541-4 for the general description of this result code.

Comments

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

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

Table 17 specifies the *AddressSpace* representation for the *AddComment Method*.

Table 17 – AddComment Method AddressSpace definition

Attribute	Value				
BrowseName	AddComment				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	*Variable*	InputArguments	Argument[]	PropertyType	Mandatory
~~AlwaysGenerates Event~~ AlwaysGeneratesEvent	ObjectType	AuditConditionComment EventType	Defined in 5.10.4		

5.5.7 ConditionRefresh Method

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

Signature

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

The parameters are defined in Table 18.

Table 18 – ConditionRefresh parameters

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

Method result codes in Table 19 (defined in Call Service).

Table 19 – ConditionRefresh ~~ReturnCodes~~ result codes

Result Code	Description
Bad_SubscriptionIdInvalid	See IEC 62541-4 for the description of this result code
Bad_RefreshInProgress	See Table 101 for the description of this result code
Bad_UserAccessDenied	The *Method* was not called in the context of the *Session* that owns the *Subscription* See IEC 62541-4 for the general description of this result code.

Comments

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

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

1) The *Server* issues ~~a *RefreshStartEvent*~~ an event of *RefreshStartEvent*Type(defined in 5.11.2) marking the start of *Refresh*. A copy of the ~~*RefreshStartEvent*~~ 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*.~~ 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* might 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*~~ an instance of *RefreshEndEventType* (defined in 5.11.3) to signal the end of the *Refresh.* A copy of the ~~*RefreshEndEvent*~~ 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 20 specifies the *AddressSpace* representation for the *ConditionRefresh Method*.

Table 20 – ConditionRefresh Method AddressSpace definition

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

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* or its subtypes. To invoke

this *Method*, the call shall pass the well-known *MethodId* of the *Method* on the *ConditionType* and the *ObjectId* shall be the well-known *ObjectId* of the *ConditionType Object*.

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 21.

Table 21 – ConditionRefresh2 parameters

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

Method result codes in Table 22 (defined in Call Service).

Table 22 – 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 101 for the description of this result code
Bad_UserAccessDenied	The *Method* was not called in the context of the Session that owns the *Subscription*. 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

Subclause 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* which 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 23 specifies the *AddressSpace* representation for the *ConditionRefresh2 Method*.

Table 23 – ConditionRefresh2 Method AddressSpace definition

Attribute	Value				
BrowseName	ConditionRefresh2				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	*Variable*	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGeneratesEvent	ObjectType	RefreshStartEvent	Defined in 5.11.2		
AlwaysGeneratesEvent	ObjectType	RefreshEndEvent	Defined in 5.11.3		

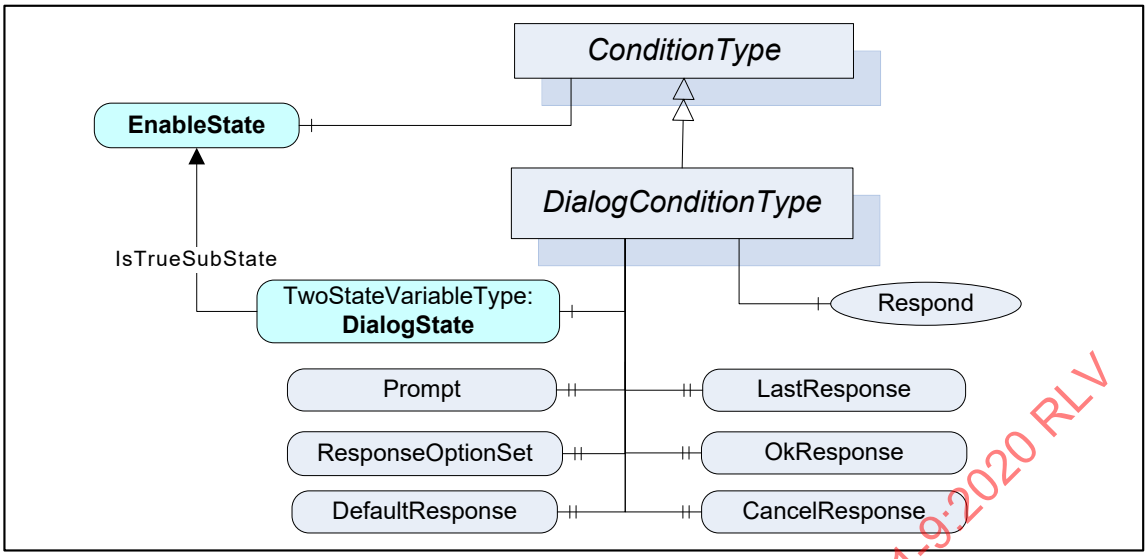
5.6 Dialog Model

5.6.1 General

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

5.6.2 DialogConditionType

The *DialogConditionType* is used to represent *Conditions* as dialogs. It is illustrated in Figure 10 and formally defined in Table 24.



IEC

Figure 10 – DialogConditionType overview

Table 24 – DialogConditionType definition

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

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

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

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

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

Typical combinations of response options are

- OK

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

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

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

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

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

5.6.3 Respond Method

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

Signature

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

The parameters are defined in Table 25.

Table 25 – Respond parameters

Argument	Description
SelectedResponse	Selected index of the *ResponseOptionSet* array.

Method result codes in Table 26 (defined in Call Service).

Table 26 – Respond Result Codes

Result Code	Description
Bad_DialogNotActive	See Table 101 for the description of this result code.
Bad_DialogResponseInvalid	See Table 101 for the description of this result code.

Table 27 specifies the *AddressSpace* representation for the *Respond Method*.

Table 27 – Respond Method AddressSpace definition

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

5.7 Acknowledgeable Condition Model

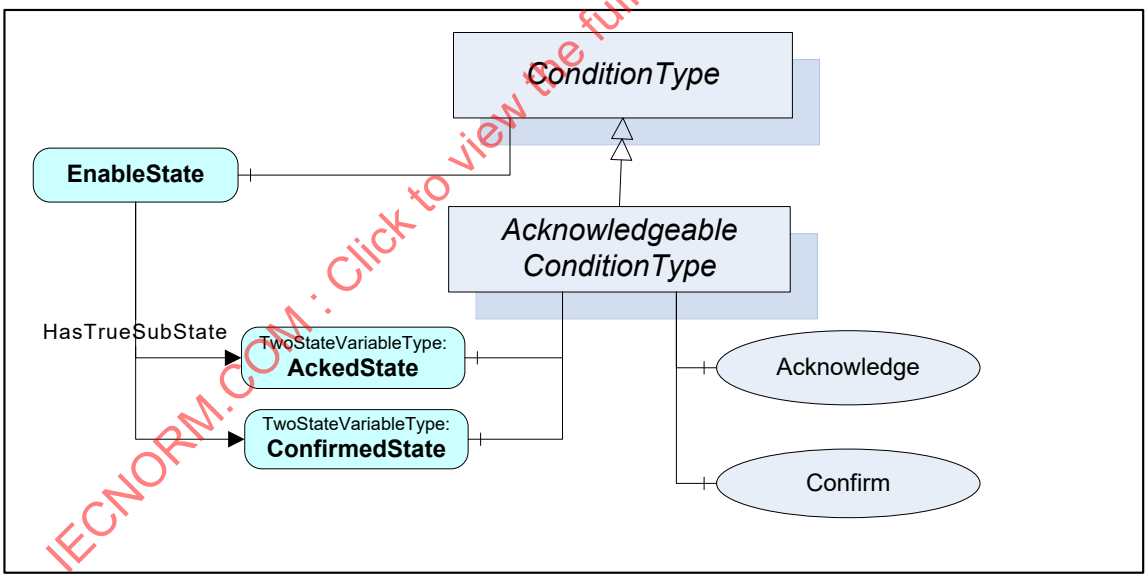
5.7.1 General

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

AcknowledgeableConditions are represented by the *AcknowledgeableConditionType* which is a subtype of the *ConditionType*. The model is formally defined in 5.7.2 to 5.7.4.

5.7.2 AcknowledgeableConditionType

The *AcknowledgeableConditionType* extends the *ConditionType* by defining acknowledgement characteristics. It is an abstract type. The *AcknowledgeableConditionType* is illustrated in Figure 11 and formally defined in Table 28.



IEC

Figure 11 – AcknowledgeableConditionType overview

Table 28 – AcknowledgeableConditionType definition

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

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

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

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

5.7.3 Acknowledge Method

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

The *Acknowledge Method* is used to acknowledge an *Event Notification* for a *Condition* instance state where *AckedState* is False. 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, *Servers* shall allow *Clients* to call the *Acknowledge Method* by specifying *ConditionId* as the *ObjectId*. The *Method* cannot be called with an *ObjectId* of the *AcknowledgeableConditionType Node*.

Signature

```
Acknowledge(
    [in] ByteString EventId
    [in] LocalizedText Comment
```

);

The parameters are defined in Table 29.

Table 29 – Acknowledge parameters

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

Method result codes in Table 30 (defined in Call Service).

Table 30 – Acknowledge result codes

Result Code	Description
Bad_ConditionBranchAlreadyAcked	See Table 101 for the description of this result code.
Bad_MethodInvalid	The method id does not refer to a method for the specified object or ConditionId.
Bad_EventIdUnknown	See Table 101 for the description of this result code.
~~Bad_NodeIdUnknown~~ Bad_NodeIdInvalid	~~See IEC 62541-4 for the description of this result code.~~ Used to indicate that the specified ~~*Condition*~~ *ObjectId* is not valid or that the *Method* was called on the ConditionType *Node*. See IEC 62541-4 for the general description of this result code.

Comments

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

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

A valid *EventId* will result in an *Event Notification* where *AckedState*/Id is set to True and the *Comment Property* contains the text of the optional comment argument. If a previous state is acknowledged, the *BranchId* and all *Condition* values of this branch will be reported. Table 31 specifies the *AddressSpace* representation for the *Acknowledge Method*.

Table 31 – Acknowledge Method AddressSpace definition

Attribute	Value				
BrowseName	Acknowledge				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	*Variable*	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGenerates Event	ObjectType	AuditConditionAcknowledge EventType	Defined in 5.10.5		

5.7.4 Confirm Method

~~*Confirm* is used to confirm an *Event Notifications* for a *Condition* instance state where *ConfirmedState* was set to False. Normally, the *MethodId* passed to the *Call Service* is found by browsing the *Condition* instance in the *AddressSpace*. However, some *Servers* do not~~

~~expose *Condition* instances in the *AddressSpace*. Therefore all *Servers* shall allow *Clients* to call the *Confirm Method* by specifying *ConditionId* as the *ObjectId* and the well known *NodeId* of the *Method* declaration on the *AcknowledgeableConditionType* as the *MethodId*. The *Method* cannot be called on the *AcknowledgeableConditionType Node*.~~

The *Confirm Method* is used to confirm an *Event Notifications* for a *Condition* instance state where *ConfirmedState* is False. 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, *Servers* shall allow *Clients* to call the *Confirm Method* by specifying *ConditionId* as the *ObjectId*. The *Method* cannot be called with an *ObjectId* of the *AcknowledgeableConditionType Node*.

Signature

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

The parameters are defined in Table 32.

Table 32 – Confirm Method parameters

Argument	Description
EventId	*EventId* identifying a particular *Event Notification*. Only *Event Notifications* where the Id property of the *ConfirmedState*~~/Id was TRUE~~ is False can be confirmed.
Comment	A localized text to be applied to the *Conditions*.

Method result codes in Table 33 (defined in Call Service).

Table 33 – Confirm result codes

Result Code	Description
Bad_ConditionBranchAlreadyConfirmed	See Table 101 for the description of this result code.
Bad_MethodInvalid	The method id does not refer to a method for the specified object or *ConditionId*. See IEC 62541-4 for the general description of this result code.
Bad_EventIdUnknown	See Table 101 for the description of this result code.
Bad_NodeIdUnknown	~~See IEC 62541-4 for the description of this result code.~~ Used to indicate that the specified ~~*Condition*~~ *ObjectId* is not valid or that the *Method* was called on the ConditionType *Node*. See IEC 62541-4 for the general description of this result code.

Comments

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

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

A valid *EventId* will result in an *Event Notification* where *ConfirmedState*/Id is set to True and the *Comment Property* contains the text of the optional comment argument. If a previous state is confirmed, the *BranchId* and all *Condition* values of this branch will be reported. A *Client* can confirm only events that have a *ConfirmedState*/Id set to False. The logic for setting *ConfirmedState*/Id to False is *Server* specific and may even be event or condition specific.

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

Table 34 – Confirm Method AddressSpace definition

Attribute	Value				
BrowseName	Confirm				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	*Variable*	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGeneratesEvent	ObjectType	~~AuditConditionConfirmEventType~~ AuditConditionConfirmEventType	Defined in 5.10.7		

5.8 Alarm model

5.8.1 General

Figure 12 informally describes the AlarmConditionType, its subtypes and where it is in the hierarchy of *Event Types*.

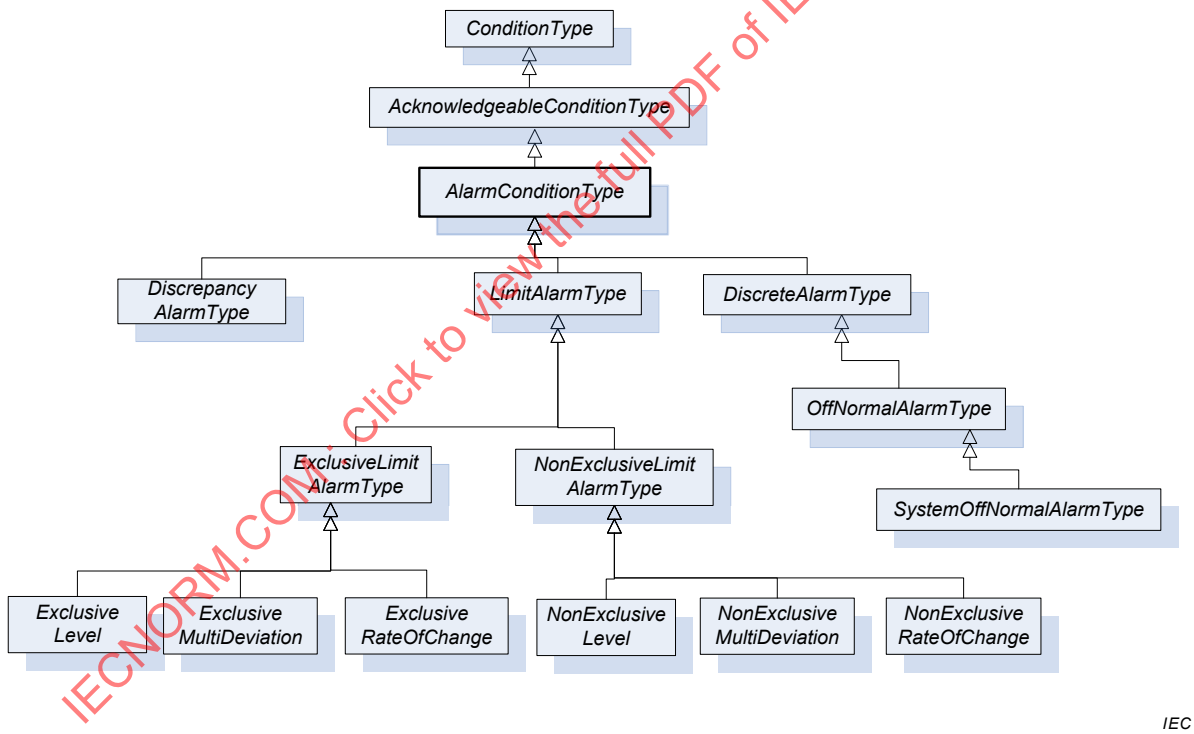


Figure 12 – AlarmConditionType Hierarchy Model

5.8.2 AlarmConditionType

The *AlarmConditionType* is an abstract type that extends the *AcknowledgeableConditionType* by introducing an *ActiveState*, *SuppressedState* and *ShelvingState*. It also adds the ability to set a delay time, re-alarm time, *Alarm* groups and audible *Alarm* settings The *Alarm* model is illustrated in Figure 13. This illustration is not intended to be a complete definition. It is formally defined in Table 35.

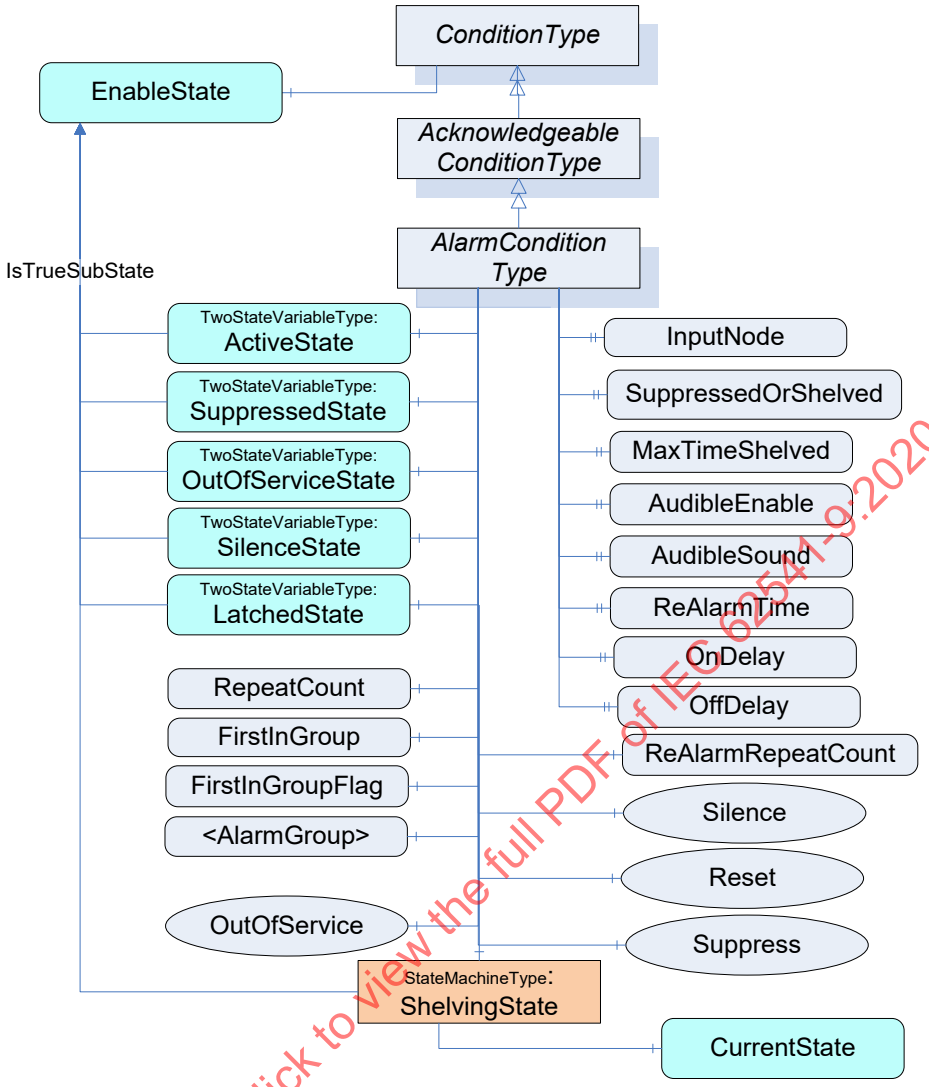


Figure 13 – Alarm Model

Table 35 – AlarmConditionType definition

Attribute	Value				
BrowseName	AlarmConditionType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the AcknowledgeableConditionType defined in clause 5.7.2					
HasComponent	Variable	ActiveState	LocalizedText	TwoStateVariableType	Mandatory
HasProperty	Variable	InputNode	NodeId	PropertyType	Mandatory
HasComponent	Variable	SuppressedState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Variable	OutOfServiceState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Object	ShelvingState		ShelvedStateMachineType	Optional
HasProperty	Variable	SuppressedOrShelved	Boolean	PropertyType	Mandatory
HasProperty	Variable	MaxTimeShelved	Duration	PropertyType	Optional
HasProperty	Variable	AudibleEnabled	Boolean	PropertyType	Optional
HasComponent	Variable	AudibleSound	AudioDataType	AudioVariableType	Optional
HasComponent	Variable	SilenceState	LocalizedText	TwoStateVariableType	Optional
HasProperty	Variable	OnDelay	Duration	PropertyType	Optional
HasProperty	Variable	OffDelay	Duration	PropertyType	Optional
HasComponent	Variable	FirstInGroupFlag	Boolean	BaseDataVariableType	Optional
HasComponent	Object	FirstInGroup		AlarmGroupType	Optional
HasComponent	Object	LatchedState	LocalizedText	TwoStateVariableType	Optional
HasAlarmSuppressionGroup	Object	<AlarmGroup>		AlarmGroupType	OptionalPlaceholder
HasProperty	Variable	ReAlarmTime	Duration	PropertyType	Optional
HasComponent	Variable	ReAlarmRepeatCount	Int16	BaseDataVariableType	Optional
HasComponent	Method	Silence	Defined in 5.8.5		Optional
HasComponent	Method	Suppress	Defined in 5.8.6		Optional
HasComponent	Method	Unsuppress	Defined in 5.8.7		Optional
HasComponent	Method	RemoveFromService	Defined in 5.8.8		Optional
HasComponent	Method	PlaceInService	Defined in 5.8.9		Optional
HasComponent	Method	Reset	Defined in 5.8.4		Optional
HasSubtype	Object	DiscreteAlarmType			
HasSubtype	Object	LimitAlarmType			
HasSubtype	Object	DiscrepancyAlarmType			

The *AlarmConditionType* inherits all *Properties* of the *AcknowledgeableConditionType*. The following states are substates of the True *EnabledState*.

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

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

SuppressedState, *OutOfServiceState* and *ShelvingState* together allow the suppression of *Alarms* on display systems. These three suppressions are generally used by different personnel or systems at a plant, i.e. automatic systems, maintenance personnel and *Operators*.

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

OutOfServiceState is used by maintenance personnel to suppress *Alarms* due to a maintenance issue. For example, if an instrument is taken out of service for maintenance or is removed temporarily while it is being replaced or serviced, the item would have the *OutOfServiceState* set. Recommended state names are described in Annex A.

ShelvingState suggests whether an *Alarm* shall (temporarily) be prevented from being displayed to the user. It is quite often used by *Operators* to block nuisance *Alarms*. The *ShelvingState* is defined in 5.8.10.

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

When an *Alarm* has any or all of the *SuppressedState, OutOfServiceState* or *ShelvingState* set to True, the *SuppressedOrShelved* property shall be set True and this *Alarm* is then typically not displayed by the *Client*. State transitions associated with the *Alarm* do occur, but they are not typically displayed by the *Client*s as long as the *Alarm* remains in any of the *SuppressedState, OutOfServiceState* or *Shelved* state.

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

The optional *Property AudibleEnabled* is a Boolean that indicates if the current state of this *Alarm* includes an audible *Alarm*.

The optional *Property AudibleSound* contains the sound file that is to be played if an audible *Alarm* is to be generated. This file would be play/generated as long as the *Alarm* is active and unacknowledged, unless the silence *StateMachine* is included, in which case it may also be silenced by this *StateMachine*.

The *SilenceState* is used to suppress the generation of audible *Alarms*. Typically, it is used when an *Operator* silences all *Alarms* on a screen, but needs to acknowledge the *Alarms* individually. Silencing an *Alarm* shall silence the *Alarm* on all systems (screens) that it is being reported on. Not all *Clients* will make use of this *StateMachine*, but it allows multiple *Clients* to synchronize audible *Alarm* states. Acknowledging an *Alarm* shall automatically silence an *Alarm*.

The *OnDelay* and *OffDelay Properties* can be used to eliminate nuisance *Alarms*. The *OnDelay* is used to avoid unnecessary Alarms when a signal temporarily overshoots its setpoint, thus preventing the *Alarm* from being triggered until the signal remains in the *Alarm* state continuously for a specified length of time (*OnDelay* time). The *OffDelay* is used to reduce chattering *Alarms* by locking the *Alarm* indication for a certain holding period after the condition has returned to normal, i.e. the *Alarm* shall stay active for the *OffDelay* time and shall not regenerate if it returns to active in that period. If the *Alarm* remains in the inactive zone for *OffDelay*, it will then become inactive.

The optional variable *FirstInGroupFlag* is used together with the *FirstInGroup* object. The FirstInGroup Object is an instance of an AlarmGroupType that groups a number of related *Alarms*. The *FirstInGroupFlag* is set on the *Alarm* instance that was the first *Alarm* to trigger in a *FirstInGroup*. If this variable is present, then the *FirstInGroup* shall also be present. These two nodes allow an alarming system to determine which *Alarm* in the list was the trigger. It is commonly used in situations where *Alarms* are interrelated, and usually multiple *Alarms* occur. For example, usually all vibration sensors in a turbine trigger if any one of them triggers, but what is important for an *Operator* is the first sensor that triggered.

The *LatchedState Object*, if present, indicates that this *Alarm* supports being latched. The *Alarm* will remain with a retain bit of True until it is no longer active, is acknowledge and is reset. The *Reset Method,* if called while active has no effect on the *Alarm* and is ignored and an error of Bad_InvalidState is return on the call. The *Object* indicates the current state, latched or not latched. Recommended state names are described in Annex A. If this *Object* is provided, the *Reset Method* shall also be provided.

An *Alarm* instance may contain HasAlarmSuppressionGroup reference(s) to instance(s) of *AlarmGroupType*. Each instance is an *AlarmSuppressionGroup.* When an *AlarmSuppressionGroup* goes active, the *Server* shall set the *SuppressedState* of the *Alarm* to True. When all of referenced *AlarmSuppressionGroups* are no longer active, then the Server shall set *SuppressedState* to False. A single *AlarmSuppressionGroup* can be assigned to multiple *Alarms*. *AlarmSuppressionGroups* are used to control *AlarmFloods* and to help manage *Alarms*.

ReAlarmTime if present sets a time that is used to bring an *Alarm* back to the top of an *Alarm* list. If an *Alarm* has not returned to normal within the provided time (from when it last was alarmed), the *Server* will generate a new *Alarm* for it (as if it just went into alarm). If it has been silenced it shall return to an un-silenced state, if it has been acknowledged it shall return to unacknowledged. The *Alarm* active time is set to the time of the re-alarm.

ReAlarmRepeatCount, if present, counts the number times an *Alarm* was re-alarmed. Some smart alarming system would use this count to raise the priority or otherwise generate additional or different annunciations for the given *Alarm*. The count is reset when an *Alarm* returns to normal.

Silence Method may be used to silence an instance of an *Alarm*. It is defined in 5.8.5.

Suppress Method may be used to suppress an instance of an *Alarm*. Most *Alarm* suppression occurs via advanced alarming, but this method allows additional access to suppress a particular *Alarm* instance. Additional details are provided in the definition in 5.8.6.

Unsuppress Method may be used to remove an instance of an *Alarm* from SuppressedState. Additional details are provided in the definition in 5.8.7.

PlaceInService Method may be used to remove an instance of an *Alarm* from *OutOfServiceState*. It is defined in 5.8.9.

RemoveFromService Method may be used to place an instance of an *Alarm* in *OutOfServiceState*. It is defined in 5.8.8.

Reset Method is used to clear a latched *Alarm*. It is defined in 5.8.4. If this *Object* is provided, the *LatchedState Object* shall also be provided.

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

5.8.3 AlarmGroupType

The *AlarmGroupType* provides a simple manner of grouping *Alarms*. This grouping may be used for *Alarm* suppression or for identifying related *Alarms*. The actual usage of the *AlarmGroupType* is specified where it is used.

The *AlarmGroupType* is formally defined in Table 36.

Table 36 – AlarmGroupType definition

Attribute	Value				
BrowseName	AlarmGroupType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the FolderType defined in IEC 62541-5.					
AlarmGroupMember	Object	<AlarmConditionInstance>		AlarmConditionType	OptionalPlaceholder

The instance of an *AlarmGroupType* should be given a name and description that describes the purpose of the *Alarm* group.

The *AlarmGroupType* instance will contain a list of instances of *AlarmConditionType* or subtype of *AlarmConditionType* referenced by *AlarmGroupMember* references. At least one *Alarm* shall be present in an instance of an *AlarmGroupType*.

5.8.4 Reset Method

The *Reset Method* is used reset a latched *Alarm* instance. It is only available on an instance of an *AlarmConditionType* that exposes the *LatchedState*. 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, *Servers* shall allow *Clients* to call the *Reset Method* by specifying *ConditionId* as the *ObjectId*. The *Method* cannot be called with an *ObjectId* of the *AlarmConditionType Node*.

Signature

```
Reset();
```

Method result codes are given in Table 37 (defined in *Call* service).

Table 37 – Silence result codes

Result Code	Description
Bad_MethodInvalid	The *MethodId* provided does not correspond to the *ObjectId* provided. See IEC 62541-4 for the general description of this result code.
Bad_NodeIdInvalid	Used to indicate that the specified *ObjectId* is not valid or that the *Method* was called on the *ConditionType Node*. See IEC 62541-4 for the general description of this result code.
Bad_InvalidState	The *Alarm* instance was not latched or still active or still required acknowledgement. For an *Alarm* Instance to be reset, it must have been in *Alarm*, and returned to normal and have been acknowledged prior to being reset.

Table 38 specifies the *AddressSpace* representation for the *Reset Method*.

Table 38 – Reset Method AddressSpace definition

Attribute	Value				
BrowseName	Reset				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	Ob jectType	AuditConditionResetEventType	Defined in 5.10.11		

5.8.5 Silence Method

The *Silence Method* is used to silence a specific *Alarm* instance. It is only available on an instance of an *AlarmConditionType* that also exposes the *SilenceState*. 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, *Servers* shall allow *Clients* to call the *Silence Method* by specifying *ConditionId* as the *ObjectId*. The *Method* cannot be called with an *ObjectId* of the *AlarmConditionType Node*.

Signature

```
Silence();
```

Method result codes in Table 39 (defined in *Call* service).

Table 39 – Silence result codes

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

Comments

If the instance is not currently in an audible state, the command is ignored.

Table 40 specifies the *AddressSpace* representation for the *Silence Method*.

Table 40 – Silence Method AddressSpace definition

Attribute	Value				
BrowseName	Silence				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
AlwaysGeneratesEvent	ObjectType	AuditConditionSilenceEventType	Defined in 5.10.10		

5.8.6 Suppress Method

The *Suppress Method* is used to suppress a specific *Alarm* instance. It is only available on an instance of an *AlarmConditionType* that also exposes the *SuppressedState*. This *Method* may be used to change the *SuppressedState* of an *Alarm* and overwrite any suppression caused by an associated *AlarmSuppressionGroup*. This *Method* works in parallel with any suppression triggered by an *AlarmSupressionGroup*, in that if the *Method* is used to suppress an *Alarm*, an *AlarmSuppressionGroup* might clear the suppression.

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, *Servers* shall allow *Clients* to call the *Suppress Method* by specifying *ConditionId* as the *ObjectId.* The *Method* may not be called with an *ObjectId* of the *AlarmConditionType Node*.

Signature

```
Suppress();
```

Method Result Codes in Table 41 (defined in Call Service).

Table 41 – Suppress result codes

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

Comments

Suppress Method applies to an *Alarm* instance, even if it is not currently active.

Table 42 specifies the *AddressSpace* representation for the *Suppress Method*.

Table 42 – Suppress Method AddressSpace definition

Attribute	Value				
BrowseName	Suppress				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
AlwaysGeneratesEvent	ObjectType	AuditConditionSuppressionEventType	Defined in 5.10.4		

5.8.7 Unsuppress Method

The *Unsuppress Method* is used to clear the *SuppressedState* of a specific *Alarm* instance. It is only available on an instance of an *AlarmConditionType* that also exposes the *SuppressedState*. This *Method* may be used to overwrite any suppression cause by an associated AlarmSuppressionGroup. This *Method* works in parallel with any suppression triggered by an AlarmSuppressionGroup, in that if the *Method* is used to clear the *SuppressedState* of an *Alarm*, any change in an AlarmSuppressionGroup might again suppress the *Alarm*.

Normally, the *NodeId* of the ObjectInstance is passed as the *ObjectId* to the *Call Service*. However, some *Servers* do not expose *Condition* instances in the *AddressSpace.* Therefore, *Servers* shall allow *Clients* to call the *Unsuppress Method* by specifying *ConditionId* as the *ObjectId.* The *Method* may not be called with an *ObjectId* of the *AlarmConditionType Node*.

Signature

```
Unsuppress();
```

Method Result Codes in Table 43 (defined in Call Service).

Table 43 – Unsuppress result codes

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

Comments

Unsuppress Method applies to an *Alarm* instance, even if it is not currently active.

Table 44 specifies the *AddressSpace* representation for the *Suppress Method*.

Table 44 – Unsuppress Method AddressSpace definition

Attribute	Value				
BrowseName	Unsuppress				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionSuppressionEventType	Defined in 5.10.4		

5.8.8 RemoveFromService Method

The *RemoveFromService Method* is used to suppress a specific *Alarm* instance. It is only available on an instance of an AlarmConditionType that also exposes the *OutOfServiceState*. 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, *Servers* shall allow *Clients* to call the *RemoveFromService Method* by specifying *ConditionId* as the *ObjectId.* The *Method* may not be called with an *ObjectId* of the *AlarmConditionType Node*.

Signature

```
RemoveFromService ();
```

Method result codes in Table 45 (defined in *Call Service*).

Table 45 – RemoveFromService result codes

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

Comments

Instances that do not expose the *OutOfService State* shall reject *RemoveFromService* calls. *RemoveFromService Method* applies to an *Alarm* instance, even if it is not currently in the *Active State*.

Table 46 specifies the *AddressSpace* representation for the *RemoveFromService Method*.

Table 46 – RemoveFromService Method AddressSpace definition

Attribute	Value				
BrowseName	RemoveFromService				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
AlwaysGeneratesEvent	ObjectType	AuditConditionOutOfServiceEventType	Defined in 5.10.12		

5.8.9 PlaceInService Method

The *PlaceInService Method* is used to set the OutOfServiceState to False of a specific *Alarm* instance. It is only available on an instance of an *AlarmConditionType* that also exposes the *OutOfServiceState*. Normally, the *NodeId* of the ObjectInstance is passed as the *ObjectId* to the *Call Service*. However, some *Servers* do not expose *Condition* instances in the *AddressSpace.* Therefore, *Servers* shall allow *Clients* to call the *PlaceInService Method* by specifying *ConditionId* as the *ObjectId.* The *Method* may not be called with an *ObjectId* of the *AlarmConditionType Node*.

Signature

```
PlaceInService ();
```

Method result codes in Table 47 (defined in *Call Service*).

Table 47 – PlaceInService result codes

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

Comments

The *PlaceInService Method* applies to an *Alarm* instance, even if it is not currently in the *Active State*.

Table 48 specifies the *AddressSpace* representation for the *PlaceInService Method*.

Table 48 – PlaceInService Method AddressSpace definition

Attribute	Value				
BrowseName	PlaceInService				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionOutOfServiceEventType	Defined in 5.10.12		

5.8.10 ShelvedStateMachineType

5.8.10.1 Overview

The *ShelvedStateMachineType* defines a substate machine that represents an advanced *Alarm* filtering model. This model is illustrated in Figure 15.

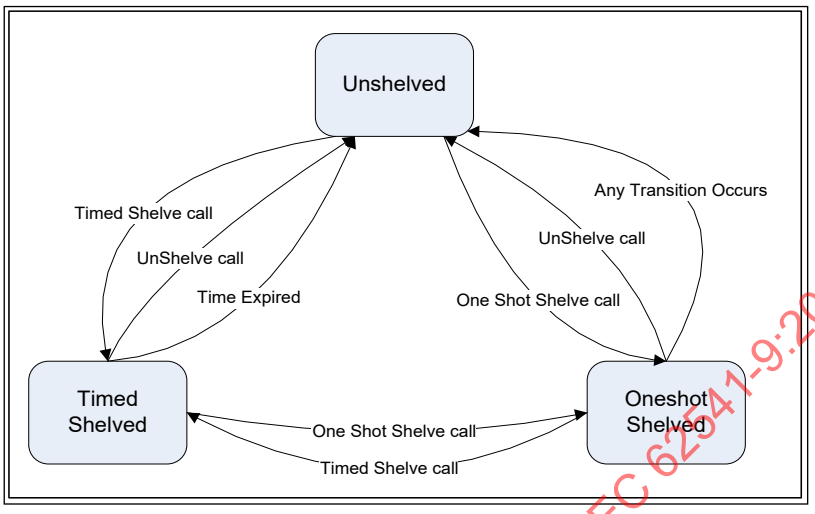
The state model supports two types of *Shelving*: *OneShotShelving* and *TimedShelving*. They are illustrated in Figure 14. The illustration includes the allowed transitions between the various substates. *Shelving* is an *Operator* initiated activity.

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

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

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

All but two transitions are caused by *Method* calls as illustrated in Figure 14. The "Time Expired" transition is simply a system generated transition that occurs when the time value defined as part of the "Timed Shelved Call" has expired. The "Any Transition Occurs" transition is also a system generated transition; this transition is generated when the *Condition* goes to an inactive state.

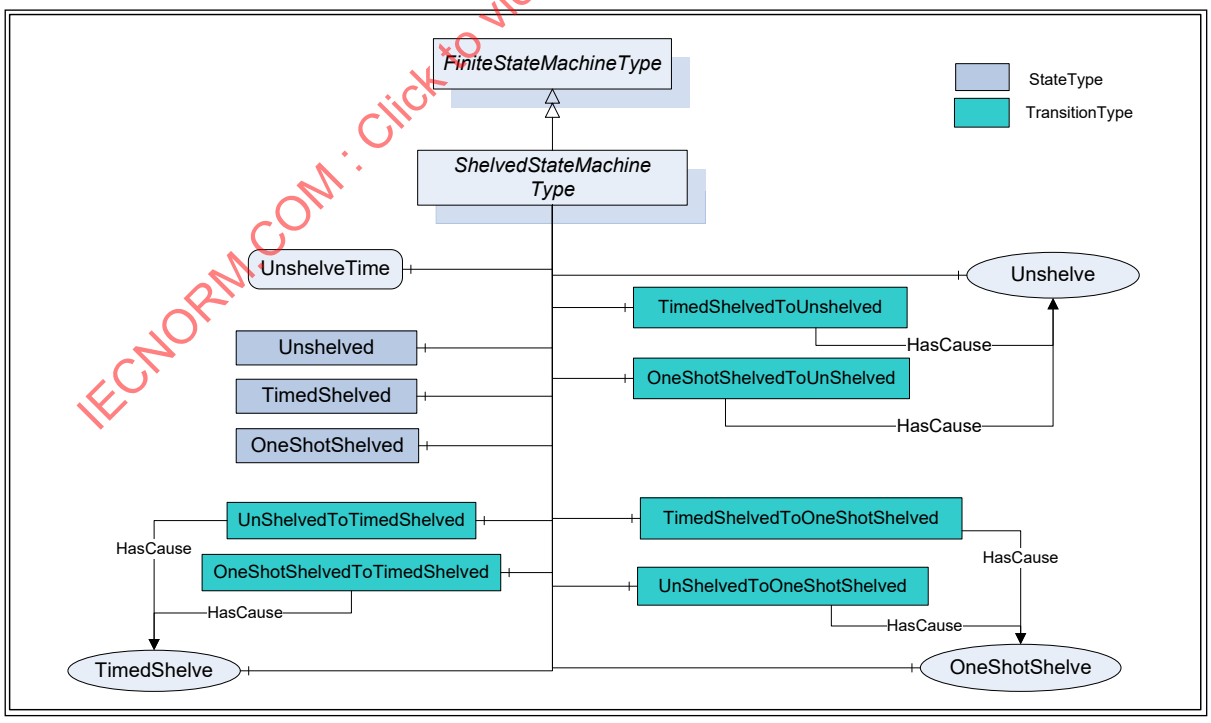


IEC

Figure 14 – Shelve state transitions

The ~~*ShelvedStateMachine*~~ *ShelvedStateMachineType* includes a hierarchy of substates. It supports all transitions between *Unshelved*, *OneShotShelved* and *TimedShelved*.

The state machine is illustrated in Figure 15 and formally defined in Table 49.



IEC

Figure 15 – ~~Shelved State Machine~~ ShelvedStateMachineType model

Table 49 – ~~ShelvedStateMachine~~ ShelvedStateMachineType definition

Attribute	Value				
BrowseName	ShelvedStateMachineType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the Finite*StateMachineType* defined in IEC 62541-5					
HasProperty	Variable	UnshelveTime	Duration	PropertyType	Mandatory
HasComponent	Object	Unshelved		StateType	~~Mandatory~~
HasComponent	Object	TimedShelved		StateType	~~Mandatory~~
HasComponent	Object	OneShotShelved		StateType	~~Mandatory~~
HasComponent	Object	UnshelvedToTimedShelved		TransitionType	~~Mandatory~~
HasComponent	Object	TimedShelvedToUnshelved		TransitionType	~~Mandatory~~
HasComponent	Object	TimedShelvedToOneShotShelved		TransitionType	~~Mandatory~~
HasComponent	Object	UnshelvedToOneShotShelved		TransitionType	~~Mandatory~~
HasComponent	Object	OneShotShelvedToUnshelved		TransitionType	~~Mandatory~~
HasComponent	Object	OneShotShelvedToTimedShelved		TransitionType	~~Mandatory~~
HasComponent	Method	TimedShelve	Defined in 5.8.10.3		Mandatory
HasComponent	Method	OneShotShelve	Defined in 5.8.10.4		Mandatory
HasComponent	Method	Unshelve	Defined in 5.8.10.2		Mandatory

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

This *FiniteStateMachine* supports three *Active* states; *Unshelved*, *TimedShelved* and *OneShotShelved*. It also supports six transitions. The states and transitions are described in Table 50. This *FiniteStateMachine* also supports three *Methods*; *TimedShelve*, *OneShotShelve* and *Unshelve*.

Table 50 – ~~ShelvedStateMachine~~ ShelvedStateMachineType transitions

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

5.8.10.2 Unshelve Method

~~*Unshelve* sets the *AlarmCondition* to the Unshelved state.~~

The *Unshelve Method* sets the instance of *AlarmConditionType* to the *Unshelved* state. Normally, the *MethodId* found in the *Shelving* child of the *Condition* instance and the *NodeId* of the *Shelving* object as the *ObjectId* are passed 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 *Unshelve Method* by specifying *ConditionId* as the *ObjectId*. The *Method* may not be called with an *ObjectId* of the *ShelvedStateMachineType Node*.

Signature

```
Unshelve( );
```

Method Result Codes in Table 51 (defined in Call Service).

Table 51 – Unshelve result codes

Result Code	Description
Bad_ConditionNotShelved	See Table 101 for the description of this result code.

Table 52 specifies the *AddressSpace* representation for the *Unshelve Method*.

Table 52 – Unshelve Method AddressSpace definition

Attribute	Value				
BrowseName	Unshelve				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionShelvingEventType	Defined in 5.10.7		

5.8.10.3 TimedShelve Method

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

The *TimedShelve Method* sets the instance of *AlarmConditionType* to the *TimedShelved* state (parameters are defined in Table 53 and result codes are described in Table 54). Normally, the *MethodId* found in the *Shelving* child of the *Condition* instance and the *NodeId* of the *Shelving* object as the *ObjectId* are passed 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 *TimedShelve Method* by specifying *ConditionId* as the *ObjectId.* The *Method* may not be called with an *ObjectId* of the *ShelvedStateMachineType Node*.

Signature

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

Table 53 – TimedShelve parameters

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

Method Result Codes (defined in Call Service).

Table 54 – TimedShelve result codes

Result Code	Description
Bad_ConditionAlreadyShelved	See Table 101 for the description of this result code. The *Alarm* is already in *TimedShelved* state and the system does not allow a reset of the shelved timer.
Bad_ShelvingTimeOutOfRange	See Table 101 for the description of this result code.

Comments

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

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

Table 55 specifies the *AddressSpace* representation for the *TimedShelve Method*.

Table 55 – TimedShelve Method AddressSpace definition

Attribute	Value				
BrowseName	TimedShelve				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGeneratesEvent	ObjectType	AuditConditionShelvingEventType	Defined in 5.10.7		

5.8.10.4 OneShotShelve Method

~~*OneShotShelve* sets the *AlarmCondition* to the OneShotShelved state.~~

The *OneShotShelve Method* sets the instance of *AlarmConditionType* to the *OneShotShelved* state. Normally, the *MethodId* found in the *Shelving* child of the *Condition* instance and the *NodeId* of the *Shelving* object as the *ObjectId* are passed 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 *OneShotShelve Method* by specifying *ConditionId* as the *ObjectId*. The *Method* may not be called with an *ObjectId* of the *ShelvedStateMachineType Node*.

Signature

```
OneShotShelve( );
```

Method Result Codes are defined in Table 56 (status code field is defined in *Call* Service).

Table 56 – OneShotShelve result codes

Result Code	Description
~~Bad_AlreadyShelved~~ Bad_ConditionAlreadyShelved	See Table 101 for the description of this result code. The *Alarm* is already in *OneShotShelved* state.

Table 57 specifies the *AddressSpace* representation for the *OneShotShelve Method*.

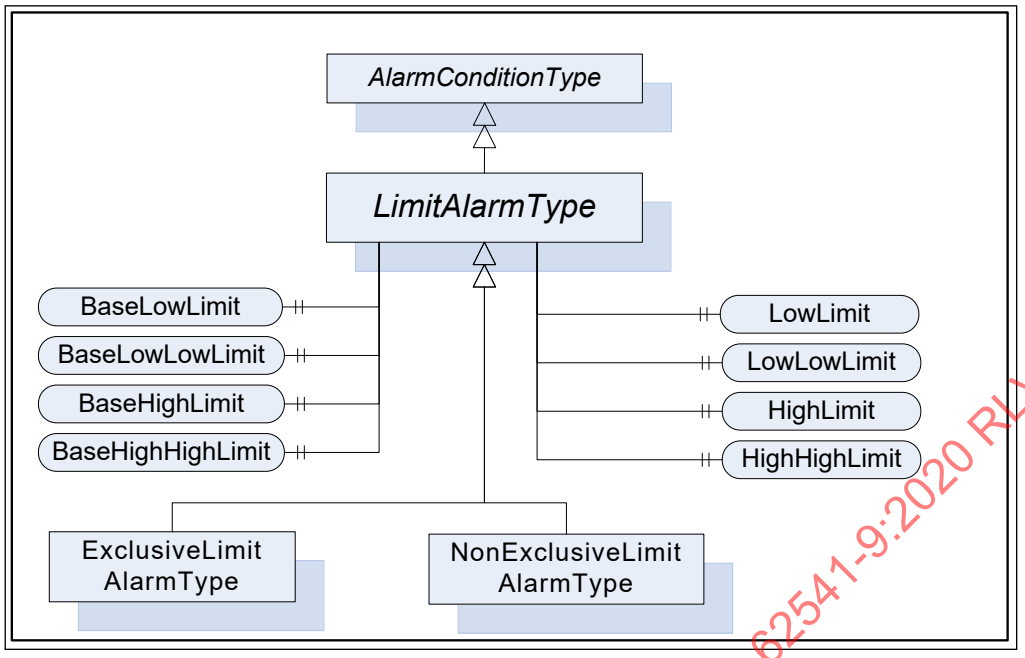
Table 57 – OneShotShelve Method AddressSpace definition

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

5.8.11 LimitAlarmType

Alarms ~~can~~ may be modelled with multiple exclusive substates and assigned limits or they may be modelled with nonexclusive limits that ~~can~~ may be used to group multiple states together.

The *LimitAlarmType* is an abstract type used to provide a base *Type* for ~~*AlarmConditions*~~ *AlarmConditionTypes* with multiple limits. The *LimitAlarmType* is illustrated in Figure 16.



IEC

Figure 16 – LimitAlarmType

The *LimitAlarmType* is formally defined in Table 58.

Table 58 – LimitAlarmType definition

Attribute	Value				
BrowseName	LimitAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the AlarmConditionType defined in 5.8.2.					
HasSubtype	ObjectType	ExclusiveLimitAlarmType	Defined in 5.8.12.3		
HasSubtype	ObjectType	NonExclusiveLimitAlarmType	Defined in 5.8.13		
HasProperty	Variable	HighHighLimit	Double	PropertyType	Optional
HasProperty	Variable	HighLimit	Double	PropertyType	Optional
HasProperty	Variable	LowLimit	Double	PropertyType	Optional
HasProperty	Variable	LowLowLimit	Double	PropertyType	Optional
HasProperty	Variable	BaseHighHighLimit	Double	PropertyType	Optional
HasProperty	Variable	BaseHighLimit	Double	PropertyType	Optional
HasProperty	Variable	BaseLowLimit	Double	PropertyType	Optional
HasProperty	Variable	BaseLowLowLimit	Double	PropertyType	Optional

Four optional limits are defined which configure the states of the derived limit *Alarm* Types. These *Properties* shall be set for any *Alarm* limits that are exposed by the derived limit *Alarm* types. These *Properties* are listed as optional but at least one is required. For cases where an underlying system cannot provide the actual value of a limit, the limit *Property* shall still be provided, but will have its *AccessLevel* set to not readable. It is assumed that the limits are described using the same Engineering Unit that is assigned to the variable that is the source of the *Alarm*. For Rate of change limit *Alarms*, it is assumed this rate is units per second unless otherwise specified.

Four optional base limits are defined which are used for *AdaptiveAlarming*. They contain the configured *Alarm* limit. If a *Server* supports *AdaptiveAlarming* for *Alarm* limits, the corresponding base *Alarm* limit shall be provided for any limits that are exposed by the derived limit Alarm types. The value of this property is the value of the limit to which an *AdaptiveAlarm* can be reset if any algorithmic changes need to be discarded.

The *Alarm* limits listed may cause an *Alarm* to be generated when a value equals the limit or it may generate the *Alarm* when the limit is exceeded, (i.e. the Value is above the limit for *HighLimit* and below the limit for *LowLimit*). The exact behaviour when the value is equal to the limit is *Server*-specific.

The *Variable* that is the source of the *LimitAlarmType Alarm* shall be a scalar. This *LimitAlarmType* can be subtyped if the *Variable* that is the source is an array. The subtype shall describe the expected behaviour with respect to limits and the array values. Some possible options:

- if any element of the array exceeds the limit, an *Alarm* is generated,
- if all elements exceed the limit, an *Alarm* is generated,
- the limits may also be an array, in which case if any array limit is exceeded by the corresponding source array element, an *Alarm* is generated.

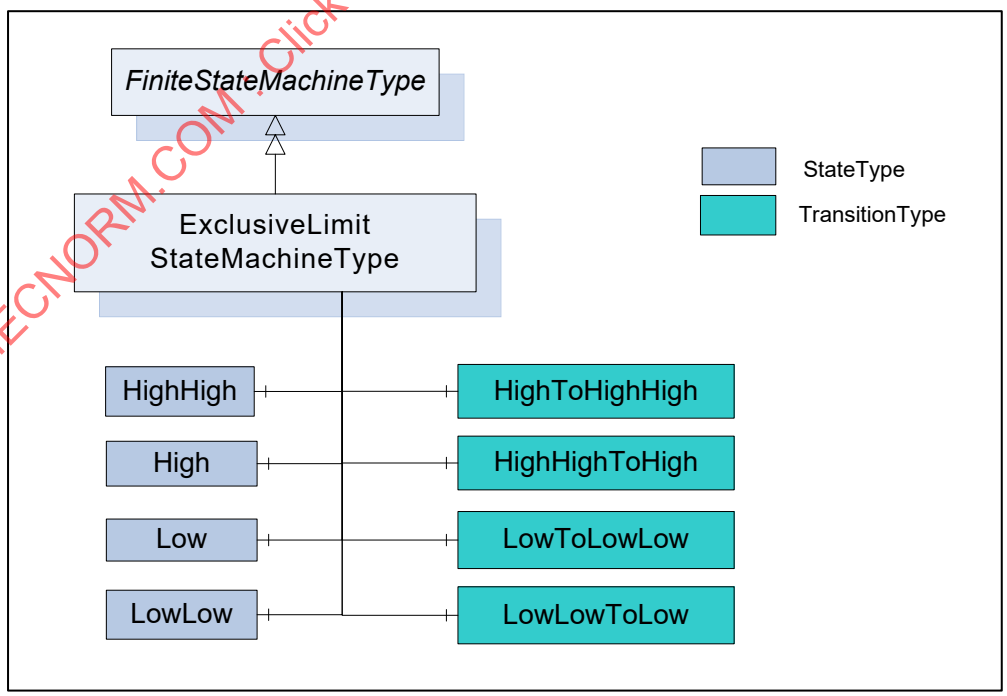
5.8.12 Exclusive limit types

5.8.12.1 Overview

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

5.8.12.2 ExclusiveLimitStateMachineType

The *ExclusiveLimitStateMachineType* defines the state machine used by ~~*AlarmTypes*~~ *AlarmConditionTypes* that handle multiple mutually exclusive limits. It is illustrated in Figure 17.



IEC

Figure 17 – ~~ExclusiveLimitStateMachine~~ ExclusiveLimitStateMachineType

It is created by extending the *FiniteStateMachineType*. It is formally defined in Table 59 and the state transitions are described in Table 60.

Table 59 – ExclusiveLimitStateMachineType definition

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

Table 60 – ExclusiveLimitStateMachineType transitions

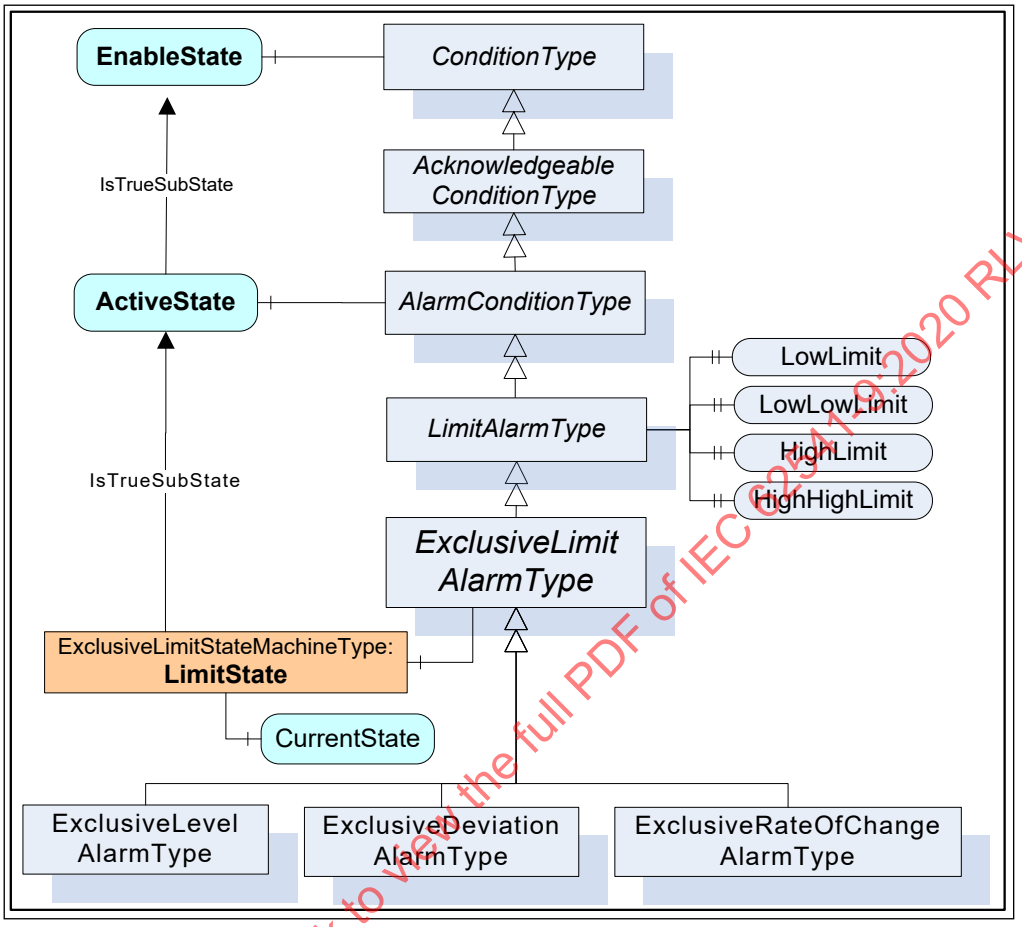
BrowseName	References	BrowseName	TypeDefinition
Transitions			
HighHighToHigh	FromState	HighHigh	StateType
	ToState	High	StateType
	HasEffect	AlarmConditionType	
HighToHighHigh	FromState	High	StateType
	ToState	HighHigh	StateType
	HasEffect	AlarmConditionType	
LowLowToLow	FromState	LowLow	StateType
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

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

The *LowLow*, *Low*, *High*, *HighHigh* are typical for many industries. Vendors ~~can~~ may introduce substate models that include additional limits; they may also omit limits in an instance. If a model omits states or transitions in the *StateMachine*, it is recommended that they provide the optional *Property AvailableStates* and/or *AvailableTransitions* (see IEC 62541-5).

5.8.12.3 ExclusiveLimitAlarmType

The *ExclusiveLimitAlarmType* is used to specify the common behaviour for *Alarm Types* with multiple mutually exclusive limits. The *ExclusiveLimitAlarmType* is illustrated in Figure 18.



IEC

Figure 18 – ExclusiveLimitAlarmType

The *ExclusiveLimitAlarmType* is formally defined in Table 61.

Table 61 – ExclusiveLimitAlarmType definition

Attribute	Value				
BrowseName	ExclusiveLimitAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the LimitAlarmType defined in 5.8.11.					
HasSubtype	ObjectType	ExclusiveLevelAlarmType	Defined in 5.8.14.3		
HasSubtype	ObjectType	ExclusiveDeviationAlarmType Type	Defined in 5.8.15.3		
HasSubtype	ObjectType	ExclusiveRateOfChangeAlarm Type	Defined in 5.8.16.3		
HasComponent	Object	LimitState		*ExclusiveLimitStateMachineType*	Mandatory

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

5.8.13 NonExclusiveLimitAlarmType

The *NonExclusiveLimitAlarmType* is used to specify the common behaviour for *Alarm Types* with multiple non-exclusive limits. The *NonExclusiveLimitAlarmType* is illustrated in Figure 19.

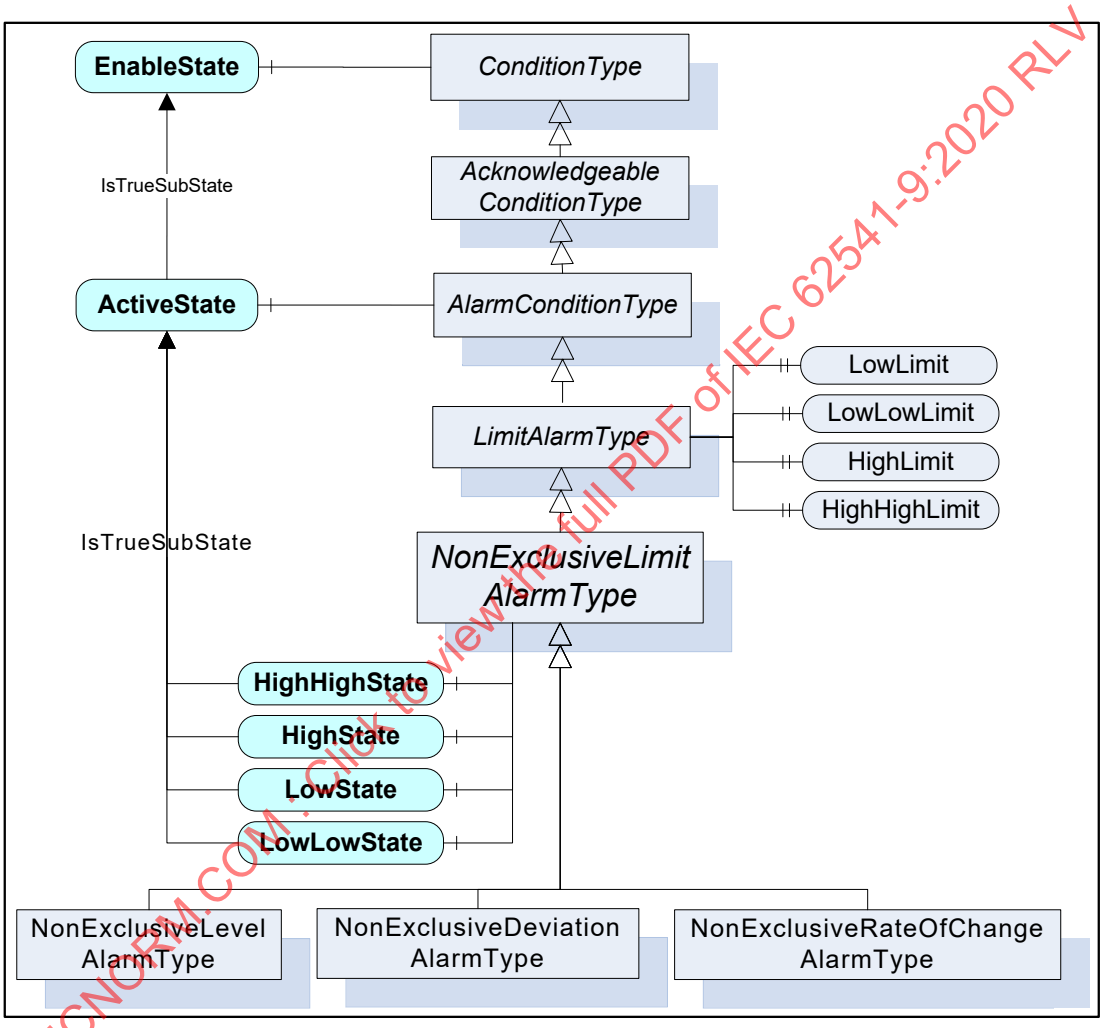


Figure 19 – NonExclusiveLimitAlarmType

The *NonExclusiveLimitAlarmType* is formally defined in Table 62.

Table 62 – NonExclusiveLimitAlarmType definition

Attribute	Value				
BrowseName	NonExclusiveLimitAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the LimitAlarmType defined in 5.8.11.					
HasSubtype	ObjectType	NonExclusiveLevelAlarmType	Defined in 5.8.14.2		
HasSubtype	ObjectType	NonExclusiveDeviationAlarmType	Defined in 5.8.15.2		
HasSubtype	ObjectType	NonExclusiveRateOfChangeAlarmType	Defined in 5.8.16.2		
HasComponent	Variable	HighHighState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Variable	HighState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Variable	LowState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Variable	LowLowState	LocalizedText	TwoStateVariableType	Optional

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

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

5.8.14 Level Alarm

5.8.14.1 Overview

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

5.8.14.2 NonExclusiveLevelAlarmType

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

The *NonExclusiveLevelAlarmType* is based on the *NonExclusiveLimitAlarmType*. It is formally defined in Table 63.

Table 63 – NonExclusiveLevelAlarmType definition

Attribute	Value				
BrowseName	NonExclusiveLevelAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the NonExclusiveLimitAlarmType defined in 5.8.13.					

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

5.8.14.3 ExclusiveLevelAlarmType

The *ExclusiveLevelAlarmType* is a special level *Alarm* utilized with multiple mutually exclusive limits. It is formally defined in Table 64.

Table 64 – ExclusiveLevelAlarmType definition

Attribute	Value				
BrowseName	ExclusiveLevelAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherits the Properties of the ExclusiveLimitAlarmType defined in 5.8.12.3.					

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

5.8.15 Deviation Alarm

5.8.15.1 Overview

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

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

For example, if a set point had a value of 10, a high deviation *Alarm* limit of 2 and a low deviation *Alarm* limit of −1, then the low substate is entered if the process value drops below 9; the high substate is entered if the process value raises above 12. If the set point were changed to 11 then the new deviation values would be 10 and 13 respectively. The set point may be fixed by a configuration, adjusted by an *Operator* or it may be adjusted by an algorithm, the actual functionality exposed by the set point is application specific. The deviation *Alarm* may also be used to report a problem between a redundant data source where the difference between the primary source and the secondary source exceeds the included limit. In this case, the *SetpointNode* would point to the secondary source.

5.8.15.2 NonExclusiveDeviationAlarmType

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

The *NonExclusiveDeviationAlarmType* is based on the *NonExclusiveLimitAlarmType*. It is formally defined in Table 65.

Table 65 – NonExclusiveDeviationAlarmType definition

Attribute	Value				
BrowseName	NonExclusiveDeviationAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the NonExclusiveLimitAlarmType defined in 5.8.13.					
HasProperty	Variable	SetpointNode	NodeId	PropertyType	Mandatory
HasProperty	Variable	BaseSetpointNode	NodeId	PropertyType	Optional

The *SetpointNode Property* provides the *NodeId* of the set point used in the deviation calculation. In cases where the *Alarm* is generated by an underlying system and if the *Variable* is not in the *AddressSpace*, a NULL *NodeId* shall be provided.

The *BaseSetpointNode Property* provides the *NodeId* of the original or base setpoint. The value of this node is the value of the setpoint to which an *AdaptiveAlarm* may be reset if any algorithmic changes need to be discarded. The value of this node usually contains the originally configured set point.

5.8.15.3 ExclusiveDeviationAlarmType

The *ExclusiveDeviationAlarmType* is utilized with multiple mutually exclusive limits. It is formally defined in Table 66.

Table 66 – ExclusiveDeviationAlarmType definition

Attribute	Value				
BrowseName	ExclusiveDeviationAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Inherits the *Properties* of the *ExclusiveLimitAlarmType* defined in 5.8.12.3.					
HasProperty	Variable	SetpointNode	NodeId	PropertyType	Mandatory
HasProperty	Variable	BaseSetpointNode	NodeId	PropertyType	Optional

The *SetpointNode Property* provides the *NodeId* of the set point used in the *Deviation* calculation. If this *Variable* is not in the *AddressSpace*, a NULL *NodeId* shall be provided.

The *BaseSetpointNode Property* provides the *NodeId* of the original or base setpoint. The value of this node is the value of the set point to which an *AdaptiveAlarm* may be reset if any algorithmic changes need to be discarded. The value of this node usually contains the originally configured set point.

5.8.16 Rate of change Alarms

5.8.16.1 Overview

A *Rate* of *Change Alarm* is commonly used to report an unusual change or lack of change in a measured value related to the speed at which the value has changed. The *Rate* of *Change Alarm* becomes active when the rate at which the value changes exceeds or drops below a defined limit.

A *Rate* of *Change* is measured in some time unit, such as seconds or minutes and some unit of measure, such as percent or metre. For example, a tank may have a High limit for the *Rate* of *Change* of its level (measured in metres) which would be 4 metres per minute. If the tank

level changes at a rate that is greater than 4 metres per minute, then the High substate is entered.

5.8.16.2 NonExclusiveRateOfChangeAlarmType

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

The *NonExclusiveRateOfChangeAlarmType* is based on the *NonExclusiveLimitAlarmType*. It is formally defined in Table 67.

Table 67 – NonExclusiveRateOfChangeAlarmType definition

Attribute	Value				
BrowseName	NonExclusiveRateOfChangeAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the NonExclusiveLimitAlarmType defined in clause 5.8.13.					
HasProperty	Variable	EngineeringUnits	EUInformation	PropertyType	Optional

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

EngineeringUnits provides the engineering units associated with the limits values. If this is not provided, the assumed Engineering Unit is the same as the EU associated with the parent variable per second e.g. if parent is meters, this unit is meters/second.

5.8.16.3 ExclusiveRateOfChangeAlarmType

ExclusiveRateOfChangeAlarmType is utilized with multiple mutually exclusive limits. It is formally defined in Table 68.

Table 68 – ExclusiveRateOfChangeAlarmType definition

Attribute	Value				
BrowseName	ExclusiveRateOfChangeAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherits the *Properties* of the *ExclusiveLimitAlarmType* defined in 5.8.12.3.					
HasProperty	Variable	EngineeringUnits	EUInformation	PropertyType	Optional

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

EngineeringUnits provides the engineering units associated with the limits values. If this is not provided, the assumed Engineering Unit is the same as the EU associated with the parent variable per second; e.g. if parent is metres, this unit is metres/second.

5.8.17 Discrete Alarms

5.8.17.1 DiscreteAlarmType

The *DiscreteAlarmType* is used to classify *Types* into *Alarm Conditions* where the input for the *Alarm* may take on only a certain number of possible values (e.g. True/False, running/stopped/terminating). The *DiscreteAlarmType* with subtypes defined in this document is illustrated in Figure 20. It is formally defined in Table 69.

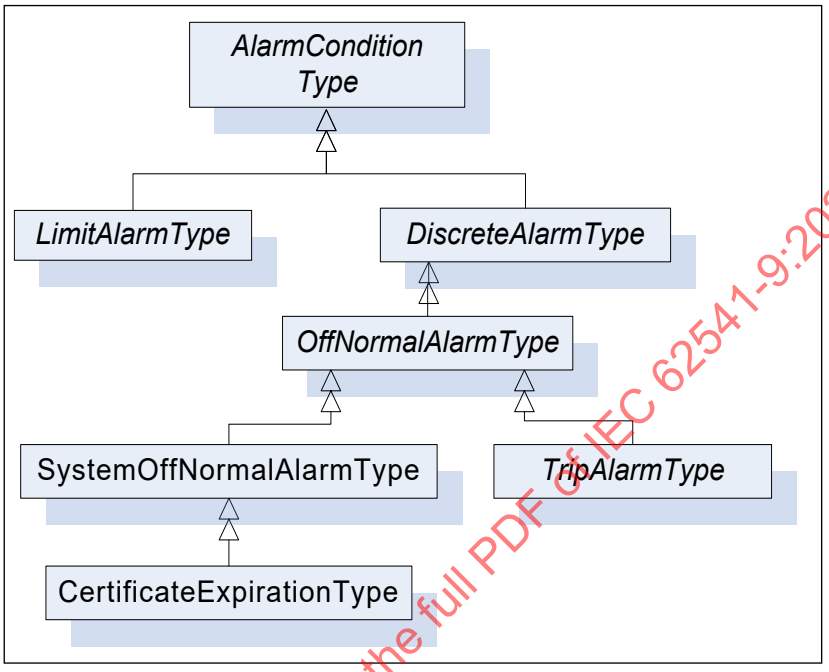


Figure 20 – DiscreteAlarmType Hierarchy

Table 69 – DiscreteAlarmType definition

Attribute	Value				
BrowseName	DiscreteAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the AlarmConditionType defined in 5.8.2.					
HasSubtype	*ObjectType*	OffNormalAlarmType	Defined in 5.8.15		

5.8.17.2 OffNormalAlarmType

The *OffNormalAlarmType* is a specialization of the *DiscreteAlarmType* intended to represent a discrete *Condition* that is considered to be not normal. It is formally defined in Table 70. This subtype is usually used to indicate that a discrete value is in an *Alarm* state, it is active as long as a non-normal value is present.

Table 70 – OffNormalAlarmType Definition

Attribute	Value				
BrowseName	OffNormalAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the DiscreteAlarmType defined in 5.8.17.1					
HasSubtype	*ObjectType*	TripAlarmType	Defined in 5.8.17.4		
HasSubtype	*ObjectType*	SystemOffNormalAlarmType	Defined in 5.8.17.3		
HasProperty	*Variable*	NormalState	NodeId	PropertyType	Mandatory

The *NormalState Property* is a *Property* that points to a *Variable* which has a value that corresponds to one of the possible values of the *Variable* pointed to by the *InputNode Property* where the *NormalState Property Variable* value is the value that is considered to be the normal state of the *Variable* pointed to by the *InputNode Property*. When the value of the *Variable* referenced by the *InputNode Property* is not equal to the value of the *NormalState Property* the *Alarm* is *Active*. If this *Variable* is not in the *AddressSpace*, a NULL *NodeId* shall be provided.

5.8.17.3 SystemOffNormalAlarmType

This *Condition* is used by a *Server* to indicate that an underlying system that is providing *Alarm* information is having a communication problem and that the *Server* may have invalid or incomplete *Condition* state in the *Subscription*. Its representation in the *AddressSpace* is formally defined in Table 71.

Table 71 – SystemOffNormalAlarmType definition

Attribute	Value				
BrowseName	SystemOffNormalAlarmType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasSubtype	*ObjectType*	CertificateExpirationAlarmType	Defined in 5.8.17.7		
Subtype of the *OffNormalAlarmType*, i.e. it has HasProperty *References* to the same *Nodes*.					

5.8.17.4 TripAlarmType

The *TripAlarmType* is a specialization of the *OffNormalAlarmType* intended to represent an equipment trip *Condition*. The *Alarm* becomes active when the monitored piece of equipment experiences some abnormal fault such as a motor shutting down due to an overload condition. It is formally defined in Table 72. This *Type* is mainly used for categorization.

Table 72 – TripAlarmType definition

Attribute	Value				
BrowseName	TripAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the OffNormalAlarmType defined in 5.8.17.2.					

5.8.17.5 InstrumentDiagnosticAlarmType

The *InstrumentDiagnosticAlarmType* is a specialization of the *OffNormalAlarmType* intended to represent a fault in a field device. The *Alarm* becomes active when the monitored device experiences a fault such as a sensor failure. It is formally defined in Table 73. This *Type* is mainly used for categorization.

Table 73 – InstrumentDiagnosticAlarmType definition

Attribute	Value				
BrowseName	InstrumentDiagnosticAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the OffNormalAlarmType defined in clause 5.8.17.2.					

5.8.17.6 SystemDiagnosticAlarmType

The *SystemDiagnosticAlarmType* is a specialization of the *OffNormalAlarmType* intended to represent a fault in a system or sub-system. The *Alarm* becomes active when the monitored system experiences a fault. It is formally defined in Table 74. This *Type* is mainly used for categorization.

Table 74 – SystemDiagnosticAlarmType definition

Attribute	Value				
BrowseName	SystemDiagnosticAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the OffNormalAlarmType defined in 5.8.17.2.					

5.8.17.7 CertificateExpirationAlarmType

This *SystemOffNormalAlarmType* is raised by the *Server* when the *Server's* Certificate is within the *ExpirationLimit* of expiration. This *Alarm* automatically returns to normal when the certificate is updated.

The *SystemOffNormalAlarmType* is formally defined in Table 75.

Table 75 – CertificateExpirationAlarmType definition

Attribute	Value				
BrowseName	CertificateExpirationAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the SystemOffNormalAlarmType defined in 5.8.17.3					
HasProperty	Variable	ExpirationDate	DateTime	PropertyType	Mandatory
HasProperty	Variable	ExpirationLimit	Duration	PropertyType	Optional
HasProperty	Variable	CertificateType	NodeId	PropertyType	Mandatory
HasProperty	Variable	Certificate	ByteString	PropertyType	Mandatory

ExpirationDate is the date and time this certificate will expire.

ExpirationLimit is the time interval before the *ExpirationDate* at which this *Alarm* will trigger. This shall be a positive number. If the property is not provided, a default of 2 weeks shall be used.

CertificateType – See Part 12 for definition of *CertificateType*.

Certificate is the certificate that is about to expire.

5.8.18 DiscrepancyAlarmType

The *DiscrepancyAlarmType* is commonly used to report an action that did not occur within an expected time range.

The *DiscrepancyAlarmType* is based on the *AlarmConditionType*. It is formally defined in Table 76.

Table 76 – DiscrepancyAlarmType definition

Attribute	Value				
BrowseName	DiscrepancyAlarmType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AlarmConditionType* defined in 5.8.2.					
HasProperty	Variable	TargetValueNode	NodeId	PropertyType	Mandatory
HasProperty	Variable	ExpectedTime	Duration	PropertyType	Mandatory
HasProperty	Variable	Tolerance	Double	PropertyType	Optional

The *TargetValueNode Property* provides the *NodeId* of the *Variable* that is used for the target value.

The *ExpectedTime Property* provides the *Duration* within which the value pointed to by the InputNode shall equal the value specified by the *TargetValueNode* (or be within the *Tolerance* range, if specified).

The *Tolerance Property* is a value that may be added to or subtracted from the *TargetValueNode's* value, providing a range that the value can be in without generating the *Alarm*.

A *DiscrepancyAlarmType* may be used to indicate a motor has not responded to a start request within a given time, or that a process value has not reached a given value after a setpoint change within a given time interval.

The *DiscrepancyAlarmType* shall return to normal when the value has reached the target value.

5.9 ConditionClasses

5.9.1 Overview

Conditions are used in specific application domains like Maintenance, System or Process. The *ConditionClass* hierarchy is used to specify domains and is orthogonal to the *ConditionType* hierarchy. The *ConditionClassId Property* of the *ConditionType* is used to assign a *Condition* to a *ConditionClass*. *Clients* ~~can~~ may use this *Property* to filter out essential classes. OPC UA defines the base *ObjectType* for all *ConditionClasses* and a set of common classes used across many industries. Figure 21 informally describes the hierarchy of *ConditionClass Types* defined in this document.

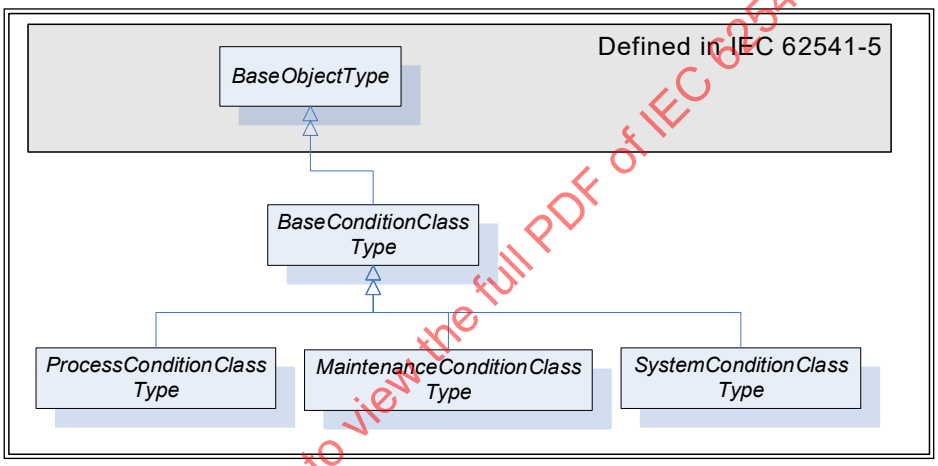


Figure 21 – ConditionClass type hierarchy

ConditionClasses are not representations of *Objects* in the underlying system and, therefore, only exist as *Type Nodes* in the *Address Space*.

5.9.2 BaseConditionClassType

BaseConditionClassType is used as class whenever a *Condition* cannot be assigned to a more concrete class. *Servers* should use a more specific *ConditionClass*, if possible. All *ConditionClass Types* derive from *BaseConditionClassType*. It is formally defined in Table 77.

Table 77 – BaseConditionClassType definition

Attribute	Value				
BrowseName	BaseConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the BaseObjectType defined in IEC 62541-5.					

5.9.3 ProcessConditionClassType

The *ProcessConditionClassType* is used to classify *Conditions* related to the process itself. Examples of a process would be a control system in a boiler, or the instrumentation associated with a chemical plant or paper machine. The *ProcessConditionClassType* is formally defined in Table 78.

Table 78 – ProcessConditionClassType definition

Attribute	Value				
BrowseName	ProcessConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the BaseConditionClassType defined in 5.9.2.					

5.9.4 MaintenanceConditionClassType

The *MaintenanceConditionClassType* is used to classify *Conditions* related to maintenance. Examples of maintenance would be Asset Management systems or conditions, which occur in process control systems, which are related to calibration of equipment. The *MaintenanceConditionClassType* is formally defined in Table 79. No further definition is provided here. It is expected that other standards development groups will define domain-specific subtypes.

Table 79 – MaintenanceConditionClassType definition

Attribute	Value				
BrowseName	MaintenanceConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the BaseConditionClassType defined in 5.9.2.					

5.9.5 SystemConditionClassType

The *SystemConditionClassType* is used to classify *Conditions* related to the System. It is formally defined in Table 80. System *Conditions* occur in the controlling or monitoring system process. Examples of System related items could include available disk space on a computer, Archive media availability, network loading issues or a controller error. No further definition is provided here. It is expected that other standards development groups or vendors will define domain-specific subtypes.

Table 80 – SystemConditionClassType definition

Attribute	Value				
BrowseName	SystemConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseConditionClassType* defined in 5.9.2.					

5.9.6 SafetyConditionClassType

The *SafetyConditionClassType* is used to classify *Conditions* related to safety. It is formally defined in Table 81.

Safety *Conditions* occur in the controlling or monitoring system process. Examples of safety related items could include emergency shutdown systems or fire suppression systems.

Table 81 – SafetyConditionClassType definition

Attribute	Value				
BrowseName	SafetyConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseConditionClassType* defined in 5.9.2.					

5.9.7 HighlyManagedAlarmConditionClassType

In *Alarm* systems some *Alarms* may be classified as highly managed *Alarms*. This class of *Alarm* requires special handling that varies according to the individual requirements. It might require individual acknowledgement or not allow suppression or any of a number of other special behaviours. The *HighlyManagedAlarmConditionClassType* is used to classify *Conditions* as highly managed *Alarms*. It is formally defined in Table 82.

Table 82 – HighlyManagedAlarmConditionClassType definition

Attribute	Value				
BrowseName	HighlyManagedAlarmConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseConditionClassType* defined in 5.9.2.					

5.9.8 TrainingConditionClassType

The *TrainingConditionClassType* is used to classify *Conditions* related to training system or training exercises. It is formally defined in Table 83. These *Conditions* typically occur in a training system or are generated as part of a simulation for a training exercise. Training *Conditions* might be process or system conditions. It is expected that other standards development groups or vendors will define domain-specific subtypes.

Table 83 – TrainingConditionClassType definition

Attribute	Value				
BrowseName	TrainingConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseConditionClassType* defined in 5.9.2.					

5.9.9 StatisticalConditionClassType

The *StatisticalConditionClassType* is used to classify *Conditions* related that are based on statistical calculations. It is formally defined in Table 84. These *Conditions* are generated as part of a statistical analysis. They might be any of an *Alarm* number of types.

Table 84 – StatisticalConditionClassType definition

Attribute	Value				
BrowseName	StatisticalConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseConditionClassType* defined in 5.9.2.					

5.9.10 TestingConditionSubClassType

The *TestingConditionSubClassType* is used to classify *Conditions* related to testing of an *Alarm* system or *Alarm* function. It is formally defined in Table 85. Testing *Conditions* might include a condition to test an alarm annunciation such as a horn or other panel. It might also be used to temporarily reclassify a *Condition* to check response times or suppression logic. It is expected that other standards development groups or vendors will define domain-specific subtypes.

Table 85 – TestingConditionSubClassType definition

Attribute	Value				
BrowseName	TestingConditionSubClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseConditionClassType* defined in 5.9.2.					

5.10 Audit Events

5.10.1 Overview

Following are subtypes of *AuditUpdateMethodEventTypes* that will be generated in response to the *Methods* defined in this document. They are illustrated in Figure 22.

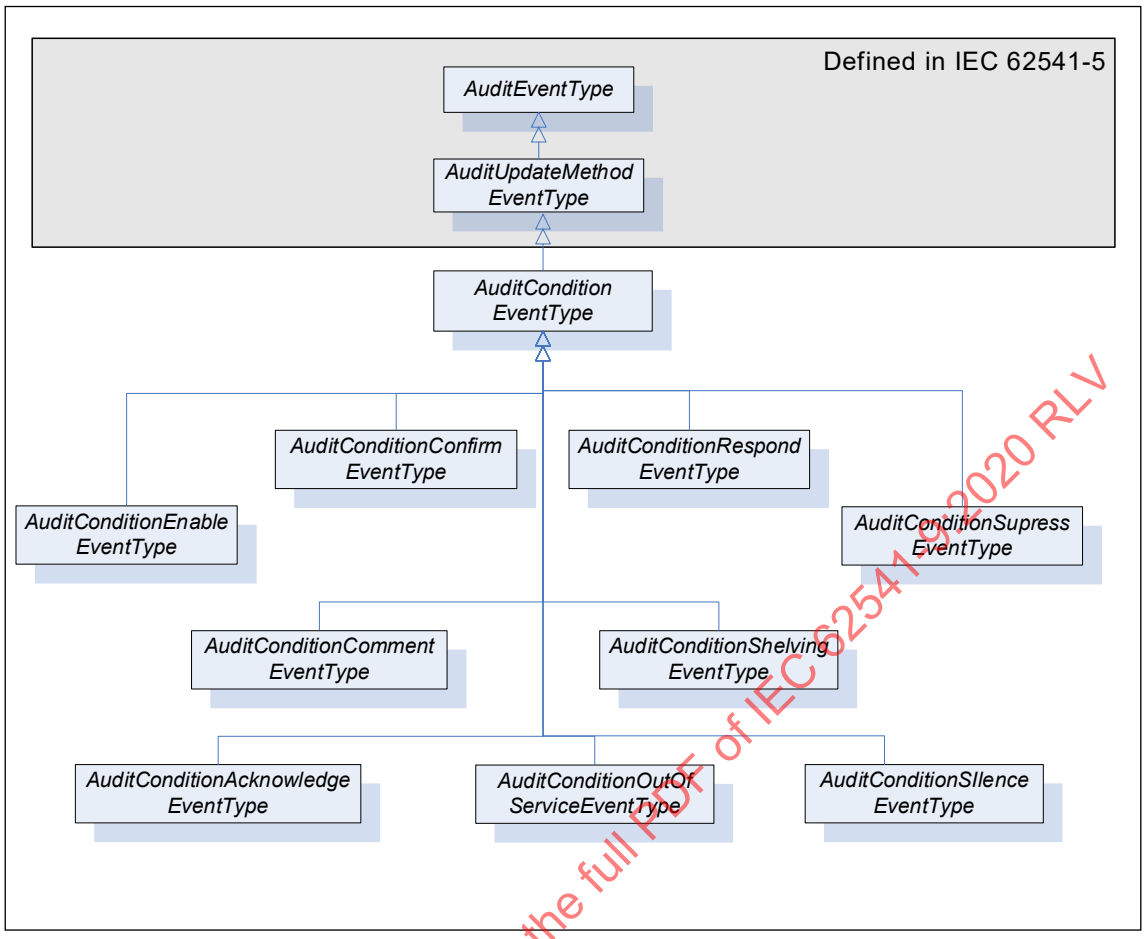


Figure 22 – AuditEvent hierarchy

AuditConditionEventTypes are normally used in response to a *Method* call. However, these *Events* shall also be notified if the functionality of such a *Method* is performed by some other *Server*-specific means. In this case, the *SourceName Property* shall contain a proper description of this internal means and the other *Properties* should be filled in as described for the given *EventType*.

5.10.2 AuditConditionEventType

This *EventType* is used to subsume all *AuditConditionEventTypes*. It is formally defined in Table 86.

Table 86 – AuditConditionEventType definition

Attribute	Value				
BrowseName	AuditConditionEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditUpdateMethodEventType* defined in IEC 62541-5					

AuditConditionEventTypes inherit all *Properties* of the *AuditUpdateMethodEventType* defined in IEC 62541-5. Unless a subtype overrides the definition, the inherited *Properties* of the *Condition* will be used as defined.

- The inherited *Property SourceNode* shall be filled with the *ConditionId*.
- The *SourceName* shall be "Method/" and the name of the *Service* that generated the *Event* (e.g. *Disable*, Enable, *Acknowledge,* etc.*)*.

This *EventType* can be further customized to reflect particular *Condition* related actions.

5.10.3 AuditConditionEnableEventType

This *EventType* is used to indicate a change in the enabled state of a *Condition* instance. It is formally defined in Table 87.

Table 87 – AuditConditionEnableEventType definition

Attribute		Value			
BrowseName		AuditConditionEnableEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

The SourceName shall indicate Method/Enable or Method/Disable. If the audit *Event* is not the result of a *Method* call, but due to an internal action of the *Server,* the *SourceName* shall reflect Enable or Disable, it may be preceded by an appropriate description such as "Internal/Enable" or "Remote/Enable".

5.10.4 AuditConditionCommentEventType

This *EventType* is used to report an *AddComment* action. It is formally defined in Table 88.

Table 88 – AuditConditionCommentEventType definition

Attribute		Value			
BrowseName		AuditConditionCommentEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	ConditionEventId	ByteString	PropertyType	Mandatory
HasProperty	Variable	Comment	LocalizedText	PropertyType	Mandatory
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

The *ConditionEventId* field shall contain the id of the event for which the comment was added.

The *Comment* contains the actual comment that was added.

5.10.5 AuditConditionRespondEventType

This *EventType* is used to report a *Respond* action (see 5.6). It is formally defined in Table 89.

Table 89 – AuditConditionRespondEventType definition

Attribute		Value			
BrowseName		AuditConditionRespondEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	SelectedResponse	Uint32	PropertyType	Mandatory
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

The SelectedResponse field shall contain the response that was selected.

5.10.6 AuditConditionAcknowledgeEventType

This *EventType* is used to indicate acknowledgement or confirmation of one or more *Conditions*. It is formally defined in Table 90.

Table 90 – AuditConditionAcknowledgeEventType definition

Attribute		Value			
BrowseName		~~AuditConditionCommentEventType~~ AuditConditionAcknowledgeEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	ConditionEventId	ByteString	PropertyType	Mandatory
HasProperty	Variable	Comment	LocalizedText	PropertyType	Mandatory
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

The *ConditionEventId* field shall contain the id of the *Event* that was acknowledged.

The *Comment* contains the actual comment that was added; it may be a blank comment or a NULL.

5.10.7 AuditConditionConfirmEventType

This *EventType* is used to report a *Confirm* action. It is formally defined in Table 91.

Table 91 – AuditConditionConfirmEventType definition

Attribute		Value			
BrowseName		~~AuditConditionCommentEventType~~ AuditConditionConfirmEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	ConditionEventId	ByteString	PropertyType	Mandatory
HasProperty	Variable	Comment	LocalizedText	PropertyType	Mandatory
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

The *ConditionEventId* field shall contain the id of the *Event* that was confirmed.

The *Comment* contains the actual comment that was added; it may be a blank comment or a NULL.

5.10.8 AuditConditionShelvingEventType

This *EventType* is used to indicate a change to the *Shelving* state of a *Condition* instance. It is formally defined in Table 92.

Table 92 – AuditConditionShelvingEventType definition

Attribute			Value		
BrowseName			AuditConditionShelvingEventType		
IsAbstract			False		
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	ShelvingTime	Duration	PropertyType	Optional
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

If the *Method* indicates a *TimedShelve* operation, the *ShelvingTime* field shall contain duration for which the *Alarm* is to be shelved. For other *Shelving Methods*, this parameter may be omitted or NULL.

5.10.9 AuditConditionSuppressionEventType

This *EventType* is used to indicate a change to the *Suppression* state of a *Condition* instance. It is formally defined in Table 93.

Table 93 – AuditConditionSuppressionEventType definition

Attribute			Value		
BrowseName			AuditConditionSuppressionEventType		
IsAbstract			False		
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

This *Event* indicates an *Alarm* suppression operation. An audit *Event* of this type shall be generated, if audit events are supported for any suppression action, including automatic system-based suppression.

5.10.10 AuditConditionSilenceEventType

This *EventType* is used to indicate a change to the *Silence* state of a *Condition* instance. It is formally defined in Table 94.

Table 94 – AuditConditionSilenceEventType definition

Attribute			Value		
BrowseName			AuditConditionSilenceEventType		
IsAbstract			False		
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

This event indicates that an *Alarm* was silenced, but not acknowledged. An audit event of this type shall be generated, if Audit events are supported for any silence action, including automatic system-based silence.

5.10.11 AuditConditionResetEventType

This *EventType* is used to indicate a change to the *Latched* state of a *Condition* instance. It is formally defined in Table 95.

Table 95 – AuditConditionResetEventType definition

Attribute		Value			
BrowseName		AuditConditionResetEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

This event indicates that an *Alarm* was reset. An audit event of this type shall be generated, if Audit events are supported for any *Alarm* action.

5.10.12 AuditConditionOutOfServiceEventType

This *EventType* is used to indicate a change to the *OutOfService State* of a *Condition* instance. It is formally defined in Table 96.

Table 96 – AuditConditionOutOfServiceEventType definition

Attribute		Value			
BrowseName		AuditConditionOutOfServiceEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

An audit *Event* of this type shall be generated if audit *Events* are supported.

5.11 Condition Refresh related Events

5.11.1 Overview

Following are subtypes of *SystemEventType~~s~~* that will be generated in response to a *Refresh Methods* call. They are illustrated in Figure 23.

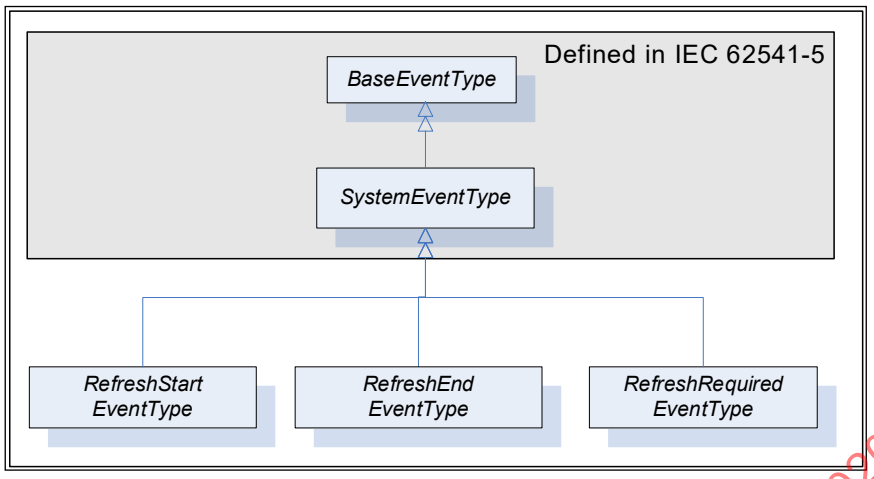


Figure 23 – Refresh Related Event Hierarchy

5.11.2 RefreshStartEventType

This *EventType* is used by a *Server* to mark the beginning of a *Refresh Notification* cycle. Its representation in the *AddressSpace* is formally defined in Table 97.

Table 97 – RefreshStartEventType definition

Attribute	Value				
BrowseName	RefreshStartEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *SystemEventType* defined in IEC 62541-5, i.e. it has HasProperty *References* to the same *Nodes*.					

5.11.3 RefreshEndEventType

This *EventType* is used by a *Server* to mark the end of a *Refresh Notification* cycle. Its representation in the *AddressSpace* is formally defined in Table 98.

Table 98 – RefreshEndEventType definition

Attribute	Value				
BrowseName	RefreshEndEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *SystemEventType* defined in IEC 62541-5, i.e. it has HasProperty *References* to the same *Nodes*.					

5.11.4 RefreshRequiredEventType

This *EventType* is used by a *Server* to indicate that a significant change has occurred in the *Server* or in the subsystem below the *Server* that may or does invalidate the *Condition* state of a *Subscription*. Its representation in the *AddressSpace* is formally defined in Table 99.

Table 99 – RefreshRequiredEventType definition

Attribute	**Value**				
BrowseName	RefreshRequiredEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *SystemEventType* defined in IEC 62541-5, i.e. it has HasProperty *References* to the same *Nodes*.					

When a *Server* detects an *Event* queue overflow, it shall track if any *Condition Event*s have been lost, if any *Condition Event*s were lost, it shall issue a *RefreshRequiredEventType Event* to the *Client* after the *Event* queue is no longer in an overflow state.

5.12 HasCondition Reference type

The *HasCondition ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *NonHierarchicalReferences*. The representation in the *AddressSpace* is specified in Table 100.

The semantic of this *ReferenceType* is to specify the relationship between a *ConditionSource* and its *Conditions*. Each *ConditionSource* shall be the target of a *HasEventSource Reference* or a subtype of *HasEventSource*. The *AddressSpace* organisation that shall be provided for *Clients* to detect *Conditions* and *ConditionSources* is defined in Clause 6. Various examples for the use of this *ReferenceType* ~~can~~ may be found in Clause B.2.

HasCondition References can be used in the *Type* definition of an *Object* or a *Variable*. In this case, the *SourceNode* of this *ReferenceType* shall be an *ObjectType* or *VariableType Node* or one of their *InstanceDeclaration Nodes*. The *TargetNode* shall be a *Condition* instance declaration or a *ConditionType*. The following rules for instantiation apply:

- all *HasCondition References* used in a *Type* shall exist in instances of these *Types* as well;
- if the *TargetNode* in the *Type* definition is a *ConditionType*, the same *TargetNode* will be referenced on the instance.

HasCondition References may be used solely in the instance space when they are not available in *Type* definitions. In this case, the *SourceNode* of this *ReferenceType* shall be an *Object*, *Variable* or *Method Node*. The *TargetNode* shall be a *Condition* instance or a *ConditionType*.

Table 100 – HasCondition *ReferenceType*

Attributes	**Value**		
BrowseName	HasCondition		
InverseName	IsConditionOf		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**

5.13 Alarm and Condition status codes

Table 101 defines the *StatusCodes* defined for *Alarm* and *Conditions* (A&C).

Table 101 – Alarm & Condition result codes

Symbolic Id	Description
Bad_ConditionAlreadyEnabled	The addressed Condition is already enabled.
Bad_ConditionAlreadyDisabled	The addressed Condition is already disabled.
Bad_ConditionAlreadyShelved	The Alarm is already in a shelved state.
Bad_ConditionBranchAlreadyAcked	The *EventId* does not refer to a state that needs acknowledgement.
Bad_ConditionBranchAlreadyConfirmed	The *EventId* does not refer to a state that needs confirmation.
Bad_ConditionNotShelved	The Alarm is not in the requested shelved state.
Bad_DialogNotActive	The *DialogConditionType* instance is not in *Active* state.
Bad_DialogResponseInvalid	The selected option is not a valid index in the *ResponseOptionSet* array.
Bad_EventIdUnknown	The specified *EventId* is not known to the *Server*.
Bad_RefreshInProgress	A ConditionRefresh operation is already in progress.
Bad_ShelvingTimeOutOfRange	The provided *Shelving* time is outside the range allowed by the *Server* for *Shelving*

5.14 Expected A&C server behaviours

5.14.1 General

This subclause describes behaviour that is expected from an OPC UA *Server* that is implementing the *A&C Information Model*. In particular this subclause describes specific behaviours that apply to various aspect of the A&C *Information Model*.

5.14.2 Communication problems

In some implementation of an OPC UA A&C *Server*, the *Alarms* and *Condition* are provided by an underlying system. The expected behaviour of an A&C *Server* when it is encountering communication problems with the underlying system is:

- If communication fails to the underlying system,
  - For any *Event* field related information that is exposed in the address space, the Value/*StatusCode* obtained when reading the *Event* fields that are associated with the communication failure shall have a value of NULL and a *StatusCode* of Bad_CommunicationError.
  - For *Subscriptions* that contain *Conditions* for which the failure applies, the effected *Conditions* generate an *Event,* if the *Retain* field is set to True. These *Events* shall have their *Event* fields that are associated with the communication failure contain a *StatusCode* of Bad_CommunicationError for the value.
  - A *Condition* of the SystemOffNormalAlarmType shall be used to report the communication failure to *Alarm Clients*. The *NormalState* field shall contain the *NodeId* of the *Variable* that indicates the status of the underlying system.
- For start-up of an A&C *Server* that is obtaining A&C information from an already running underlying system:
  - If a value is unavailable for an *Event* field that is being reported due to a start-up of the UA *Server* (i.e. the information is just not available for the *Event*) the *Event* field shall contain a *StatusCode* set to Bad_WaitingForInitialData for the value.
  - If the "Time" field is normally provided by the underlying system and is unavailable, the Time will be reported as a *StatusCode* with a value of Bad_WaitingForInitialData.

5.14.3 Redundant A&C servers

In an OPC UA *Server* that is implementing the A&C *Information Model* and that is configured to be a redundant OPC UA *Server*, the following behaviour is expected:

- The EventId is used to uniquely identify an *Event*. For an *Event* that is in each of the redundant *Servers*, it shall be identical. This applies to all standard *Events*, *Alarms* and *Conditions*. This may be accomplished by sharing of information between redundant *Server* (such as actual *Events*) or it may be accomplished by providing a strict *EventId* generating algorithm that will generate an identical *EventId* for each *Event*.
- It is expected that for cold or warm failovers of redundant *Servers*, *Subscription* for *Events* shall require a Refresh operation. The *Client* shall initiate this *Refresh* operation.
- It is expected that for hot failovers of redundant *Servers*, *Subscriptions* for *Events* may require a *Refresh* operation. The *Server* shall issue a *RefreshRequiredEventType Event* if it is required.
- For transparent redundancy, a *Server* shall not require any action be performed by a *Client*.

6 AddressSpace organisation

6.1 General

The *AddressSpace* organisation described in this clause allows *Clients* to detect *Conditions* and *ConditionSources*. An additional hierarchy of *Object Nodes* that are notifiers may be established to define one or more areas; the *Client* can subscribe to specific areas to limit the *Event Notifications* sent by the *Server*. Additional examples can be found in Clause B.2.

6.2 EventNotifier and source hierarchy

HasNotifier and *HasEventSource References* are used to expose the hierarchical organization of *Event* notifying *Objects* and *ConditionSources*. An *Event* notifying *Object* represents typically an area of *Operator* responsibility. The definition of such an area configuration is outside the scope of this document. If areas are available, they shall be linked together and with the included *ConditionSources* using the *HasNotifier* and the *HasEventSource Reference Types*. The *Server Object* shall be the root of this hierarchy.

Figure 24 shows such a hierarchy. Note that *HasNotifier* is a subtype of *HasEventSource*. I.e. the target *Node* of a *HasNotifier Reference* (an *Event* notifying *Object*) ~~may~~ can also be a *ConditionSource*. The *HasEventSource Reference* is used if the target *Node* is a *ConditionSource* but cannot be used as *Event* notifier. See IEC 62541-3 for the formal definition of these *Reference Types*.

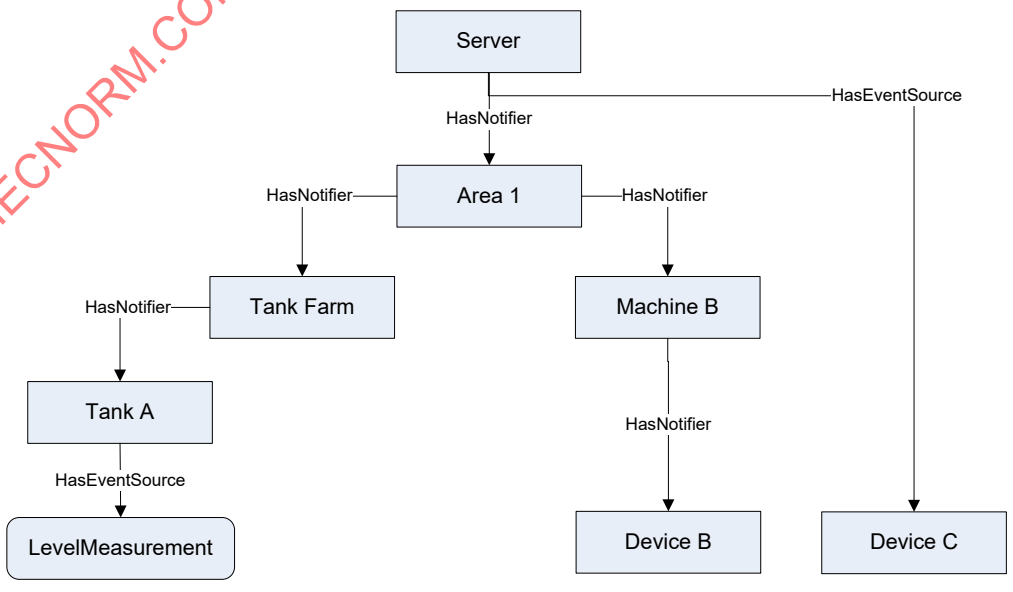


Figure 24 – Typical ~~Event~~ HasNotifier Hierarchy

6.3 Adding Conditions to the hierarchy

HasCondition is used to reference *Conditions*. The *Reference* is from a *ConditionSource* to a *Condition* instance or – if no instance is exposed by the *Server* – to the *ConditionType*.

Clients can locate *Conditions* by first browsing for *ConditionSources* following *HasEventSource References* (including subtypes like the *HasNotifier Reference*) and then browsing for *HasCondition References* from all target *Nodes* of the discovered *References*.

Figure 25 shows the application of the *HasCondition Reference* in ~~an *Event*~~ a *HasNotifier* hierarchy. The *Variable* LevelMeasurement and the *Object* "Device B" *Reference Condition* instances. The *Object* "Tank A" *References* a *ConditionType* (MySystemAlarmType) indicating that a *Condition* exists but is not exposed in the *AddressSpace*.

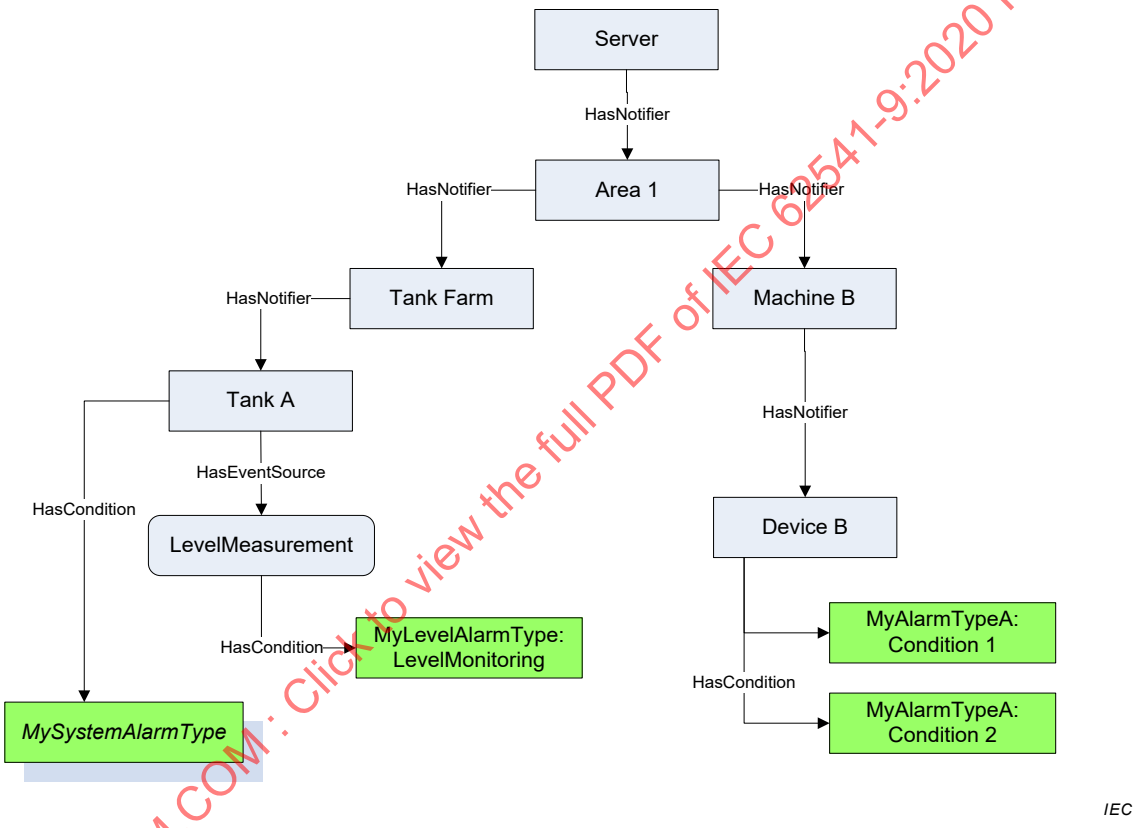


Figure 25 – Use of HasCondition in ~~an Event~~ a HasNotifier hierarchy

6.4 Conditions in InstanceDeclarations

Figure 26 shows the use of the *HasCondition Reference* and the *HasEventSource Reference* in an *InstanceDeclaration*. They are used to indicate what *References* and *Conditions* are available on the instance of the *ObjectType*.

The use of the *HasEventSource Reference* in the context of *InstanceDeclarations* and *TypeDefinition Nodes* has no effect for *Event* generation.

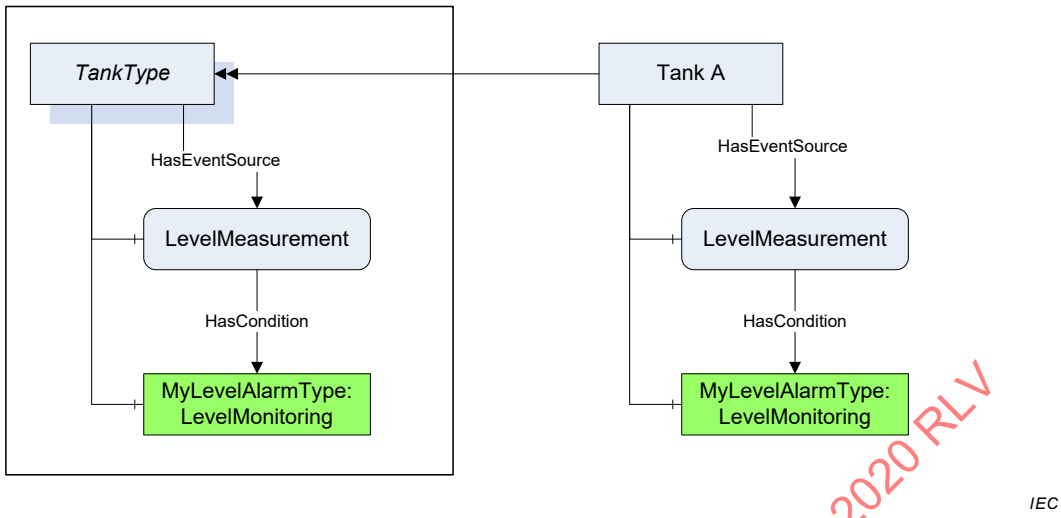


Figure 26 – Use of HasCondition in an InstanceDeclaration

6.5 Conditions in a VariableType

Use of *HasCondition* in a *VariableType* is a special use case since *Variables* (and *VariableTypes*) may not have *Conditions* as components. Figure 27 provides an example of this use case. Note that there is no component relationship for the "LevelMonitoring" *Alarm*. It is *Server*-specific whether and where they assign a *HasComponent Reference*.

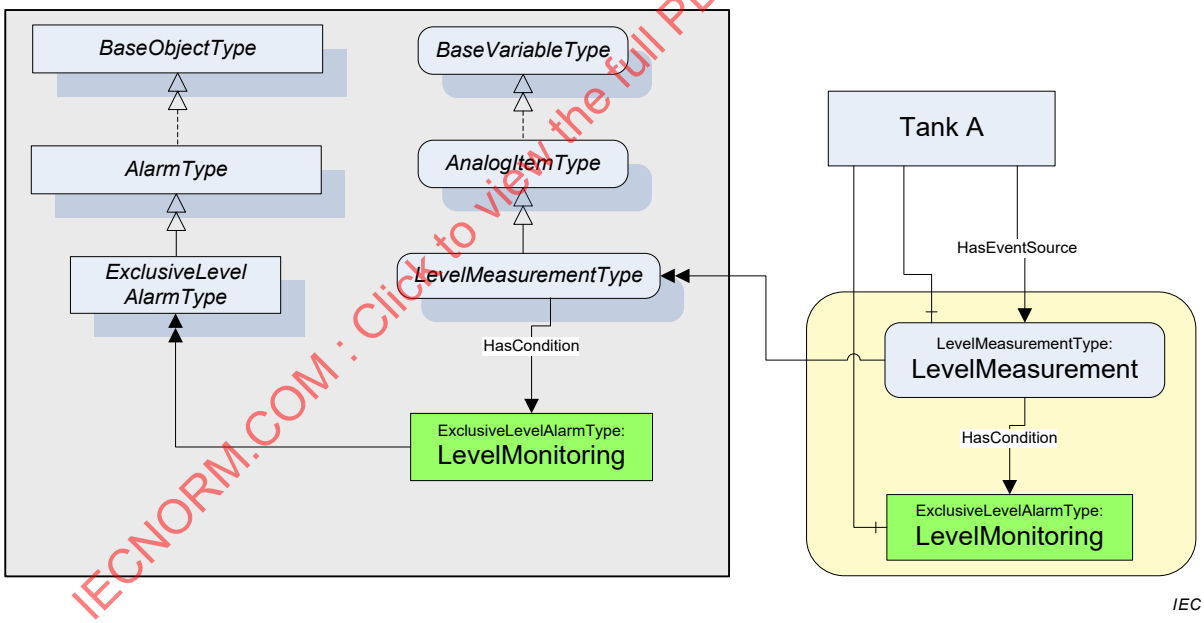


Figure 27 – Use of HasCondition in a VariableType

7 System State and alarms

7.1 Overview

The state of alarms is affected by the state of the process, equipment, system or plant. For example, when a tank is taken out of service, the level alarms associated with the tank would be no longer used, until the tank is returned to service. This clause describes *ReferenceTypes* that can be used by a *StateMachine* to indicate that a specific *Effect* on *Alarms* caused by the transition of a *StateMachine*. *StateMachines* that describe the state of a process, system or equipment can vary, but an example *StateMachine* is provided in Annex F.

7.2 HasEffectDisable

The *HasEffectDisable ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *HasEffect*.

The semantic of this *ReferenceType* is to point form a *Transition* to an *Alarm* that will be disabled.

- If the *Reference* is to an *Object* then all *Alarms* in the *HasNotifier* hierarchy below that *Object* are disabled,
- If the target is an *AlarmType* then all instances of that *AlarmType* in the *HasNotifier* hierarchy below the *Object* containing the *StateMachine* are disabled,
- If the target is an *Alarm* instance then the given *Alarm* instance is disabled.

The *SourceNode* of this *ReferenceType* shall be an *Object* of the *ObjectType TransitionType* or one of its subtypes. The *TargetNode* can be of an *Object* or *AlarmType*.

The representation of the *HasEffectDisable ReferenceType* in the *AddressSpace* is specified in Table 102.

Table 102 – HasEffectDisable ReferenceType

Attributes	Value		
BrowseName	*HasEffectDisable*		
InverseName	MayBeDisabledBy		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**

7.3 HasEffectEnable

The *HasEffectEnable ReferenceType* is a concrete *ReferenceType* and may be used directly. It is a subtype of *HasEffect*..

The semantic of this *ReferenceType* is to point form a *Transition* to an *Alarm* that will be enabled.

- If the *Reference* is to an *Object* then all *Alarms* in the *HasNotifier* hierarchy below that *Object* are enabled.
- If the target is an *AlarmType* then all instances of that *AlarmType* in the *HasNotifier* hierarchy below the *Object* containing the *StateMachine* are enabled.
- If the target is an *Alarm* instance then the given *Alarm* instance is enabled.

The *SourceNode* of this *ReferenceType* shall be an *Object* of the *ObjectType TransitionType* or one of its subtypes. The *TargetNode* can be of any *NodeClass*.

The representation of the *HasEffectenable ReferenceType* in the *AddressSpace* is specified in Table 103.

Table 103 – HasEffectEnable ReferenceType

Attributes	Value		
BrowseName	HasEffectEnable		
InverseName	MayBeEnabledBy		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**

7.4 HasEffectSuppress

The *HasEffectSuppress ReferenceType* is a concrete *ReferenceType* and may be used directly. It is a subtype of *HasEffect*.

The semantic of this *ReferenceType* is to point form a *Transition* to an *Alarm* that will be suppressed.

- If the reference is to an *Object* then all *Alarms* in the *EventNotifer* hierarchy below that *Object* are suppressed.
- If the target is an *AlarmType* then all instance of that AlarmType in the *HasNotifier* hierarchy below the *Object* containing the *StateMachine* are suppressed.
- If the target is an *Alarm* instance then the given *Alarm* instance is suppressed.

The *SourceNode* of this *ReferenceType* shall be an *Object* of the *ObjectType TransitionType* or one of its subtypes. The *TargetNode* can be of any *NodeClass*.

The representation of the *HasEffectSuppress ReferenceType* in the *AddressSpace* is specified in Table 104.

Table 104 – HasEffectSuppress ReferenceType

Attributes	Value		
BrowseName	HasEffectSuppress		
InverseName	MayBeSuppressedBy		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**

7.5 HasEffectUnsuppressed

The *HasEffectUnsuppressed ReferenceType* is a concrete *ReferenceType* and may be used directly. It is a subtype of *HasEffect*.

The semantic of this *ReferenceType* is to point form a *Transition* to an *Alarm* that will no longer be suppressed.

- If the *Reference* is to an *Object* then all *Alarms* in the *HasNotifier* hierarchy below that *Object* are removed from being suppressed.
- If the target is an *AlarmType* then all instance of that AlarmType are no longer suppressed below the *Object* containing the *StateMachine*.

- if the target is an *Alarm* instance then the given *Alarm* instance is no longer suppressed. No errors are logged if the *Alarm* was not suppressed.

The *SourceNode* of this *ReferenceType* shall be an *Object* of the *ObjectType TransitionType* or one of its subtypes. The *TargetNode* can be of any *NodeClass*.

The representation of the *HasEffectUnsuppress ReferenceType* in the *AddressSpace* is specified in Table 105.

Table 105 – HasEffectUnsuppress ReferenceType

Attributes	Value		
BrowseName	HasEffectUnsuppress		
InverseName	MayBeUnsuppressedBy		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**

8 Alarm metrics

8.1 Overview

The goal of a well-designed alarm system is to ensure that an *Operator* is made aware of issues, both critical and non-critical, but is not overwhelmed by alarms/alerts or other messages. When designing an alarm system, criteria are defined for alarm rates and general performance of the system at various levels (*Operator* station, plant area, overall system etc.). Evaluating the performance of an alarm system with regard to these design criteria requires the collection of alarm metrics. These metrics provide summaries of alarm rates and other alarm-related information.

This clause defines a standard structure for metrics. This structure may be implemented at multiple levels allowing a *Server* to collect metrics as needed. For example, an *Object* of this type might be added to the *Server Object* providing a summary of the *Alarm* performance for the entire *Server*. An instance might also be provided on an *Object* that includes a *HasNotifier* hierarchy, such as a tank *Object*. In this case, it would provide the summary of all of the *Alarms* that are part of the tank *HasNotifier* hierarchy.

8.2 AlarmMetricsType

This Object*Type* is used for metric information. The *ObjectType* is formally defined in Table 106.

Table 106 – AlarmMetricsType Definition

Attribute		Value			
BrowseName		AlarmMetricsType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the BaseObjectType defined in IEC 62541-5.					
HasComponent	Variable	AlarmCount	UInt32	BaseDataVariableType	Mandatory
HasComponent	Variable	StartTime	UtcTime	BaseDataVariableType	Mandatory
HasComponent	Variable	MaximumActiveState	Duration	BaseDataVariableType	Mandatory
HasComponent	Variable	MaximumUnAck	Duration	BaseDataVariableType	Mandatory
HasComponent	Variable	CurrentAlarmRate	Double	AlarmRateVariableType	Mandatory
HasComponent	Variable	MaximumAlarmRate	Double	AlarmRateVariableType	Mandatory
HasComponent	Variable	MaximumReAlarmCount	UInt32	BaseDataVariableType	Mandatory
HasComponent	Variable	AverageAlarmRate	Double	AlarmRateVariableType	Mandatory
HasComponent	Method	Reset			Mandatory

An instance of *AlarmMetricsType* can be added, with a *HasComponent* reference, to any *Object* that has its "SubscribeToEvents" bit set within the *EventNotifier Attribute*. It will collect the *Alarm* metrics for all *Alarm* sources assigned to this notifier *Object*. For example, if *Alarm* metrics are desired for Tank A *Object* (see Figure B.3) that is in the *HasNotifier* hierarchy than an instance of this object would be referenced by the Tank A object. When this object is associated with the *Server Object* it will report *Alarm* metrics for the entire *Server*.

AlarmCount is the total count of *Alarm*s since the last restart of the system or reset of this counter.

StartTime is the time at which the *Server* started or the time of the last *Reset* Method invocation, whichever is later.

MaximumActiveState is the maximum time for which an *Alarm* was in the active state.

MaximumUnAck is the maximum time for which an *Alarm* was in the unacknowledged state.

CurrentAlarmRate is the sum of *Alarms* that occurred in the last *Rate* number of minutes (see 8.3). This sum should not include nuisance Alarms (i.e. chattering alarms). It is updated every *Rate* number of minutes.

MaximumAlarmRate is the maximum *Alarm* rate detected since the start of the *Server*, where the rate is calculated as for *CurrentAlarmRate*.

MaximumReAlarmCount is the maximum *ReAlarmCount* for any *Alarm*.

AverageAlarmRate is the average *Alarm* rate since the start of the *Server* or the last invocation of *Reset Method*, where the rate is calculated as for *CurrentAlarmRate*.

Reset is a *Method* that will reset all of the counters, rates or times in this *Object*

8.3 AlarmRateVariableType

This variable type provides a unit field for the rate for which the *Alarm* diagnostic applies.

Table 107 – AlarmRateVariableType definition

Attribute		Value			
BrowseName		AlarmRateVariableType			
IsAbstract		False			
ValueRank		Scalar			
DataType		Double			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	Rate	UInt16	PropertyType	Mandatory

Rate is the number of minutes over which the item is calculated.

8.4 Reset Method

The *Reset Method* is used to reset all of the counters, rates and time in the *Object.*

Signature

```
Reset();
```

Method Result Codes in Table 108 (defined in Call Service).

Table 108 – Suppress result codes

Result Code	Description
Bad_MethodInvalid	The MethodId provided does not correspond to the ObjectId provided. See IEC 62541-4 for the general description of this result code.
Bad_NodeIdInvalid	Used to indicate that the specified *ObjectId* is not valid. See IEC 62541-4 for the general description of this result code.

Comments

The Reset Method will clear all setting in the diagnostic object and initialize them to zero.

Table 109 specifies the *AddressSpace* representation for the *Reset Method*.

Table 109 – Reset Method AddressSpace definition

Attribute	Value				
BrowseName	Reset				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditUpdateMethodEventType	Defined in IEC 62541-5.		

Annex A
(informative)

Recommended localized names

A.1 Recommended state names for TwoState variables

A.1.1 LocaleId "en"

The recommended state display names for the LocaleId "en" are listed in Table A.1 and Table A.2.

Table A.1 – Recommended state names for LocaleId "en"

Condition Type	State Variable	False State Name	True State Name
ConditionType	EnabledState	Disabled	Enabled
DialogConditionType	DialogState	Inactive	Active
AcknowledgeableConditionType	AckedState	Unacknowledged	Acknowledged
	ConfirmedState	Unconfirmed	Confirmed
AlarmConditionType	ActiveState	Inactive	Active
	SuppressedState	Unsuppressed	Suppressed
	OutOfServiceState	In Service	Out of Service
	SilenceState	Silenced	Not Silenced
	LatchedState	Latched	Unlatched
NonExclusiveLimitAlarmType	HighHighState	HighHigh inactive	HighHigh active
	HighState	High inactive	High active
	LowState	Low inactive	Low active
	LowLowState	LowLow inactive	LowLow active

Table A.2 – Recommended display names for LocaleId "en"

Condition Type	Browse Name	display name
Shelved	Unshelved	Unshelved
	TimedShelved	Timed Shelved
	OneShotShelved	One Shot Shelved
Exclusive	HighHigh	HighHigh
	High	High
	Low	Low
	LowLow	LowLow

A.1.2 LocaleId "de"

The recommended state display names for the LocaleId "de" are listed in Table A.3 and Table A.4.

Table A.3 – Recommended state names for LocaleId "de"

Condition Type	State Variable	False State Name	True State Name
ConditionType	EnabledState	Ausgeschaltet	Eingeschaltet
DialogConditionType	DialogState	Inaktiv	Aktiv
AcknowledgeableConditionType	AckedState	Unquittiert	Quittiert
	ConfirmedState	Unbestätigt	Bestätigt
AlarmConditionType	ActiveState	Inaktiv	Aktiv
	SuppressedState	Nicht unterdrückt	Unterdrückt
	OutOfServiceState	In Betrieb	Außer Betrieb
	SilenceState	Stumm	Nicht Stumm
	LatchedState	Verriegelt	Entriegelt
NonExclusiveLimitAlarmType	HighHighState	HighHigh inaktiv	HighHigh aktiv
	HighState	High inaktiv	High aktiv
	LowState	Low inaktiv	Low aktiv
	LowLowState	LowLow inaktiv	LowLow aktiv

Table A.4 – Recommended display names for LocaleId "de"

Condition Type	Browse Name	display name
Shelved	Unshelved	Nicht zurückgestellt
	TimedShelved	Befristet zurückgestellt
	OneShotShelved	Einmalig zurückgestellt
Exclusive	HighHigh	HighHigh
	High	High
	Low	Low
	LowLow	LowLow

A.1.3 LocaleId "fr"

The recommended state display names for the LocaleId "fr" are listed in Table A.5 and Table A.6.

Table A.5 – Recommended state names for LocaleId "fr"

Condition Type	State Variable	False State Name	True State Name
ConditionType	EnabledState	Hors Service	En Service
DialogConditionType	DialogState	Inactive	Active
AcknowledgeableConditionType	AckedState	Non-acquitté	Acquitté
	ConfirmedState	Non-Confirmé	Confirmé
AlarmConditionType	ActiveState	Inactive	Active
	SuppressedState	Présent	Supprimé
	OutOfServiceState	En Fonction	Hors Fonction
	SilenceState	Muette	Non-Muette
	LatchedState		
NonExclusiveLimitAlarmType	HighHighState	Très Haute Inactive	Très Haute Active
	HighState	Haute inactive	Haute active
	LowState	Basse inactive	Basse active
	LowLowState	Très basse inactive	Très basse active

Table A.6 – Recommended display names for LocaleId "fr"

Condition Type	Browse Name	display name
Shelved	Unshelved	Surveillée
	TimedShelved	Mise de coté temporelle
	OneShotShelved	Mise de coté unique
Exclusive	HighHigh	Très haute
	High	Haute
	Low	Basse
	LowLow	Très basse

A.2 Recommended dialog response options

The recommended *Dialog* response option names in different locales are listed in Table A.7.

Table A.7 – Recommended dialog response options

Locale "en"	Locale "de"	Locale "fr"
Ok	OK	Ok
Cancel	Abbrechen	Annuler
Yes	Ja	Oui
No	Nein	Non
Abort	Abbrechen	Abandonner
Retry	Wiederholen	Réessayer
Ignore	Ignorieren	Ignorer
Next	Nächster	Prochain
Previous	Vorheriger	Precedent

Annex B
(informative)

Examples

B.1 Examples for Event sequences from Condition instances

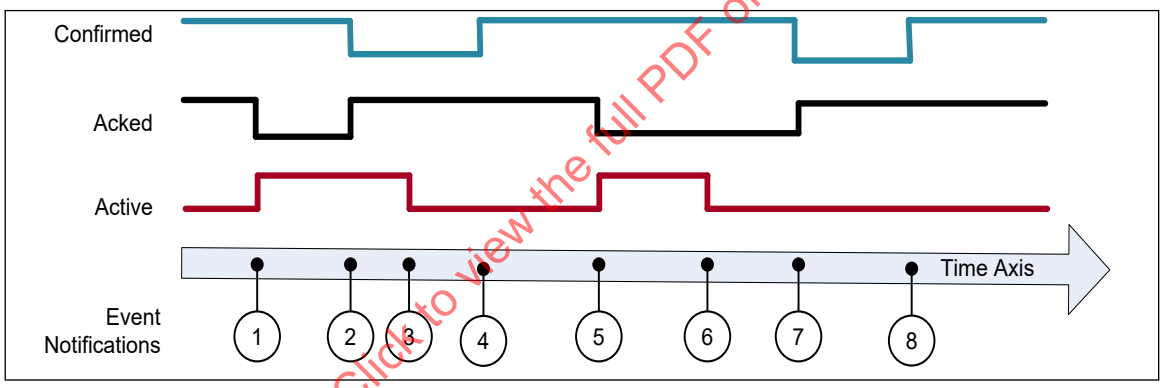
B.1.1 Overview

The following examples show the *Event* flow for typical *Alarm* situations. Table B.1 and Table B.2 list the value of state *Variables* for each *Event Notification*.

B.1.2 Server maintains current state only

This example is for *Servers* that do not support previous states and therefore do not create and maintain *Branches* of a single *Condition*.

Figure B.1 shows an *Alarm* as it becomes active and then inactive and also the acknowledgement and confirmation cycles. Table B.1 lists the values of the state *Variables*. All *Events* are coming from the same *Condition* instance and therefore have the same *ConditionId*.



IEC

Figure B.1 – Single state example

Table B.1 – Example of a Condition that only keeps the latest state

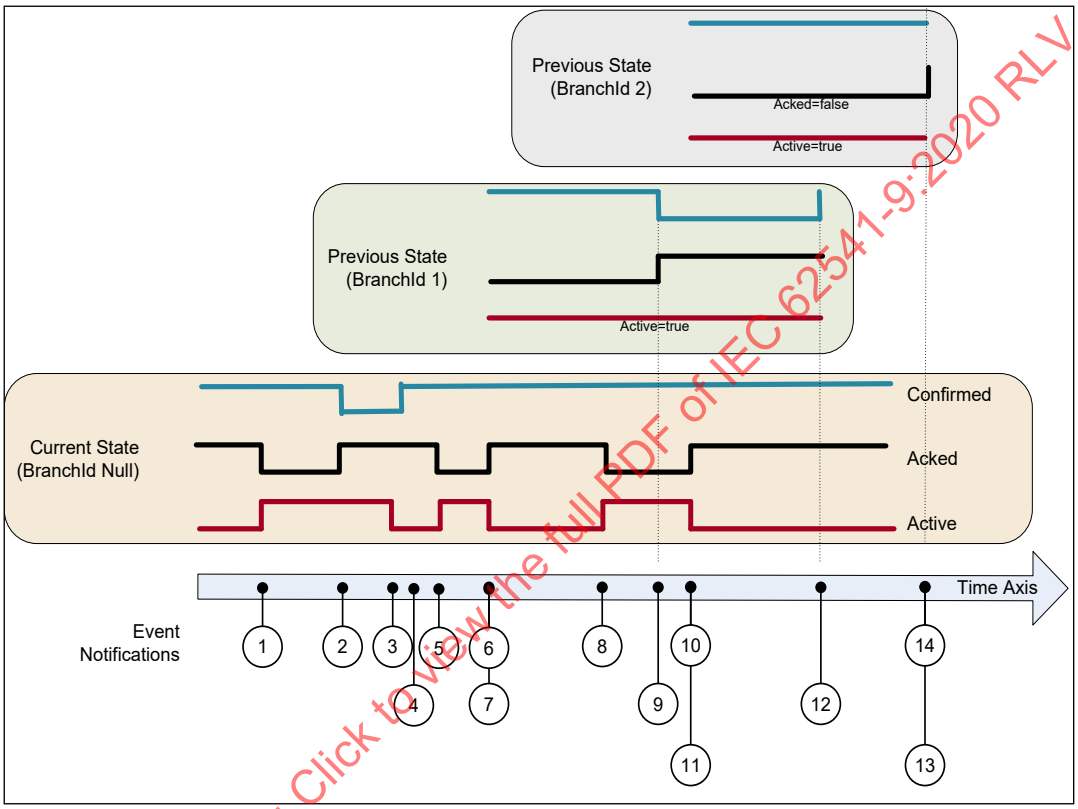
EventId	BranchId	Active	Acked	Confirmed	Retain	Description
-*)	NULL	False	True	True	False	Initial state of *Condition*.
1	NULL	True	False	True	True	*Alarm* goes active.
2	NULL	True	True	False	True	*Condition* acknowledged Confirm required
3	NULL	False	True	False	True	*Alarm* goes inactive.
4	NULL	False	True	True	False	*Condition* confirmed
5	NULL	True	False	True	True	*Alarm* goes active.
6	NULL	False	False	True	True	*Alarm* goes inactive.
7	NULL	False	True	False	True	*Condition* acknowledged, Confirm required.
8	NULL	False	True	True	False	*Condition* confirmed.

*) The first row is included to illustrate the initial state of the *Condition*. This state will not be reported by an *Event*.

B.1.3 Server maintains previous states

This example is for *Servers* that are able to maintain previous states of a *Condition* and therefore create and maintain *Branches* of a single *Condition*.

Figure B.2 illustrates the use of branches by a *Server* requiring acknowledgement of all transitions into *Active* state, not just the most recent transition. In this example no acknowledgement is required on a transition into an inactive state. Table B.2 lists the values of the state *Variables*. All *Events* are coming from the same *Condition* instance and have therefore the same *ConditionId*.



IEC

Figure B.2 – Previous state example

Table B.2 – Example of a *Condition* that maintains previous states via branches

EventId	BranchId	Active	Acked	Confirmed	Retain	Description
[a]	NULL	False	True	True	False	Initial state of *Condition*.
1	NULL	True	False	True	True	Alarm goes active.
2	NULL	True	True	True	True	Condition acknowledged requires Confirm
3	NULL	False	True	False	True	*Alarm* goes inactive.
4	NULL	False	True	True	False	Confirmed
5	NULL	True	False	True	True	*Alarm* goes active.
6	NULL	False	True	True	True	*Alarm* goes inactive.
7	1	True	False	True	True [b]	Prior state needs acknowledgment. Branch #1 created.
8	NULL	True	False	True	True	*Alarm* goes active again.
9	1	True	True	False	True	Prior state acknowledged. Confirm required.
10	NULL	False	True	True	True [b]	*Alarm* goes inactive again.
11	2	True	False	True	True	Prior state needs acknowledgment. Branch #2 created.
12	1	True	True	True	False	Prior state confirmed. Branch #1 deleted.
13	2	True	True	True	False	Prior state acknowledged, Auto Confirmed by system. Branch #2 deleted. The confirmation of the previous transition allows the system to auto confirm this transition
14	NULL	False	True	True	False	No longer of interest.

[a] *The first row* is included to illustrate the initial state of the *Condition*. This state will not be reported by an *Event*.

Notes on specific situations shown with this example:

If the current state of the *Condition* is acknowledged then the *Acked* flag is set and the new state is reported (*Event* #2). If the *Condition* state changes before it can be acknowledged (*Event* #6) then a branch state is reported (*Event* #7). Timestamps for the *Event*s #6 and #7 is identical.

The branch state can be updated several times (*Event*s #9) before it is cleared (*Event* #12).

A single *Condition* can have many branch states active (*Event*s #11).

[b] *It is recommended* as in this table to leave Retain=True as long as there exist previous states (branches).

B.2 AddressSpace examples

This clause provides additional examples for the use of *HasNotifier*, *HasEventSource* and *HasCondition References* to expose the organization of areas and sources with their associated *Conditions*. This hierarchy is additional to a hierarchy provided with *Organizes* and *Aggregates References*.

Figure B.3 illustrates the use of the *HasCondition Reference* with *Condition* instances.

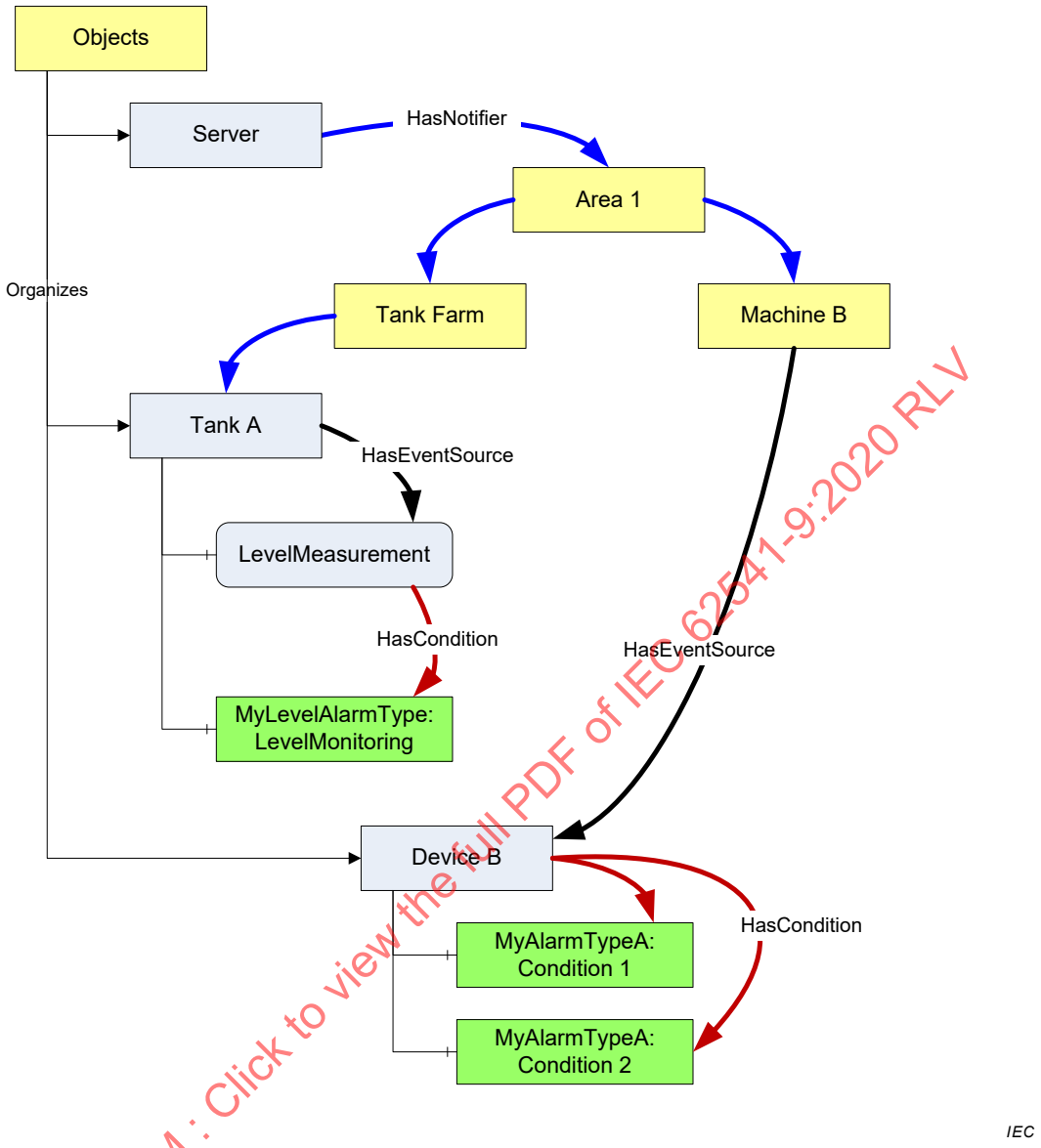


Figure B.3 – HasCondition used with Condition instances

In systems where *Conditions* are not available as instances, the *ConditionSource* ~~can~~ may reference the *ConditionTypes* instead. This is illustrated with the example in Figure B.4.

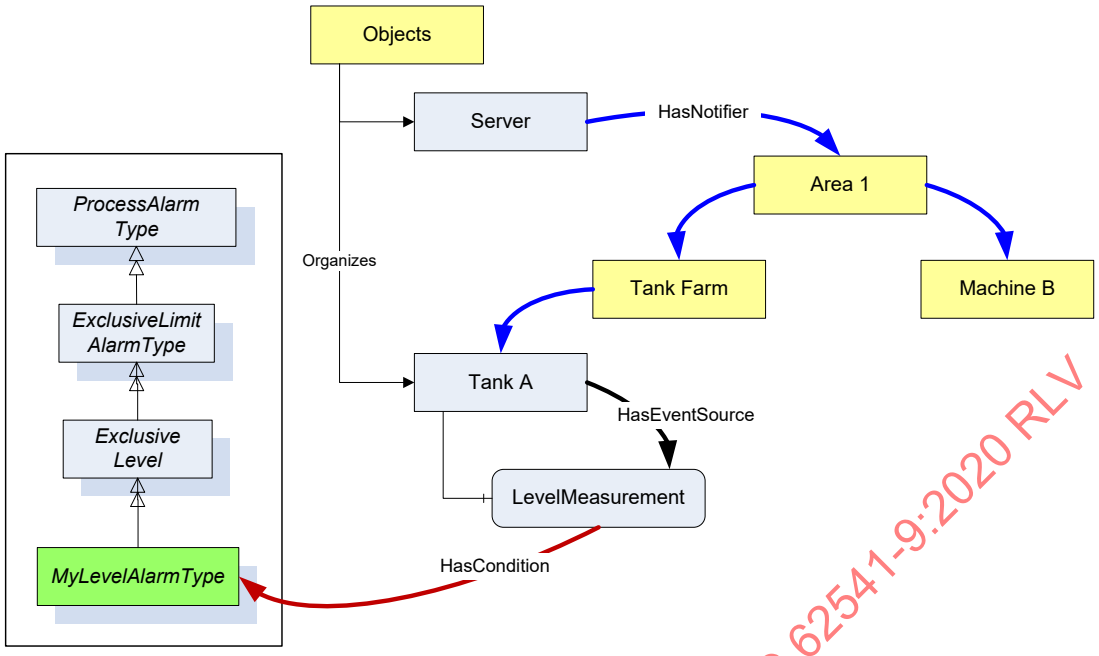


Figure B.4 – HasCondition reference to a Condition type

Figure B.5 provides an example where the *HasCondition Reference* is already defined in the *Type* system. The *Reference* ~~can~~ may point to a *Condition Type* or to an instance. Both variants are shown in this example. A *Reference* to a *Condition Type* in the *Type* system will result in a *Reference* to the same *Type Node* in the instance.

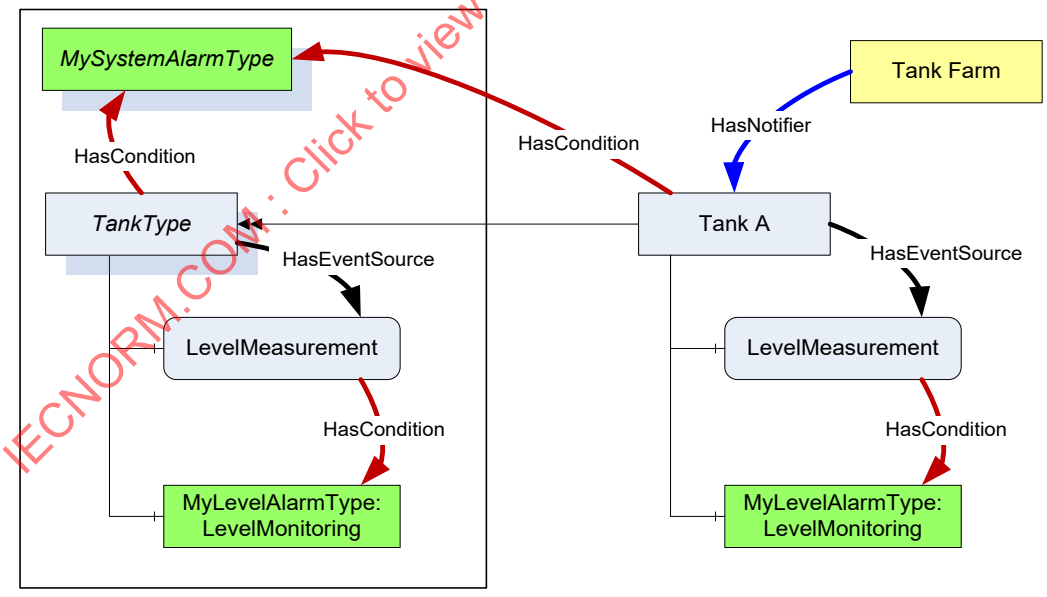


Figure B.5 – HasCondition used with an instance declaration

Annex C
(informative)

Mapping to EEMUA

Table C.1 lists EEMUA terms and how OPC UA terms maps to them.

Table C.1 – EEMUA Terms

EEMUA Term	OPC UA Term	EEMUA Definition
Accepted	Acknowledged=True	An *Alarm* is accepted when the *Operator* has indicated awareness of its presence. In OPC UA, this ~~can~~ may be accomplished with the *Acknowledge Method*.
Active Alarm	Active = True	An *Alarm Condition* which is on (i.e. limit has been exceeded and *Condition* continues to exist).
Alarm Message	Message Property (defined in IEC 62541-5.)	Test information presented to the *Operator* that describes the *Alarm Condition*.
Alarm Priority	Severity Property (defined in IEC 62541-5.)	The ranking of *Alarms* by severity and response time.
Alert	-	A lower priority *Notification* than an *Alarm* that has no serious consequence if ignored or missed. In some Industries also referred to as a "Prompt" or "Warning". No direct mapping! In UA the concept of *Alerts* ~~can~~ may be accomplished by the use of severity. E.g., *Alarms* that have a severity below 50 may be considered as *Alerts*.
Cleared	Active = False	An *Alarm* state that indicates the *Condition* has returned to normal.
Disable	Enabled = False	An *Alarm* is disabled when the system is configured such that the *Alarm* will not be generated even though the base *Alarm Condition* is present.
Prompt	Dialog	A request from the control system that the *Operator* perform some process action that the system cannot perform or that requires *Operator* authority to perform.
Raised	Active = True	An *Alarm* is *Raised* or initiated when the *Condition* creating the *Alarm* has occurred.
Release	OneShotShelving	A "release" is a facility that ~~can~~ may be applied to a standing (UA = active) *Alarm* in a similar way to which *Shelving* is applied. A released *Alarm* is temporarily removed from the *Alarm* list and put on the shelf. There is no indication to the *Operator* when the *Alarm* clears, but it is taken off the shelf. Hence, when the *Alarm* is raised again it appears on the *Alarm* list in the normal way.
Reset	Retain=False	An *Alarm* is Reset when it is in a state that can be removed from the Display list. OPC UA includes *Retain* flag which as part of its definition states: "when a *Client* receives an *Event* with the *Retain* flag set to False, the *Client* should consider this as a *Condition*/Branch that is no longer of interest, in the case of a "current *Alarm* display" the *Condition*/Branch would be removed from the display"
Shelving	Shelving	*Shelving* is a 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. A Shelved *Alarm* will be removed from the list and will not re-annunciate until un-shelved.
Standing	Active = True	An *Alarm* is *Standing* whilst the *Condition* persists (*Raised* and *Standing* are often used interchangeably).
Suppress	Suppress	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.
Unaccepted	Acknowledged = False	An *Alarm* is accepted when the *Operator* has indicated awareness of its presence. It is unaccepted until this has been done.

Annex D
(informative)

Mapping from OPC A&E to OPC UA A&C

D.1 Overview

Serving as a bridge between COM and OPC UA components, the Alarm and *Event*s proxy and wrapper enable existing A&E COM *Clients* and *Servers* to connect to UA *Alarms* and *Conditions* components.

Simply stated, there are two aspects to the migration strategy. The first aspect enables a UA *Alarms* and *Conditions Client* to connect to an existing Alarms and *Event*s COM *Server* via a UA *Server* wrapper. This wrapper is notated from this point forward as the A&E COM UA Wrapper. The second aspect enables an existing Alarms and *Event*s COM *Client* to connect to a UA *Alarms* and *Conditions Server* via a COM proxy. This proxy is notated from this point forward as the A&E COM UA Proxy.

An Alarms and *Event*s COM *Client* is notated from this point forward as A&E COM *Client*.

A UA *Alarms* and *Conditions Server* is notated from this point forward as UA A&C *Server*.

The mappings describe generic A&E COM interoperability components. It is recommended that vendors use this mapping if they develop their own components; however, some applications may benefit from vendor-specific mappings.

D.2 Alarms and Events COM UA wrapper

D.2.1 Event Areas

Event Areas in the A&E COM *Server* are represented in the A&E COM UA Wrapper as *Objects* with a *TypeDefinition* of *BaseObjectType*. The EventNotifier *Attribute* for these *Objects* always has the SubscribeToEvents flag set to True.

The root Area is represented by an *Object* with a *BrowseName* that depends on the UA *Server*. It is always the target of a HasNotifier *Reference* from the *Server Node*. The root Area allows multiple A&E COM *Server*s to be wrapped within a single UA *Server*.

The Area hierarchy is discovered with the BrowseOPCAreas and the GetQualifiedAreaName *Method*s. The Area name returned by BrowseOPCAreas is used as the *BrowseName* and *DisplayName* for each Area *Node*. The QualifiedAreaName is used to construct the *NodeId*. The NamespaceURI qualifying the *NodeId* and *BrowseName* is a unique URI assigned to the combination of machine and COM *Server*.

Each Area is the target of *HasNotifier Reference* from its parent Area. It may be the source of one or more HasNotifier *Reference*s to its child Areas. It may also be a source of a *HasEventSource Reference* to any sources in the Area.

The A&E COM *Server* may not support filtering by Areas. If this is the case, then no Area *Node*s are shown in the UA *Server* address space. Some implementations could use the AREAS *Attribute* to provide filtering by Areas within the A&E COM UA Wrapper.

D.2.2 Event sources

Event Sources in the A&E COM *Server* are represented in the A&E COM UA Wrapper as *Objects* with a *TypeDefinition* of *BaseObjectType*. If the A&E COM *Server* supports source filtering then the SubscribeToEvents flag is True and the Source is a target of a HasNotifier *Reference*. If source filtering is not supported the SubscribeToEvents flag is False and the Source is a target of a HasEventSource *Reference*.

The Sources are discovered by calling BrowseOPCAreas and the GetQualifiedSourceName *Method*s. The Source name returned by BrowseOPCAreas is used as the BrowseName and DisplayName. The QualifiedSourceName is used to construct the *NodeId*. *Event* Source *Node*s are always targets of a HasEventSource *Reference* from an Area.

D.2.3 Event categories

Event Categories in the A&E COM *Server* are represented in the UA *Server* as *ObjectTypes* which are subtypes of *BaseEventType*. The *BrowseName* and *DisplayName* of the *ObjectType Node* for Simple and Tracking *Event* Types are constructed by appending the text 'EventType' to the Description of the *Event* Category. For *Condition Event* Types the text 'AlarmType' is appended to the *Condition* Name.

These *ObjectType Node*s have a super type which depends on the A&E *Event* Type, the *Event* Category Description and the *Condition* Name; however, the best mapping requires knowledge of the semantics associated with the *Event* Categories and *Condition* Names. If an A&E COM UA Wrapper does not know these semantics then Simple *Event* Types are subtypes of *BaseEventType*, Tracking *Event* Types are subtypes of *AuditEventType* and *Condition Event* Types are subtypes of the *AlarmType*. Table D.1 defines mappings for a set of "well known" Category description and *Condition* Names to a standard super type.

Table D.1 – Mapping from standard Event categories to OPC UA Event types

COM A&E Event Type	Category Description	Condition Name	OPC UA EventType
Simple	---	---	BaseEventType
Simple	Device Failure	---	DeviceFailureEventType
Simple	System Message	---	SystemEventType
Tracking	---	---	AuditEventType
Condition	---	---	AlarmType
Condition	Level	---	LimitAlarmType
Condition	Level	PVLEVEL	ExclusiveLevelAlarmType
Condition	Level	SPLEVEL	ExclusiveLevelAlarmType
Condition	Level	HI HI	NonExclusiveLevelAlarmType
Condition	Level	HI	NonExclusiveLevelAlarmType
Condition	Level	LO	NonExclusiveLevelAlarmType
Condition	Level	LO LO	NonExclusiveLevelAlarmType
Condition	Deviation	---	NonExclusiveDeviationAlarmType
Condition	Discrete	---	DiscreteAlarmType
Condition	Discrete	CFN	OffNormalAlarmType
Condition	Discrete	TRIP	TripAlarmType

There is no generic mapping defined for A&E COM sub-*Conditions*. If an *Event* Category is mapped to a *LimitAlarmType* then the sub *Condition* name in the *Event* ~~are~~ shall be used to set the state of a suitable State *Variable*. For example, if the sub-*Condition* name is "HI HI" then that means the *HighHigh* state for the *LimitAlarmType* is active

For *Condition Event* Types, the *Event* Category is also used to define subtypes of *BaseConditionClassType*.

Figure D.1 illustrates how *ObjectType Nodes* created from the *Event* Categories and *Condition* Names are placed in the standard OPC UA ~~*Event*~~ *HasNotifier* hierarchy.

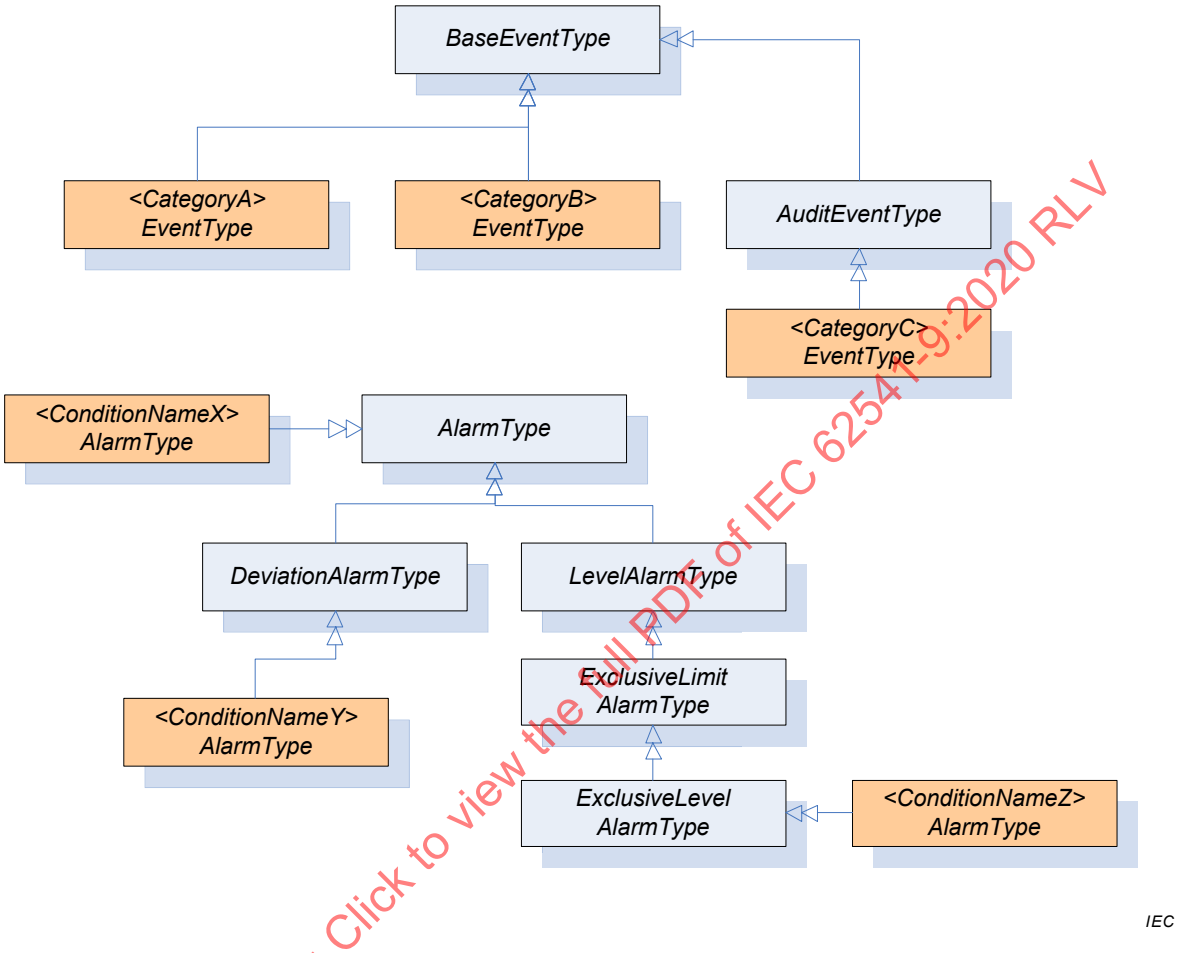


Figure D.1 – The type model of a wrapped COM A&E server

D.2.4 Event attributes

Event Attributes in the A&E COM *Server* are represented in the UA *Server* as *Variables* which are targets of *HasProperty References* from the *ObjectTypes* which represent the *Event* Categories. The *BrowseName* and *DisplayName* are the description for the *Event Attribute*. The data type of the *Event Attribute* is used to set *DataType* and *ValueRank*. The *NodeId* is constructed from the EventCategoryId, ConditionName and the AttributeId.

D.2.5 Event subscriptions

The A&E COM UA Wrapper creates a *Subscription* with the COM AE *Server* the first time a *MonitoredItem* is created for the *Server Object* or one of the *Nodes* representing Areas. The Area filter is set based on the *Node* being monitored. No other filters are specified.

If all *MonitoredItems* for an Area are disabled then the *Subscription* will be deactivated.

The *Subscription* is deleted when the last *MonitoredItem* for the *Node* is deleted.

When filtering by Area, the A&E COM UA Wrapper needs to add two Area filters: one based on the QualifiedAreaName which forms the *NodeId* and one with the text '/*' appended to it. This ensures that *Events* from sub areas are correctly reported by the COM AE *Server*.

A simple A&E COM UA Wrapper will always request all *Attributes* for all *Event* Categories when creating the *Subscription*. A more sophisticated wrapper may look at the *EventFilter* to determine which *Attributes* are actually used and only request those.

Table D.2 lists how the fields in the ONEVENTSTRUCT which are used by the A&E COM UA Wrapper are mapped to UA BaseEventType *Variable*s.

Table D.2 – Mapping from ONEVENTSTRUCT fields to UA BaseEventType Variables

UA Event Variable	ONEVENTSTRUCT Field	Notes
EventId	szSource szConditionName ftTime ftActiveTime dwCookie	A ByteString constructed by appending the fields together.
EventType	dwEventType dwEventCategory szConditionName	The NodeId for the corresponding *ObjectType Node*. The szConditionName maybe omitted by some implementations.
SourceNode	szSource	The *NodeId* of the corresponding Source *Object Node*.
SourceName	szSource	-
Time	ftTime	-
ReceiveTime	-	Set when the *Notification* is received by the wrapper.
LocalTime	-	Set based on the clock of the machine running the wrapper.
Message	szMessage	"Locale" is the default locale for the COM AE *Server*.
Severity	dwSeverity	-

Table D.3 lists how the fields in the ONEVENTSTRUCT which are used by the A&E COM UA Wrapper are mapped to UA *AuditEventType Variable*s.

Table D.3 – Mapping from ONEVENTSTRUCT fields to UA AuditEventType Variables

UA Event Variable	ONEVENTSTRUCT Field	Notes
ActionTimeStamp	ftTime	Only set for tracking *Event*s.
Status	-	Always set to True.
ServerId	-	Set to the COM AE *Server* NamespaceURI
ClientAuditEntryId	-	Not set.
ClientUserId	szActorID	-

Table D.4 lists how the fields in the ONEVENTSTRUCT which are used by the A&E COM UA Wrapper are mapped to UA AlarmType Variables.

Table D.4 – Mapping from ONEVENTSTRUCT fields to UA AlarmType Variables

UA Event Variable	ONEVENTSTRUCT Field	Notes
ConditionClassId	dwEventType	Set to the *NodeId* of the *ConditionClassType* for the *Event* Category of a *Condition Event* Type. Set to the *NodeId* of *BaseConditionClassType Node* for non-*Condition Event* Types.
ConditionClassName	dwEventType	Set to the *BrowseName* of the *ConditionClassType* for the *Event* Category of Condition *Event* Type. To set "BaseConditionClass" non-*Condition Event* Types.
ConditionName	szConditionName	-
BranchId	-	Always set to NULL.
Retain	wNewState	Set to True if the OPC_CONDITION_ACKED bit is not set or OPC_CONDITION_ACTIVE bit is set.
EnabledState	wNewState	Set to "Enabled" or "Disabled"
EnabledState.Id	wNewState	Set to True if OPC_CONDITION_ENABLED is set
EnabledState. EffectiveDisplayName	wNewState	A string constructed from the bits in the wNewState flag. The following rules are applied in order to select the string: "Disabled" if OPC_CONDITION_ENABLED is not set. "Unacknowledged" if OPC_CONDITION_ACKED is not set. "Active" if OPC_CONDITION_ACKED is set. "Enabled" if OPC_CONDITION_ENABLED is set.
Quality	wQuality	The COM DA Quality converted to a UA StatusCode.
Severity	dwSeverity	Set based on the last *Event* received for the *Condition* instance. Set to the current value if the last *Event* is not available.
Comment	-	The value of the ACK_COMMENT *Attribute*
ClientUserId	szActorID	-
AckedState	wNewState	Set to "Acknowledged" or "Unacknowledged "
AckedState.Id	wNewState	Set to True if OPC_CONDITION_ACKED is set
ActiveState	wNewState	Set to "Active" or "Inactive "
ActiveState.Id	wNewState	Set to True if OPC_CONDITION_ACTIVE is set
ActiveState.TransitionTime	ftActiveTime	This time is set when the *ActiveState* transitions from False to True. NOTE Additional logic applies to exclusive limit alarms, in that the LimitState.TransitionTime also needs to be set, but this is set each time a limit is crossed (multiple limits might exist). For the initial transition to True the ftActiveTime is used for both LimitState.TransitionTime and ActiveState.TransitionTime. For subsequent transition the ActiveState.Transition time does not change, but the LimitState.TransitionTime will be updated with the new ftActiveTime. For example, if an alarm has Hi and HiHi limits, when the Hi limit is crossed and the alarm goes active the FTActiveTime is used for both times, but when the HiHi limit is later crossed, the FtActiveTime is only be used for the LimitState.TransitionTime. NOTE The ftActiveTime is part of the key for identifying the unique event in the A&E server and needs to be saved for processing any commands back to the A&E Server.

The A&C *Condition* Model defines other optional *Variable*s which are not needed in the A&E COM UA Wrapper. Any additional fields associated with *Event Attributes* are also reported.

D.2.6 Condition instances

Condition instances do not appear in the UA *Server* address space. *Conditions* ~~can~~ may be acknowledged by passing the EventId to the *Acknowledge Method* defined on the *AcknowledgeableConditionType*.

Conditions ~~cannot~~ may not be enabled or disabled via the COM A&E Wrapper.

D.2.7 Condition Refresh

The COM A&E Wrapper does not store the state of *Conditions*. When *ConditionRefresh* is called the *Refresh Method* is called on all COM AE *Subscriptions* associated with the *ConditionRefresh* call. The wrapper needs to wait until it receives the call back with the bLastRefresh flag set to True in the OnEvent call before it can tell the UA *Client* that the *Refresh* has completed.

D.3 Alarms and Events COM UA proxy

D.3.1 General

As illustrated in the figure below, the A&E COM UA Proxy is a COM *Server* combined with a UA *Client*. It maps the *Alarms* and *Conditions* address space of UA A&C *Server* into the appropriate COM Alarms and *Event Objects*.

Subclauses D.3.2 through D.3.9 identify the design guidelines and constraints used to develop the A&E COM UA Proxy provided by the OPC Foundation. In order to maintain a high degree of consistency and interoperability, it is strongly recommended that vendors, who choose to implement their own version of the A&E COM UA Proxy, follow these same guidelines and constraints.

The A&E COM *Client* simply needs to address how to connect to the UA A&C *Server*. Connectivity approaches include the one where A&E COM *Clients* connect to a UA A&C *Server* with a CLSID just as if the target *Server* were an A&E COM *Server*. However, the CLSID ~~can~~ may be considered virtual since it is defined to connect to intermediary components that ultimately connect to the UA A&C *Server*. Using this approach, the A&E COM *Client* calls co-create instance with a virtual CLSID as described above. This connects to the A&E COM UA Proxy components. The A&E COM UA Proxy then establishes a secure channel and session with the UA A&C *Server*. As a result, the A&E COM *Client* gets a COM *Event Server* interface pointer.

D.3.2 Server status mapping

The A&E COM UA Proxy reads the UA A&C *Server* status from the *Server Object Variable Node*. Status enumeration values that are returned in *ServerStatusDataType* structure ~~can~~ may be mapped 1 for 1 to the A&E COM *Server* status values with the exception of UA A&C *Server* status values *Unknown* and *Communication Fault.* These both map to the A&E COM *Server* status value of *Failed*.

The VendorInfo string of the A&E COM *Server* status is mapped from *ManufacturerName*.

D.3.3 Event Type mapping

Since all *Alarms* and *Conditions Events* belong to a subtype of *BaseEventType*, the A&E COM UA Proxy maps the subtype as received from the UA A&C *Server* to one of the three A&E *Event* types: Simple, Tracking and *Condition*. Figure D.2 shows the mapping as follows:

- those A&C *Events* which are of subtype *AuditEventType* are marked as A&E *Event* type Tracking;
- those A&C *Events* which are *ConditionType* are marked as A&E *Event* type *Condition*;
- those A&C *Events* which are of any subtype except *AuditEventType* or *ConditionType* are marked as A&E *Event* type Simple.

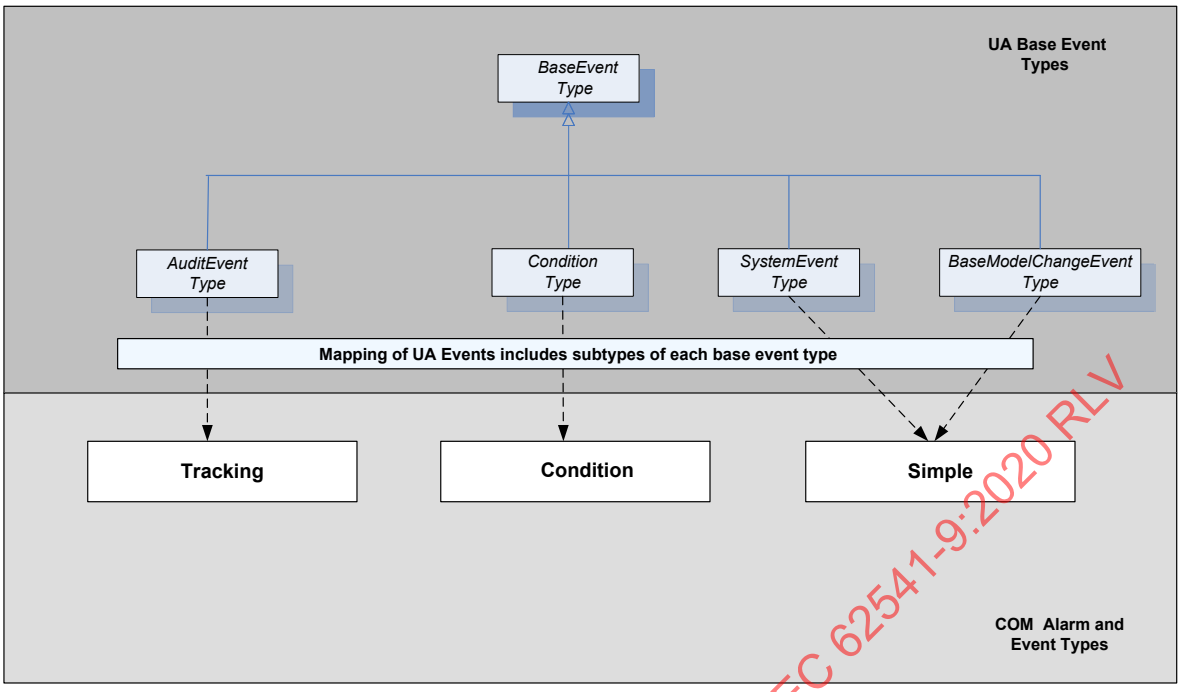


Figure D.2 – Mapping UA Event Types to COM A&E Event Types

Note that the *Event* type mapping described above also applies to the children of each subtype.

D.3.4 Event category mapping

Each A&E *Event* type (e.g. Simple, Tracking, *Condition*) has an associated set of *Event* categories which are intended to define groupings of A&E *Event*s. For example, Level and Deviation are possible *Event* categories of the *Condition Event* type for an A&E COM *Server*. However, since A&C does not explicitly support *Event* categories, the A&E COM UA Proxy uses A&C *Event* types to return A&E *Event* categories to the A&E COM *Client*. The A&E COM UA Proxy builds the collection of supported categories by traversing the type definitions in the address space of the UA A&C *Server*. Figure D.3 shows the mapping as follows:

- A&E Tracking categories consist of the set of all *Event* types defined in the hierarchy of subtypes of AuditEventType and TransitionEventType, including AuditEventType itself and TransitionEventType itself.
- A&E *Condition* categories consist of the set of all *Event* types defined in the hierarchy of subtypes of *ConditionType*, including *ConditionType* itself.
- A&E Simple categories consist of the set of *Event* types defined in the hierarchy of subtypes of *BaseEventType* excluding *AuditEventType* and *ConditionType* and their respective subtypes.

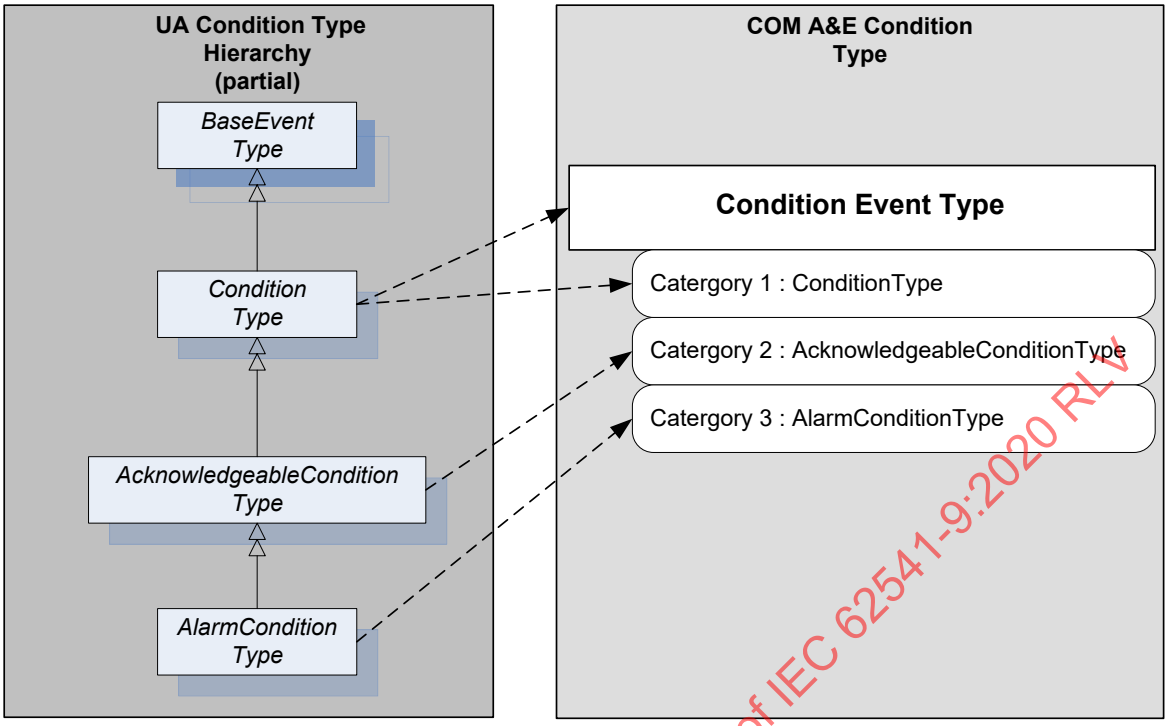


Figure D.3 – Example mapping of UA Event Types to COM A&E categories

Category name is derived from the display name *Attribute* of the *Node* type as discovered in the type hierarchy of the UA A&C *Server*.

Category description is derived from the description *Attribute* of the *Node* type as discovered in the type hierarchy of the UA A&C *Server*.

The A&E COM UA Proxy assigns Category IDs.

D.3.5 Event Category attribute mapping

The collection of *Attribute*s associated with any given A&E *Event* is encapsulated within the ONEVENTSTRUCT. Therefore, the A&E COM UA Proxy populates the *Attribute* fields within the ONEVENTSTRUCT using corresponding values from UA *Event Notification*s either directly (e.g. Source, Time, Severity) or indirectly (e.g. OPC COM *Event* category determined by way of the UA *Event* type). Table D.5 lists the *Attribute*s currently defined in the ONEVENTSTRUCT in the leftmost column. The rightmost column of Table D.5 indicates how the A&E COM UA proxy defines that *Attribute*.

Table D.5 – Event category attribute mapping table

A&E ONEVENTSTRUCT "attribute"	A&E COM UA Proxy Mapping
The following items are present for all A&E event types	
szSource	UA *BaseEventType Property*: *SourceName*
ftTime	UA *BaseEventType Property*: *Time*
szMessage	UA *BaseEventType Property*: *Message*
dwEventType	See D.3.3
dwEventCategory	See D.3.4
dwSeverity	UA *BaseEventType Property*: *Severity*
dwNumEventAttrs	Calculated within A&E COM UA Proxy
pEventAttributes	Constructed within A&E COM UA Proxy
The following items are present only for A&E Condition-Related Events	
szConditionName	UA *ConditionType Property*: *ConditionName*
szSubConditionName	UA *ActiveState Property*: *EffectiveDisplayName*
wChangeMask	Calculated within Alarms and Events COM UA proxy
wNewState: OPC_CONDITION_ACTIVE	A&C *AlarmConditionType Property*: ActiveState Note that events mapped as non-*Condition Event*s and those that do not derive from *AlarmConditionType* are set to ACTIVE by default.
wNewState: OPC_CONDITION_ENABLED	A&C *ConditionType Property*: EnabledState Note, *Event*s mapped as non-*Condition Event*s are set to ENABLED (state bit mask = 0x1) by default.
wNewState: OPC_CONDITION_ACKED	A&C *AcknowledgeableConditionType Property*: AckedState Note that A&C *Event*s mapped as non-*Condition Event*s or which do not derive from *AcknowledgeableConditionType* are set to UNACKNOWLEDGED and AckRequired = False by default.
wQuality	A&C ConditionType *Property*: Quality Note that *Event*s mapped as non-*Condition Event*s are set to OPC_QUALITY_GOOD by default. In general, the Severity field of the StatusCode is used to map COM status codes OPC_QUALITY_BAD, OPC_QUALITY_GOOD and OPC_QUALITY_UNCERTAIN. When possible, specific status' are mapped directly. These include (UA => COM): Bad status codes Bad_ConfigurationError => OPC_QUALITY_CONFIG_ERROR Bad_NotConnected => OPC_QUALITY_NOT_CONNECTED Bad_DeviceFailure => OPC_QUALITY_DEVICE_FAILURE Bad_SensorFailure => OPC_QUALITY_SENSOR_FAILURE Bad_NoCommunication => OPC_QUALITY_COMM_FAILURE Bad_OutOfService => OPC_QUALITY_OUT_OF_SERVICE Uncertain status codes Uncertain_NoCommunicationLastUsableValue => OPC_QUALITY_LAST_USABLE Uncertain_LastUsableValue => OPC_QUALITY_LAST_USABLE Uncertain_SensorNotAccurate => OPC_QUALITY_SENSOR_CAL Uncertain_EngineeringUnitsExceeded => OPC_QUALITY_EGU_EXCEEDED Uncertain_SubNormal => OPC_QUALITY_SUB_NORMAL Good status codes Good_LocalOverride => OPC_QUALITY_LOCAL_OVERRIDE
bAckRequired	If the ACKNOWLEDGED bit (OPC_CONDITION_ACKED) is set then the Ack Required Boolean is set to False, otherwise the Ack Required Boolean is set to True. If the *Event* is not of type *AcknowledgeableConditionType* or subtype, then the AckRequired Boolean is set to False.

A&E ONEVENTSTRUCT "attribute"	A&E COM UA Proxy Mapping
ftActiveTime	If the *Event* is of type *AlarmConditionType* or subtype and a transition from *ActiveState* of False to *ActiveState* to True is being processed then the *TransitionTime Property* of *ActiveState* is used. If the *Event* is not of type *AlarmConditionType* or subtype then this field is set to current time. Note: Additional logic applies to exclusive limit alarms, This value should be mapped to the LimitState.TransitionTime.
dwCookie	Generated by the A&E COM UA Proxy. These unique *Condition Event* cookies are not associated with any related identifier from the address space of the UA A&C *Server*.
The following is used only for A&E tracking events and for A&E condition-relate events which are acknowledgement notifications	
szActorID	
Vendor specific *Attributes* – ALL	
ACK Comment	
AREAS	All A&E *Event*s are assumed to support the "Areas" *Attribute*. However, no *Attribute* or *Property* of an A&C *Event* is available which provides this value. Therefore, the A&E COM UA Proxy initializes the value of the Areas *Attribute* based on the *MonitoredItem* producing the *Event*. If the A&E COM *Client* has applied no area filtering to a *Subscription*, the corresponding A&C *Subscription* will contain just one *MonitoredItem* – that of the UA A&C *Server Object*. *Event*s forwarded to the A&E COM *Client* on behalf of this *Subscription* will carry an Areas *Attribute* value of empty string. If the A&E COM *Client* has applied an area filter to a *Subscription* then the related UA A&C *Subscription* will contain one or more *MonitoredItem*s for each notifier *Node* identified by the area string(s). *Event*s forwarded to the A&E COM *Client* on behalf of such a *Subscription* will carry an areas *Attribute* whose value is the relative path to the notifier which produced the *Event* (i.e. the fully qualified area name).
Vendor specific *Attribute*s – based on category	
SubtypeProperty1	All the UA A&C subtype *Properties* that are not part of the standard set exposed by *BaseEventType* or *ConditionType*
SubtypeProperty*n*	

Condition Event instance records are stored locally within the A&E COM UA Proxy. Each record holds ONEVENTSTRUCT data for each EventSource/*Condition* instance. When the *Condition* instance transitions to the state INACTIVE|ACKED, where AckRequired = True or simply INACTIVE, where AckRequired = False, the local *Condition* record is deleted. When a *Condition Event* is received from the UA A&C *Server* and a record for this *Event* (identified by source/*Condition* pair) already exists in the proxy *Condition Event* store, the existing record is simply updated to reflect the new state or other change to the *Condition*, setting the change mask accordingly and producing an OnEvent callback to any subscribing *Client*s. In the case where the *Client* application acknowledges an *Event* which is currently unacknowledged (AckRequired = True), the UA A&C *Server* Acknowledge *Method* associated with the *Condition* is called and the subsequent *Event* produced by the UA A&C *Server* indicating the transition to acknowledged will result in an update to the current state of the local *Condition* record, as well as an OnEvent *Notification* to any subscribing *Client*s.

The A&E COM UA Proxy maintains the mapping of *Attribute*s on an *Event* category basis. An *Event* category inherits its *Attribute*s from the *Properties* defined on all supertypes in the UA *Event* Type hierarchy. New *Attribute*s are added for any *Properties* defined on the direct UA *Event* type to A&E category mapping. The A&E COM UA Proxy adds two *Attribute*s to each category: AckComment and Areas. Figure D.4 shows an example of this mapping.

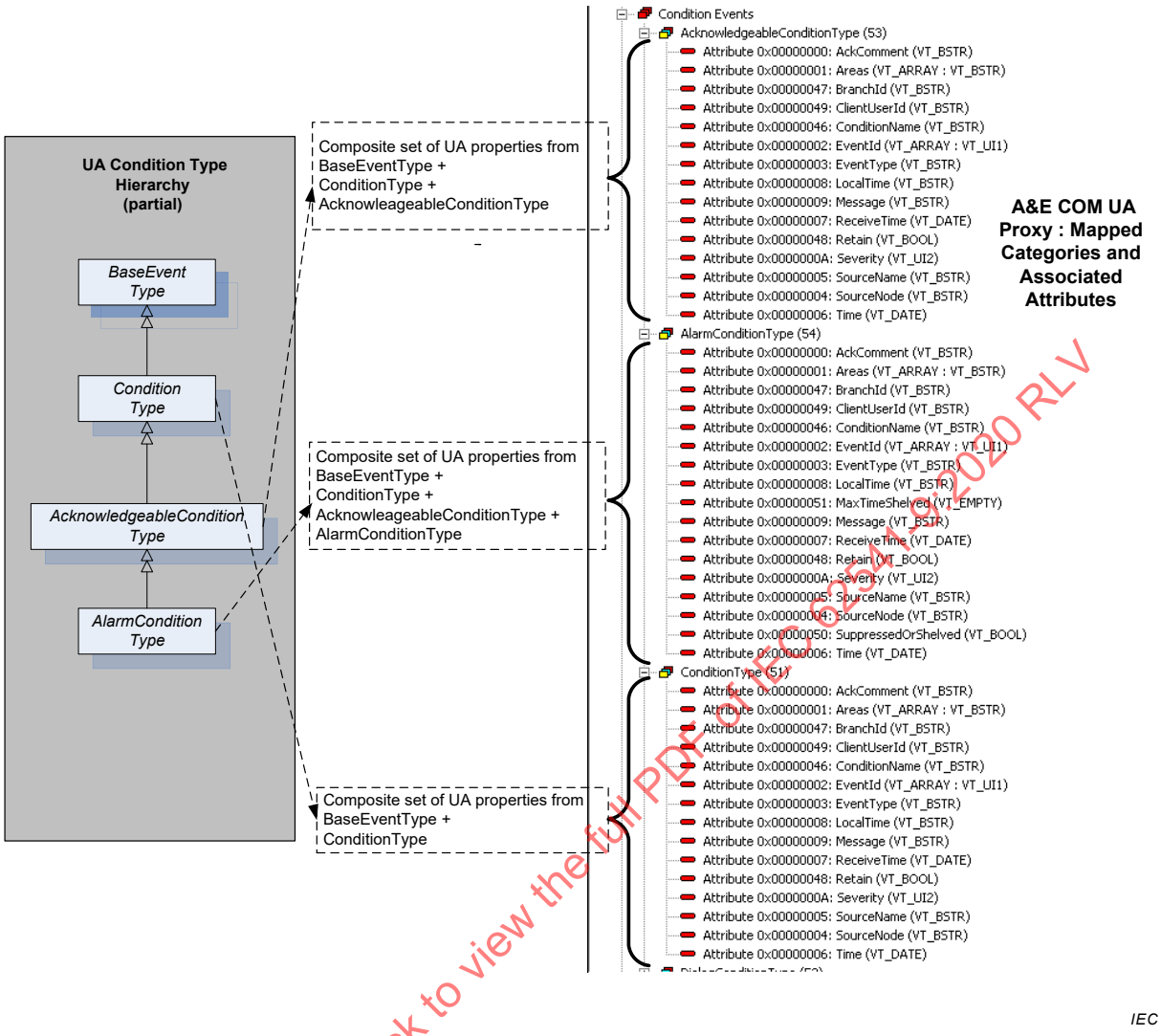


Figure D.4 – Example mapping of UA Event Types to A&E categories with attributes

D.3.6 Event Condition mapping

*Event*s of any subtype of *ConditionType* are designated COM *Condition Event*s and are subject to additional processing due to the stateful nature of *Condition Event*s. COM *Condition Event*s transition between states composed of the triplet ENABLED|ACTIVE|ACKNOWLEDGED. In UA A&C, *Event* subtypes of *ConditionType* only carry a value which can be mapped to ENABLED (DISABLED) and optionally, depending on further sub typing, may carry additional information which can be mapped to ACTIVE (INACTIVE) or ACKNOWLEDGED (UNACKNOWLEGED). *Condition Event* processing proceeds as described in Table D.5 (see A&E ONEVENTSTRUCT "*Attribute*" rows: OPC_CONDITION_ACTIVE, OPC_CONDITION_ENABLED and OPC_CONDITION_ACKED).

D.3.7 Browse mapping

A&E COM browsing yields a hierarchy of areas and sources. Areas can contain both sources and other areas in tree fashion where areas are the branches and sources are the leaves. The A&E COM UA Proxy relies on the "HasNotifier" *Reference* to assemble a hierarchy of branches/areas such that each *Object Node* which contains a *HasNotifier Reference* and whose *EventNotifier Attribute* is set to *SubscribeToEvents* is considered an area. The root for the ~~*Event*~~ *HasNotifier* hierarchy is the *Server Object*. Starting at the *Server Object*, ~~eventNotifier~~ *HasNotifier References* are followed and each *HasNotifier* target whose EventNotifier *Attribute* is set to SubscribeToEvents becomes a nested COM area within the hierarchy.

~~Note that~~ The HasNotifier target ~~can~~ may also be a HasNotifier source. Further, any *Node* which is a HasEventSource source and whose EventNotifier *Attribute* is set to SubscribeToEvents is also considered a COM Area. The target *Node* of any HasEventSource *Reference* is considered an A&E COM "source" or leaf in the A&E COM browse tree.

In general, *Node*s which are the source *Node*s of the HasEventSource *Reference* and/or are the source *Node*s of the HasNotifier *Reference* are always A&ECOM Areas. *Node*s which are the target *Node*s of the HasEventSource *Reference* are always A&E COM Sources. Note however that targets of HasEventSource which cannot be found by following the HasNotifier *Reference*s from the *Server Object* are ignored.

Given the above logic, the A&E COM UA Proxy browsing will have the following limitations: Only those *Node*s in the UA A&C *Server*'s address space which are connected by the HasNotifier *Reference* (with exception of those contained within the top level *Object*s folder) are considered for area designation. Only those *Node*s in the UA A&C *Server*'s address space which are connected by the HasEventSource *Reference* (with exception of those contained within the top level *Object*s folder) are considered for area or source designation. To be an area, a *Node* shall contain a HasNotifier *Reference* and its EventNotifier *Attribute* shall be set to SubscribeToEvents. To be a source, a *Node* shall be the target *Node* of a HasEventSource *Reference* and shall have been found by following HasNotifier *Reference*s from the *Server Object*.

D.3.8 Qualified names

D.3.8.1 Qualified name syntax

From the root of any sub tree in the address space of the UA A&C *Server*, the A&E COM *Client* may request the list of areas and/or sources contained within that level. The resultant list of area names or source names will consist of the set of browse names belonging to those *Node*s which meet the criteria for area or source designation as described above. These names are "short" names meaning that they are not fully qualified. The A&E COM *Client* may request the fully qualified representation of any of the short area or source names. In the case of sources, the fully qualified source name returned to the A&E COM *Client* will be the string encoded value of the *NodeId* as defined in IEC 62541-6 (e.g., "ns=10;i=859"). In the case of areas, the fully qualified area name returned to the COM *Client* will be the relative path to the notifier *Node* as defined in IEC 62541-4 (e.g., "/6:Boiler1/6:Pipe100X/1:Input/2:Measurement"). Relative path indices refer to the namespace table described in D.3.8.2.

D.3.8.2 Namespace table

UA *Server* Namespace table indices may vary over time. This represents a problem for those A&E COM *Client*s which cache and reuse fully qualified area names. One solution to this problem would be to use a qualified name syntax which includes the complete URIs for all referenced table indices. This, however, would result in fully qualified area names which are unwieldy and impractical for use by A&E COM *Client*s. As an alternative, the A&E COM UA Proxy will maintain an internal copy of the UA A&C *Server*'s namespace table together with the locally cached endpoint description. The A&E COM UA Proxy will evaluate the UA A&C *Server*'s namespace table at connect time against the cached copy and automatically handle any re-mapping of indices if required. The A&E COM *Client* can continue to present cached fully qualified area names for filter purposes and the A&E COM UA Proxy will ensure these names continue to reference the same notifier *Node* even if the *Server*'s namespace table changes over time.

To implement the relative path, the A&E COM UA Proxy maintains a stack of *INode* interfaces of all the *Node*s browsed leading to the current level. When the A&E COM *Client* calls GetQualifiedAreaName, the A&E COM UA Proxy first validates that the area name provided is a valid area at the current level. Then looping through the stack, the A&E COM UA Proxy builds the relative path. Using the browse name of each *Node*, the A&E COM UA Proxy constructs the translated name as follows:

QualifiedName translatedName = new QualifiedName(Name,(ushort) ServerMappingTable[NamespaceIndex]) where

Name – the unqualified browse name of the *Node*

NamespaceIndex – the *Server* index

the *ServerMappingTable* provides the *Client* namespace index that corresponds to the *Server* index.

A '/' is appended to the translated name and the A&E COM UA Proxy continues to loop through the stack until the relative path is fully constructed.

D.3.9 Subscription filters

D.3.9.1 General

The A&E COM UA Proxy supports all of the defined A&E COM filter criteria.

D.3.9.2 Filter by Event, category or severity

These filter types are implemented using simple numeric comparisons. For *Event* filters, the received *Event* shall match the *Event* type(s) specified by the filter. For Category filters, the received *Event*'s category (as mapped from UA *Event* type) shall match the category or categories specified by the filter. For severity filters, the received *Event* severity shall be within the range specified by the *Subscription* filter.

D.3.9.3 Filter by source

In the case of source filters, the UA A&C *Server* is free to provide any appropriate, *Server*-specific value for SourceName. There is no expectation that source *Node*s discovered via browsing can be matched to the SourceName *Property* of the *Event* returned by the UA A&C Server using string comparisons. Further, the A&E COM *Client* may receive *Event*s from sources which are not discoverable by following only HasNotifier and/or HasEventSource *Reference*s. Thus, source filters will only apply if the source string can be matched to the SourceName *Property* of an *Event* as received from the target UA A & C *Server*. Source filter logic will use the pattern matching rules documented in the A&E COM specification, including the use of wildcard characters.

D.3.9.4 Filter by area

The A&E COM UA Proxy implements Area filtering by adjusting the set of *MonitoredItem*s associated with a *Subscription*. In the simple case where the *Client* selects no area filter, the A&E COM UA Proxy will create a UA *Subscription* which contains just one *MonitoredItem*, the *Server Object*. In doing so, the A&E COM UA Proxy will receive *Event*s from the entire *Server* address space – that is, all Areas. The A&E COM *Client* will discover the areas associated with the UA *Server* address space by browsing. The A&E COM *Client* will use GetQualifiedAreaName as usual in order to obtain area strings which ~~can~~ may be used as filters. When the A&E COM *Client* applies one or more of these area strings to the COM *Subscription* filter, the A&E COM UA Proxy will create *MonitoredItem*s for each notifier *Node* identified by the area string(s). Recall that the fully qualified area name is in fact the namespace qualified relative path to the associated notifier *Node*.

The A&E COM UA Proxy calls the *TranslateBrowsePathsToNodeIds Service* to get the *Node* ids of the fully qualified area names in the filter. The *Node* ids are then added as *MonitoredItem*s to the UA *Subscription* maintained by the A&E COM UA Proxy. The A&E COM UA Proxy also maintains a reference count for each of the areas added, to handle the case of multiple A&E COM *Subscription* applying the same area filter. When the A&E COM *Subscription*s are removed or when the area name is removed from the filter, the ref count on the *MonitoredItem* corresponding to the area name is decremented. When the ref count goes to zero, the *MonitoredItem* is removed from the UA *Subscription.*

As with source filter strings, area filter strings ~~can~~ may contain wildcard characters. Area filter strings which contain wildcard characters require more processing by the A&E COM UA Proxy. When the A&E COM *Client* specifies an area filter string containing wildcard characters, the A&E COM UA Proxy will scan the relative path for path elements that are completely specified. The partial path containing just those segments which are fully specified represents the root of the notifier sub tree of interest. From this sub tree root *Node*, the A&E COM UA Proxy will collect the list of notifier *Node*s below this point. The relative path associated with each of the collected notifier *Node*s in the sub tree will be matched against the *Client* supplied relative path containing the wildcard character. A *MonitoredItem* is created for each notifier *Node* in the sub-tree whose relative path matches that of the supplied relative path using established pattern matching rules. An area filter string which contains wildcard characters may result in multiple *MonitoredItem*s added to the UA *Subscription*. By contrast, an area filter string made up of fully specified path segments and no wildcard characters will result in one *MonitoredItem* added to the UA *Subscription*. So, the steps involved are:

1) check if the filter string contains any of these wild card characters, '*', '?', '#', '[', ']', '!', '-';
2) scan the string for path elements that are completely specified by retrieving the substring up to the last occurrence of the '/' character;
3) obtain the *Node*Id for this path using *TranslateBrowsePathsToNodeIds*;
4) browse the *Node* for all notifiers below it;
5) using the ComUtils.Match() function match the browse names of these notifiers against the *Client* supplied string containing the wild card character;
6) add the *Node* ids of the notifiers that match as *MonitoredItem*s to the UA *Subscription*.

Annex E
(informative)

IEC 62682 Mapping

E.1 Overview

This annex provides a description of how the IEC 62682 information model may be mapped to OPC UA. It highlights term differences, concepts and other functionality. IEC 62682 provides additional information about managing and limiting alarms not covered by this specification.

NOTE ISA 18.2 is not discussed by this mapping, but IEC 62682 and ISA 18.2 are related and most definitions in ISA 18.2 correspond to the definitions in IEC 62682.

E.2 Terms

IEC 62682 defines a large number of terms that are covered by the OPC UA model but not used in the text. These IEC 62682 terms are listed in Table E.1 and include a description, mapping or relationship to OPC UA Alarms and Events:

Table E.1 – IEC 62682 Mapping

IEC 62682	OPC UA Mapping / Related Concept	IEC 62682 Definition OPC UA Application of
absolute alarm	ExclusiveDeviationAlarmType NonExclusiveDeviationAlarmType	An alarm generated when the alarm set point is exceeded.
		Both OPC UA models expose a set point and process the *Alarm* as an absolute *Alarm* requires, the only difference is the interaction between relative states (High, HighHigh...)
adaptive alarm		Alarm for which the setpoint is changed by an algorithm (e.g. rate based).
		In OPC UA, adaptive alarming may be part of a vendor specific alarm application, but it would or could make use of a number of standard *Alarm* functions described in this specification. OPC UA provides limit, rate of change and deviation alarming. Vendors may easily develop algorithms to adjust any of the limits that are exposed.
adjustable alarm / operator-set alarm	ExclusiveLimitAlarmType NonExclusiveLimitAlarmType	An alarm for which the set point can be changed manually by the *Operator*.
		Both OPC UA models allow *Alarm* limits to be writeable and allow for an *Operator* to change the limit. For all changes to limits, an audit event should be generated tracking the change.
advanced alarming		A collection of techniques that can help manage annunciations during specific situations.
		In OPC UA advanced alarming may be part of a vendor specific alarm application, but it would or could make use of a number of standard *Alarm* functions described in this specification, such as adaptive setting of a setpoint for deviation *Alarm*. It might also require the definition of new *Alarm* subtypes.

IEC 62682	OPC UA Mapping / Related Concept	IEC 62682 Definition OPC UA Application of
Annunciation / Alarm Annunciation	Retain	A function of the alarm system is to call the attention of the *Operator* to an alarm.
		OPC UA provides an *Alarm* model that includes concepts such as re-alarming, *Alarm* silence and *Alarm* delays, but it is up to the *Client* application to make use of these features to generate both audible and visual annunciation to the *Operator*. OPC UA does not provide visual indication but it does provide priority information on which the client can be configured to provide the appropriate visual display. A key concept for alarm display is the concept of *Alarm* states and a Retain bit (see Annex B for more details).
alarm attribute	Various Alarm Properties	The setting for an alarm within the process control system.
		OPC UA defines a number of Properties that reflect what would be termed alarm attributes in IEC 62682 such as *Alarm* setpoint which maps to the setpoint property in an *ExclusiveDeviationAlarmType*.
alarm class	ConditionClass, ConditionSubClass	A group of alarms with a common set of alarm management requirements (e.g. testing, training, monitoring, and audit requirements).
		OPC UA provides *ConditionClasses*, but also provides other groupings, like ConditionSubClass OPC UA also specifies a number of predefined classes, but it is expected that vendors, other standards group or even end users will define their own extensions to these classes. The OPC concepts allow *Alarm*s to be categorized as needed.
alarm Deadband	ExclusiveDeviationAlarmType NonExclusiveDeviationAlarmType	A change in signal from the alarm setpoint necessary for the alarm to return to normal.
		In OPC UA, the *ExclusiveDeviationAlarmType* and *NonExclusiveDeviationAlarmType* contain an *Alarm* deadband and can be used for the same functionality described in IEC 62682.
filtering(alarm)	Event Subscription	A function which selects alarm records to be displayed according to a given element of the alarm record.
		In OPC UA, *Alarms* are received by a *Client* according to the specific filter requested by the *Client*. The filtering can be very robust or very simple according to the needs of the client. It is up to the *Client* application to generate and provide the appropriate filter to the server. OPC UA's *Alarm* model is a subscription-based model, not a push model that is configured on a server. The choice of filter is a client's responsibility.
alarm flood	Alarm diagnostics	A condition during which the *Alarm* rate is greater than the *Operator* can effectively manage – (e.g. more than 10 *Alarm*s per 10 min).
		OPC UA does not define *Alarm* flooding but it does provide the capability to collect diagnostics that would allow an engineer to review overall *Alarm* performance.
alarm group	alarm group	A set of alarms with common association (e.g. process unit, process area, equipment set, or service). Alarm groups are primarily used for display purposes.
		OPC UA allows the definition of *Alarm* groups and the assignment of *Alarms* to these groups. In addition. OPC UA allows *Alarms* to also be part of a category. OPC UA also allows *Alarms* to be organized as a *HasNotifier* hierarchy (see Clause 6). Groups, categories and hierarchies can be used for filtering or restricting *Alarms* that are being displayed.
alarm history	historical events	long-term repository for alarm records.
		IEC 62541-11 describes historical *Events*.

IEC 62682	OPC UA Mapping / Related Concept	IEC 62682 Definition OPC UA Application of
alarm log		short-term repository for alarm records.
		This part does not specify repositories for *Alarms*. *Alarm* logging is a *Client* function.
alarm management alarm system management		collection of processes and practices for determining, documenting, designing, operating, monitoring, and maintaining alarm systems.
		OPC UA provides an infrastructure to allow vendors and *Operator*s to provide *Alarm* management, as such it should be an integral part of an alarm management system.
alarm message	Events	text string displayed with the alarm indication that provides additional information to the *Operator* (e.g., *Operator* action).
		OPC UA provides an *Event* structure that includes many different pieces of information (see IEC 62541-5 for additional details). *Client*s can subscribe for as much of this information as desired and display this as an *Alarm* message. All typical fields that would be associated with an *Alarm* message are available. In addition, OPC UA provides significant additional information.
alarm priority	Priority	relative importance assigned to an alarm within the alarm system to indicate the urgency of response (e.g., seriousness of consequences and allowable response time)
		OPC UA provides a *Priority Variable* as part of the *Alarm Object* that provides the same functionality
alarm rate	Alarm diagnostics	the number of alarm annunciation, per *Operator*, in a specific time interval.
		OPC UA provides diagnostics allowing the collection of *Alarm* rate information at any level in the system.
Record (Alarm)	Events, Event filtering	a set of information which documents an alarm state change.
		In OPC UA all *Alarm*s are generated as an *Event* and the *Client* can select the fields that are to be included in the *Events*. This selection can be customized for each *AlarmConditionType*, which allows a customized *Alarm* record to be generated.
alarm setpoint, alarm limit, alarm trip point	Limit Alarms, Discrete Alarms	the threshold value of a process variable or discrete state that triggers the alarm indication.
		OPC UA supports *Alarm* limits and setpoints for multiple *Alarm* types, including limit *Alarms* and discrete *Alarms*.
Sorting (alarm)		a function which orders alarm records to be displayed according to a given element of alarm record.
		OPC UA does not provide *Alarm* sorting as part of an event subscription. Multiple filtering options are provided, but the *Client* is required to perform any ordering of *Alarms*.
alarm summary, alarm list		a display that lists alarm annunciations with selected information (e.g. date, time, priority, and alarm type).
		In OPC UA *Alarm* summaries and *Alarm* lists are *Client* functionality and are not specified. Extensive filtering capabilities are provided by the *Server* to allow easier implementation of *Alarm* summaries or lists by a *Client*.

IEC 62682	OPC UA Mapping / Related Concept	IEC 62682 Definition OPC UA Application of
Alert		An audible and/or visible means of indicating to the *Operator* an equipment or process condition that can require evaluation when time allows.
		Alerts are items that should be attended to, but are not as urgent as *Alarms*. OPC UA does not differentiate between *Alarms* and alerts, but it does provide a full range of priorities for *Alarms*. It is up to the end users to determine what range of priorities are considered an alert vs an *Alarm* etc.
allowable response time		The maximum time between the annunciation of the alarm and when the *Operator* takes corrective action to avoid the consequence.
		OPC UA does not provide any specific fields for allowable response time, but it does track the times at which an *Alarm* occurs and when any actions are taken on the *Alarm*.
annunciator		device or group of devices that call attention to changes in process conditions
		OPC UA does not define annunciators, this is *Client* functionality that can be implemented using OPC UA
Audit		comprehensive assessment that includes the evaluation of alarm system performance and the effectiveness of the work practices used to administer the alarm system.
		OPC UA does provide a number of features that can facilitate an audit, including diagnostics and audit events. Do not confuse OPC *Audit Event* with the IEC audit concept.
bad-measurement alarm		an alarm generated when the signal for a process measurement is outside the expected range (e.g. 3.8 mA for a 4 mA to 20 mA signal).
		A bad measurement *Alarm* is not defined in OPC UA, but limit *Alarms* are defined and they could be used directly to represent a bad-measurement *Alarm*. Alternatively, limit *Alarms* could be further subtyped to allow easier filtering on bad-measurement *Alarms* if desired.
bit-pattern alarm	Discrete alarm	an alarm that is generated when a pattern of digital signals matches a predetermined pattern.
		In OPC UA a bit pattern *Alarm* can be mapped to a *DiscreteAlarmType*.
calculated alarm		An alarm generated from a calculated value instead of a direct process measurement.
		In OPC UA any of the defined *Alarm* types can be applied to calculated values or to process values.
call-out alarm		alarm that notifies and informs an *Operator* by means other than, or in addition to, a console display (e.g. pager or telephone)
		OPC UA does not specify call-out alarms, since this is client functionality. OPC UA does provide the ability to categorize or group an *Alarm* such that it could be easily identified as requiring a different type of annunciation.
chattering alarm	OnDelay, OffDelay	alarm that repeatedly transitions between the alarm state and the normal state in a short period of time.
		The OPC UA features of *OnDelay* and *OffDelay* can be used to help control chattering *Alarms*.

IEC 62682	OPC UA Mapping / Related Concept	IEC 62682 Definition
		OPC UA Application of
classification	ConditionClasses	the process of separating alarms into alarm classes based on common requirements (e.g. testing, training, monitoring, and auditing requirements).
		OPC defines a number of extensible *ConditionClasses* that can be used for this purpose.
controller-output alarm		alarm generated from the output signal of a control algorithm (e.g. PID controller) instead of a direct process measurement.
		OPC UA does not provide an *Alarm* type for controller-output alarm, but a type could be created or an existing type could be used, depending on the requirements.
dynamic alarming		An automatic modification of alarm attributes based on process state or conditions.
		OPC UA does not define dynamic alarming behaviour, but it allows programmatic access to limits, set points or other parameters that would be required for a dynamic alarming solution.
enforcement		enhanced alarming technique that can verify and restore alarm attributes in the control system to the values in the master alarm database.
		OPC UA does not provide enforcement, but it enables enforcement by providing an information model that includes default setting for *Alarm* types as well as original settings for dynamic *Alarms*. These features may be used by a *Client* application to provide enforcement.
fleeting alarm	Suppression, Shelving	An alarm that transitions between an active alarm state and an inactive alarm state in a short period of time.
		OPC UA provides *Alarm Suppression* and *Shelving* which an *Operator* might use to control fleeting *Alarms*.
first-out alarm first-up alarm	FirstInGroup FirstInGroupFlag	An alarm determined (i.e. by first-out logic) to be the first, in a multiple-alarm scenario.
		OPC UA can support first-up/first-out *Alarms* as part of the *Alarm* information model, including definition of the group of *Alarms*.
instrument diagnostic alarm	InstrumentDiagnosticAlarmType	An alarm generated by a field device to indicate a fault (e.g. sensor failure).
		OPC UA provides support for InstrumentDiagnostic Alarms that can be used to represent a failed sensor or an instrument diagnostic.
monitoring	Alarm Diagnostics	measurement and reporting of quantitative (objective) aspects of alarm system performance.
		OPC UA provides diagnostic collection capabilities that can be used to measure and reports quantitative information related to alarm system performance.
nuisance alarm	Alarm Diagnostics	An alarm that annunciates excessively, unnecessarily, or does not return to normal after the *Operator* response is taken. EXAMPLE: Chattering alarm, fleeting alarm, or stale alarm.
		The OPC UA model provides Alarm Diagnostics for tracking the information needed to identify if an *Alarm* is a nuisance *Alarm* (i.e. has been in an *Alarm* state excessively or does not return to normal).
plant state plant mode	StateMachines	defined set of operational conditions for a process plant.
		OPC UA provides an example StateMachine (see Annex F) that can be customized or adapted to provide process information. This StateMachine could also be used to affect alarming.

IEC 62682	OPC UA Mapping / Related Concept	IEC 62682 Definition
		OPC UA Application of
process area	Event Hierarchies Object References (IEC 62541-5)	physical, geographical or logical grouping of resources determined by the site.
		OPC UA provides multiple manners in which an information model can be displayed, this includes grouping objects into process areas or any other desired grouping. This is an inherent part of the OPC UA information model.
re-alarming alarm, re-triggering alarm	ReAlarmTime ReAlarmRepeatCount	alarm that is automatically re-annunciated to the *Operator* under certain conditions.
		OPC UA supports re-alarming as part of its base AlarmConditionType.
recipe-driven alarm	StateMachines Alarm Limits	alarm with setpoints that depend on the recipe that is currently being executed.
		OPC UA provides support for adjustable *Alarm* limits. It also provides support for programs and other functionality that could be used to drive recipes. Annex F provides an example of a *StateMachine* and how it could be used to adjust *Alarm* settings.
Reset	LatchedState / Reset	*Operator* action that unlatches a latched alarm.
		OPC UA provides an optional *StateMachine* to indicate an Alarm is capable of being latched and is in a latched state. It also provides a *Reset Method* for clearing the latched state.
safety related alarm safety alarm	SafetyConditionClassType	an alarm that is classified as critical to process safety for the protection of human life or the environment.
		OPC UA defines a safety *ConditionClass* for grouping safety related alarms.
stale alarm	Alarm Diagnostics	alarm that remains annunciated for an extended period of time (e.g. 24 hours).
		OPC UA *Alarm Diagnostics* can track the length of time an *Alarm* is active.
state-based alarm – mode-based alarms	StateMachine	alarm that has attributes modified or is suppressed based on operating states or process conditions.
		OPC UA can provide a system state *StateMachine* to support process, device or system states (see Annex F). With this *StateMachine Servers* can adjust *Alarm* attributes or just *Suppress* or *Disable Alarms* based on the *StateMachine*. The *StateMachine* can be applied at multiple levels in the system.
statistical alarm	StatisticalConditionClassType	alarm generated based on statistical processing of a process variable or variables.
		OPC UA provides an *Alarm Condition* class that any of the existing *AlarmConditionTypes* can be assigned to. This allows any *Alarm* types, such as limit *Alarm*s, to be generated by statistical analysis.
Suppress	SuppressedOrShelved	Any mechanism to prevent the indication of the alarm to the *Operator* when the base alarm condition is present (i.e. shelving, suppressed by design, out-of-service).
		OPC UA provides a flag *SuppressedOrShelved* that matches this functionality.
suppressed by design	SuppressedState	alarm annunciation to the *Operator* prevented based on plant state or other conditions.
		OPC UA provides a SuppressedState that matches this functionality.

IEC 62682	OPC UA Mapping / Related Concept	IEC 62682 Definition
		OPC UA Application of
system diagnostic alarm	SystemDiagnosticAlarmType	alarm generated by the control system to indicate a fault within the system hardware, software or components.
		OPC UA defines a system diagnostic *Alarm* that can be used to represent faults with system hardware, software or components.,

The following terms in IEC 62682 match the terms/concepts defined in the OPC UA specification and do not need any additional mapping or discussion:

- Acknowledge
- Active
- Alarm
- Alarm OffDelay
- Alarm OnDelay
- Alarm Type
- Deviation Alarm
- Discrepancy Alarm
- Event
- Highly Managed Alarm
- LatchingAlarm
- OutofService
- Rateofchange alarms
- Return to normal
- Shelve
- Silence
- Unacknowledged

E.3 Alarm records and State indications

OPC UA provides all of the items listed as both required and recommended as part of its alarm definitions, but it is up to the client to subscribe for the information. In OPC UA the Client controls what alarm information is requested and obtained from the *Server*. The *Server* does not define visual aspects of the alarm system, but does provide priority information from which the visual aspect can be set on the client side.

OPC UA also supports all of the states described in IEC 62682. This includes tracking the process states, system states and individual alarm states. OPC UA also provides a StateMachine model that can be used in conjunction with an alarm system to alter alarm behaviour based on the state of a system or process. For example, during start-up or shutdown of a process or a system, some alarms might be suppressed.

The behaviour of an OPC UA alarm system also mimics that required by IEC 62682. All behaviour described in IEC 62682 can easily be mapped to functionality define in OPC UA Alarm & Conditions.

Annex F
(informative)

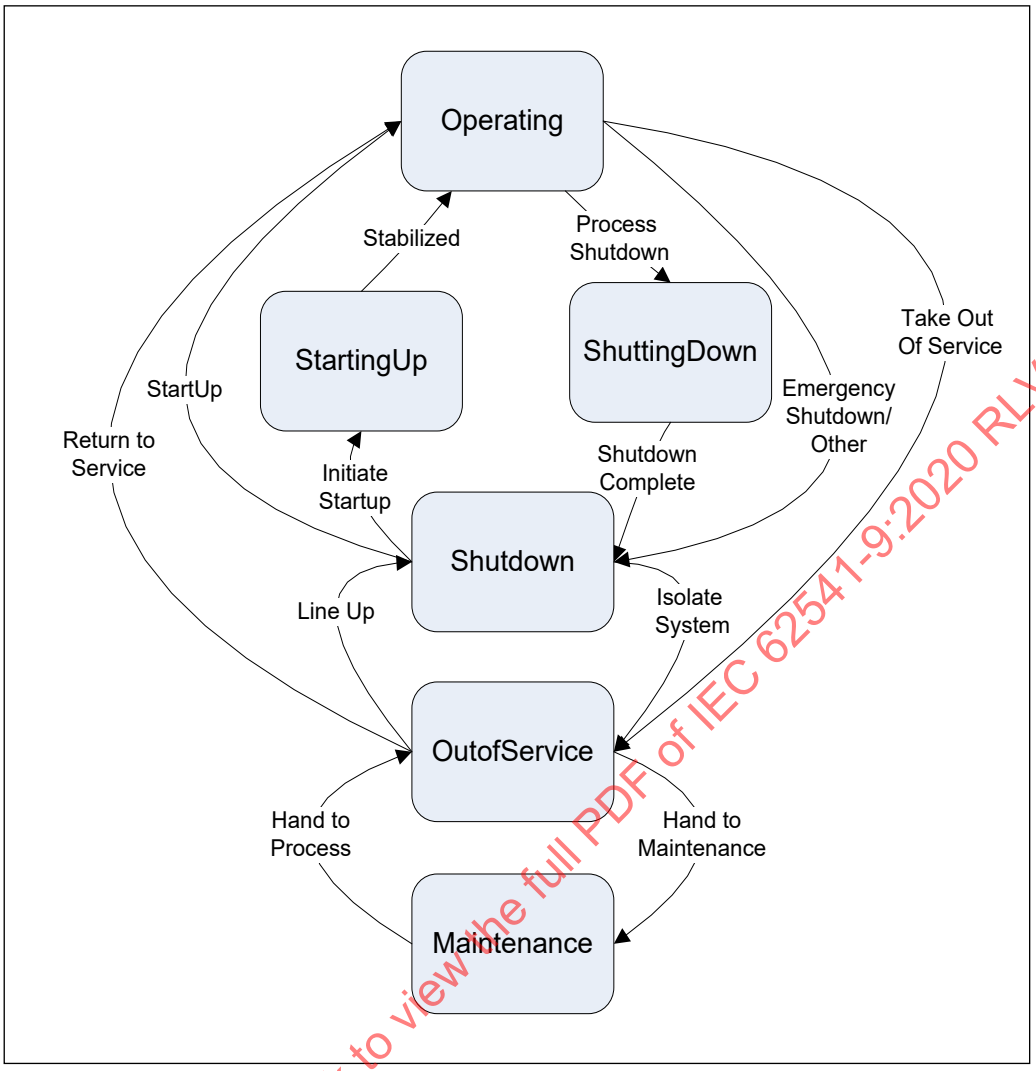
System State

F.1 Overview

The state of alarms is affected by the state of the process, equipment, system or plant. For example, when a tank is taken out of service, the level alarms associated with the tank would be no longer used, until the tank is returned to service. This annex describes a *StateMachine* that can be deployed as part of a system designed and used to reflect the current state of the system, process, equipment or item. A customized version of this model can be implemented for any system, this sample is just an illustration.

The current state from the *StateMachine* is applied to all items in the *HasNotifier* hierarchy below the object with which the *StateMachine* is associated. The *SystemState StateMachine* can be used to automatically disable, enable, suppress or un-suppress *Alarms* related to the Object (with in the hierarchy of alarms from the given object). The *StateMachine* can also be used by advanced alarming software to adjust the setpoint, limits or other items related to the *Alarms* in the hierarchy.

Optionally, multiple SystemState *StateMachines* can be deployed.



IEC

Figure F.1 – SystemState transitions

F.2 SystemStateStateMachineType

The *SystemStateStateMachineType* includes a hierarchy of substates. It supports multiple transitions between Operating, StartingUp, ShuttingDown, Shutdown, OutOfService and Maintenance.

The state machine is illustrated in Figure F.2 and formally defined in Table F.1.

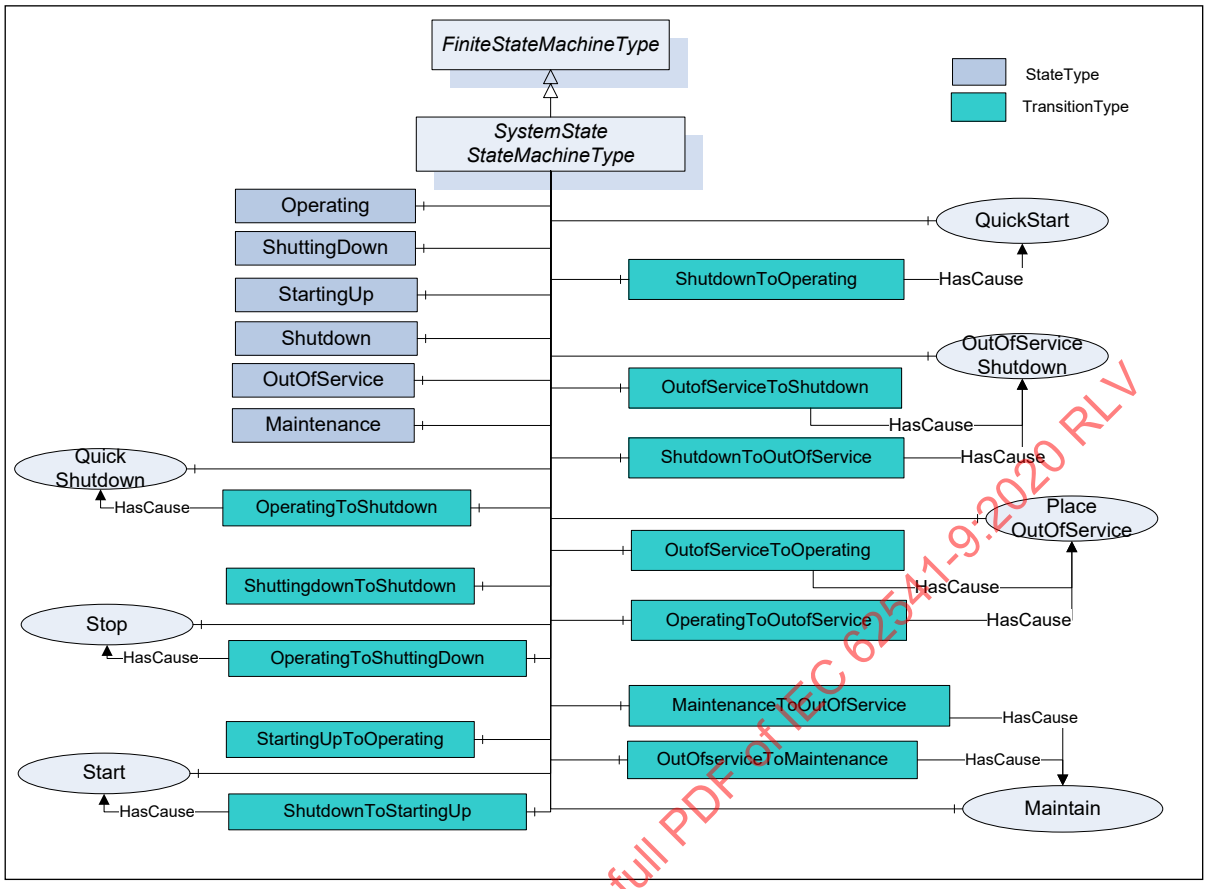


Figure F.2 – SystemStateStateMachineType Model

Table F.1 – SystemStateStateMachineType definition

Attribute	Value				
BrowseName	SystemStateStateMachineType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the Finite*StateMachineType* defined in IEC 62541-5					
HasComponent	Object	Operating		StateType	
HasComponent	Object	ShuttingDown		StateType	
HasComponent	Object	StartingUp		StateType	
HasComponent	Object	Shutdown		StateType	
HasComponent	Object	OutOfService		StateType	
HasComponent	Object	Maintenance		StateType	
HasComponent	Object	ShutdownToOperating		TransitionType	
HasComponent	Object	OperatingToShutdown		TransitionType	
HasComponent	Object	ShuttingdownToShutdown		TransitionType	
HasComponent	Object	OperatingToShuttingdown		TransitionType	
HasComponent	Object	StartingUpToOperating		TransitionType	
HasComponent	Object	ShutdownToStartingUp		TransitionType	
HasComponent	Object	OutofServiceToShutdown		TransitionType	
HasComponent	Object	ShutdownToOutOfService		TransitionType	
HasComponent	Object	OutofServiceToOperating		TransitionType	
HasComponent	Object	OperatingToOutofService		TransitionType	
HasComponent	Object	MaintenanceToOutOfService		TransitionType	
HasComponent	Object	OutOfServiceToMaintenance		TransitionType	
HasComponent	Method	Start	Defined in Clause XXX		Optional
HasComponent	Method	Maintain	Defined in Clause XXX		Optional
HasComponent	Method	Stop	Defined in Clause XXX		Optional
HasComponent	Method	PlaceOutOfservice	Defined in Clause XXX		Optional
HasComponent	Method	QuickShutdown	Defined in Clause XXX		Optional
HasComponent	Method	QuickStart	Defined in Clause XXX		Optional
HasComponent	Method	OutOfServiceShutdown	Defined in Clause XXX		Optional

The actual selection of *States* and *Transitions* would depend on the deployment of the *StateMachine*. If the *StateMachine* were being applied to a tank or other part of a process, it might have a different set of *States* than if it were applied to a meter or instrument. The meter may only have Operating, OutOfService and Maintenance, while the tank may have all of the described *States* and *Transitions*.

The *StateMachine* supports six possible states including: *Operating*, *ShuttingDown*, *StartingUp*, *Shutdown*, *OutOfService*, *Maintenance*. It supports 12 possible *Transitions* and 7 possible *Methods*.

The *SystemStateStateMachineType* transitions are formally defined in Table F.2.

Table F.2 – SystemStateStateMachineType transitions

BrowseName	References	BrowseName	TypeDefinition
Transitions			
ShutdownToOperating	FromState	Shutdown	StateType
	ToState	Operating	StateType
	HasCause	QuickStart	*Method*
OperatingToShutdown	FromState	Operating	StateType
	ToState	Shutdown	StateType
	HasCause	QuickShutdown	*Method*
ShuttingdownToShutdown	FromState	ShuttingDown	StateType
	ToState	Shutdown	StateType
OperatingToShuttingdown	FromState	Operating	StateType
	ToState	ShuttingDown	StateType
	HasCause	Stop	*Method*
StartingUpToOperating	FromState	StartingUp	StateType
	ToState	Operating	StateType
ShutdownToStartingUp	FromState	Shutdown	StateType
	ToState	StartingUp	StateType
	HasCause	Start	*Method*
OutofServiceToShutdown	FromState	OutOfService	StateType
	ToState	Shutdown	StateType
	HasCause	OutOfServiceShutdown	*Method*
ShutdownToOutOfService	FromState	Shutdown	StateType
	ToState	OutOfService	StateType
	HasCause	OutOfServiceShutdown	*Method*
OutOfServiceToOperating	FromState	OutOfService	StateType
	ToState	Operating	StateType
	HasCause	PlaceOutOfService	*Method*
OperatingToOutofService	FromState	Operating	StateType
	ToState	OutOfService	StateType
	HasCause	PlaceOutOfService	*Method*
MaintenanceToOutofService	FromState	Maintenance	StateType
	ToState	OutOfService	StateType
	HasCause	Maintain	*Method*
OutOfServiceToMaintenance	FromState	OutOfService	StateType
	ToState	Maintenance	StateType
	HasCause	Maintain	*Method*

The system can always generate additional *HasCause References*, such as internal code. No *HasEffect References* are defined, but an implementation might define *HasEffect References* (such as *HasEffectDisable*) for disabling or enabling *Alarms*, suppressing *Alarms* or adjusting setpoints or limits of *Alarms*. The targets of the reference might be an individual *Alarm* or portion of a plant or piece of equipment. See Clause 7 for a list of *HasEffect References* that could be used.

Bibliography

~~IEC 62541-7, *OPC Unified Architecture – Part 7: Profiles*~~

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

ISA 18.2, *Management of Alarm Systems for the Process Industries*
https://www.isa.org/store/ansi/isa-182-2016,-management-of-alarm-systems-for-the-process-industries/46962105

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 RFC 2047, *Multipurpose Internet Mail Extensions (MIME) Part Three*
https://www.ietf.org/rfc/rfc2047.txt

IEC 62541-9

Edition 3.0 2020-06

INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC unified architecture – Part 9: Alarms and Conditions

Architecture unifiée OPC – Partie 9: Alarmes et Conditions

IEC 62541-9:2020-06(en-fr)

CONTENTS

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 third edition cancels and replaces the second edition published in 2015. This edition constitutes a technical revision.

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

a) added optional engineering units to the definition of RateOfChange alarms;
b) to fulfill the IEC 62682 model, the following elements have been added:
   - AlarmConditionType States: Suppression, Silence, OutOfService, Latched;
   - AlarmConditionType Properties: OnDelay, OffDelay, FirstInGroup, ReAlarmTime;

- New alarm types: DiscrepancyAlarm, DeviationAlarm, InstrumentDiagnosticAlarm, SystemDiagnosticAlarm.

c) added Annex that specifies how the concepts of this OPC UA part maps to IEC 62682 and ISA 18.2;

d) added new ConditionClasses: Safety, HighlyManaged, Statistical, Testing, Training;

e) added CertificateExpiration AlarmType;

f) added Alarm Metrics model.

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

FDIS	Report on voting
65E/709/FDIS	65E/727/RVD

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 –

Part 9: Alarms and Conditions

1 Scope

This part of IEC 62541 specifies the representation of *Alarms* and *Conditions* in the OPC Unified Architecture. Included is the *Information Model* representation of *Alarms* and *Conditions* in the OPC UA address space. Other aspects of alarm systems such as alarm philosophy, life cycle, alarm response times, alarm types and many other details are captured in documents 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.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC TR 62541-1, *OPC unified architecture – Part 1: Overview and concepts*

IEC 62541-3, *OPC unified architecture – Part 3: Address Space Model*

IEC 62541-4, *OPC unified architecture – Part 4: Services*

IEC 62541-5, *OPC unified architecture – Part 5: Information Model*

IEC 62541-6, *OPC unified architecture – Part 6: Mappings*

IEC 62541-7, *OPC unified architecture – Part 7: Profiles*

IEC 62541-8, *OPC unified architecture – Part 8: Data Access*

IEC 62541-11, *OPC unified architecture – Part 11: Historical Access*

IEC 62682, *Management of alarms systems for the process industries*

EEMUA: 2nd Edition EEMUA 191 – *Alarm System – A guide to design, management and procurement (Appendixes 6, 7, 8, 9)*, available at https://www.eemua.org/Products/Publications/Print/EEMUA-Publication-191.aspx

3 Terms, definitions, abbreviated terms and data types used

3.1 Terms and definitions

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

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- IEC Electropedia: available at http://www.electropedia.org/
- ISO Online browsing platform: available at http://www.iso.org/obp

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 uses *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 can 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 specifications such as IEC 62682 or ISA 18.2.

3.1.5
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 *HasAlarmSuppressionGroup ReferenceType*.

3.1.6
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 can derive more concrete classes or define different top-level classes.

3.1.7
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.

3.1.8
ConditionSource
element which a specific *Condition* is based upon or related to

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

Note 2 to entry: In *Events* generated for *Conditions*, the *SourceNode Property* (inherited from the *BaseEventType*) will contain the *NodeId* of the *ConditionSource*.

3.1.9
confirm
Operator action informing the *Server* that a corrective action has been taken to address the cause of the *Alarm*

3.1.10
disable
action configurating a system 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.

Note 2 to entry: In IEC 62682, "disable" is referenced as "Out of Service".

3.1.11
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.

3.1.12
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 document, an Operator is a special user. All descriptions that apply to general users also apply to Operators.

3.1.13
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.

3.1.14
Retain
Alarm in a state that is interesting for a *Client* wishing to synchronize its state of *Conditions* with the *Server*'s state

3.1.15
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.

3.1.16
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 IEC62682 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 offline then low-level alarms are Suppressed for all equipment in the off-line unit.

3.2 Abbreviated terms

A&E Alarm & Event (as used for OPC COM)

COM (Microsoft Windows) Component Object Model

DA data access

UA Unified Architecture

3.3 Data types used

Table 1 and Table 2 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 document 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 which 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 subtypes of the *BaseEventType* (see IEC 62541-3 and IEC 62541-5). This part defines types that are common across many industries. It is expected that vendors or other standardisation groups will define additional *ConditionTypes* deriving from the common base types defined in this part. The *ConditionTypes* supported by a *Server* are exposed in the *AddressSpace* of the *Server*.

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

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

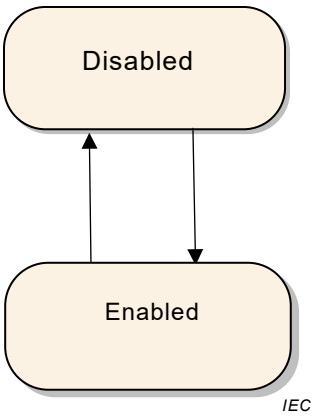


Figure 1 – Base Condition state model

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

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

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

4.3 Acknowledgeable Conditions

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

An *AckedState* and a *ConfirmedState* extend the *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*.

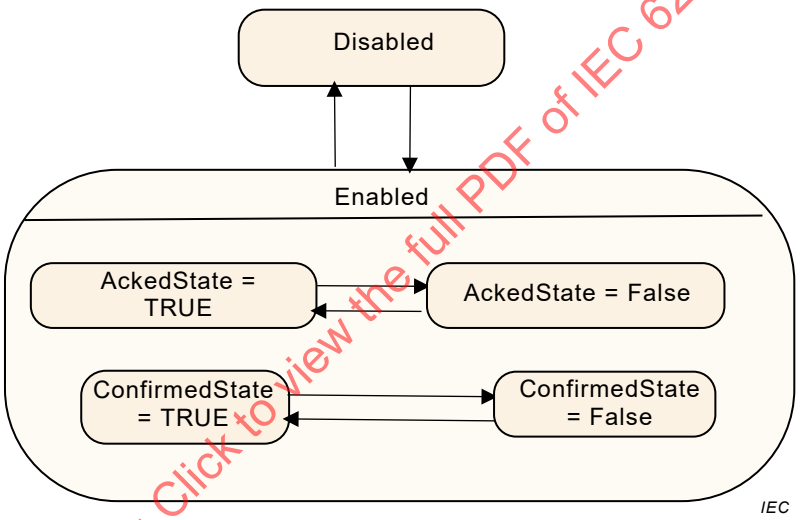


Figure 2 – AcknowledgeableConditions state model

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 document. Either of these state models can be extended to support more complex acknowledgement situations.

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

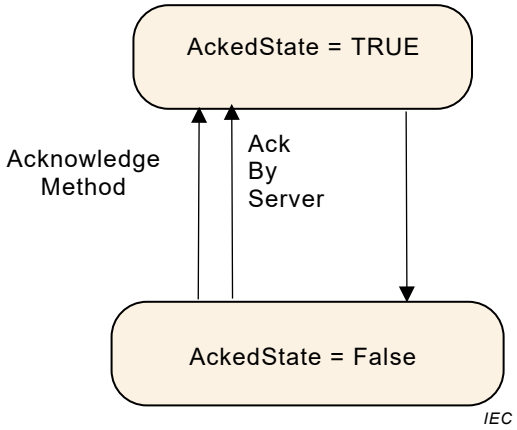


Figure 3 – Acknowledge state model

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

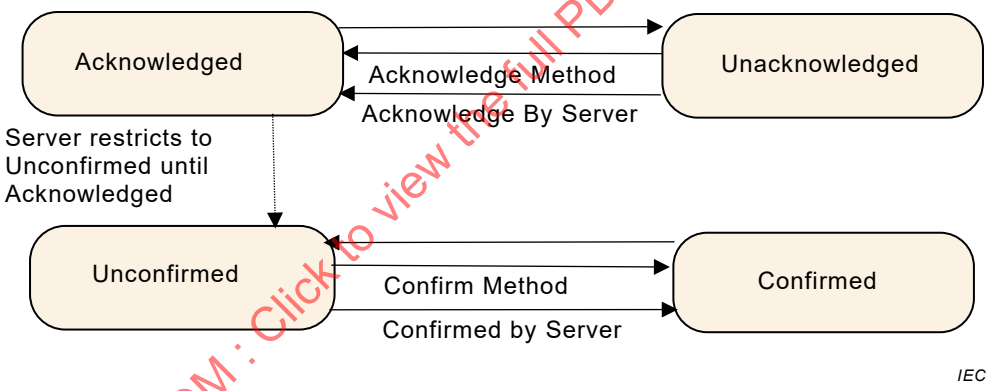


Figure 4 – Confirmed Acknowledge state model

4.4 Previous states of Conditions

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

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

4.5 Condition state synchronization

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

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

Clients request a *Refresh* by calling the ConditionRefresh *Method*. The *Server* will respond with a *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 subclause.

- 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 Notification*s.

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.

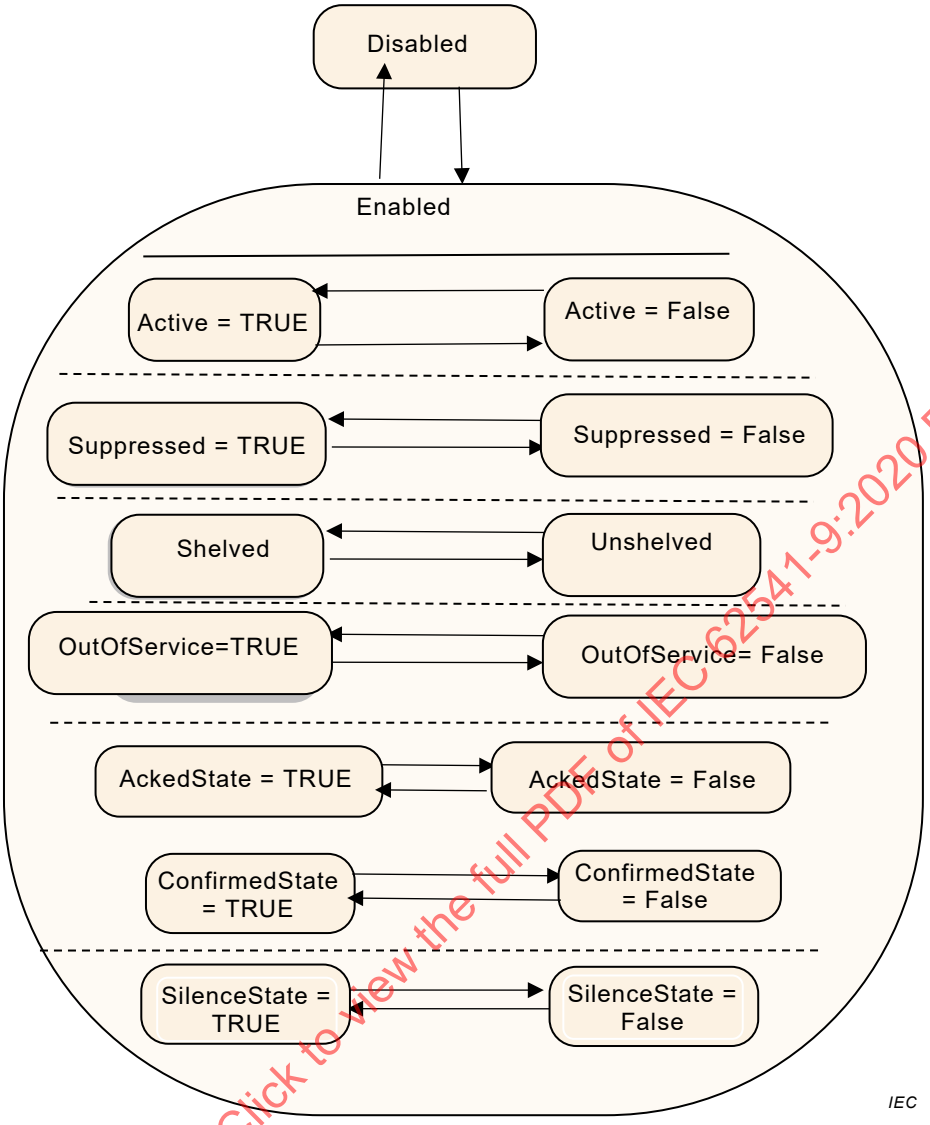


Figure 5 – Alarm state machine model

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

Some *Alarm* subtypes introduce substates 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 shelve features to help keep *Operators* from being overwhelmed during *Alarm* "storms" by limiting the number of *Alarms* an *Operator* sees on a current *Alarm* display. This is accomplished by setting the *SuppressedOrShelved* flag on second order dependent *Alarms* and/or *Alarms* of less severity, leading the *Operator* to concentrate on the most critical issues.

The shelved, 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*.

Alarms follow a typical timeline, which 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 *Operator*s about conditions in a timely manner and allow the *Operator* to take some action before some consequences occur. The consequences can be economic (product is not usable and shall be discarded), can be physical (tank overflows), can be safety related (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.

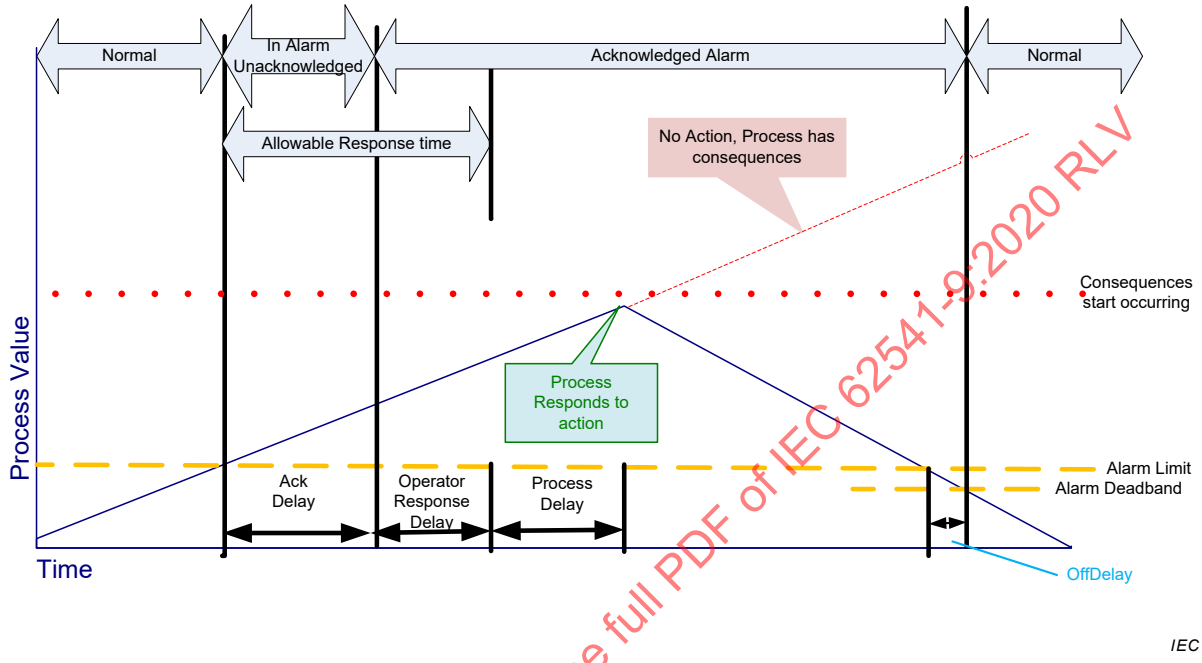


Figure 6 – Typical Alarm Timeline example

4.9 Multiple active states

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

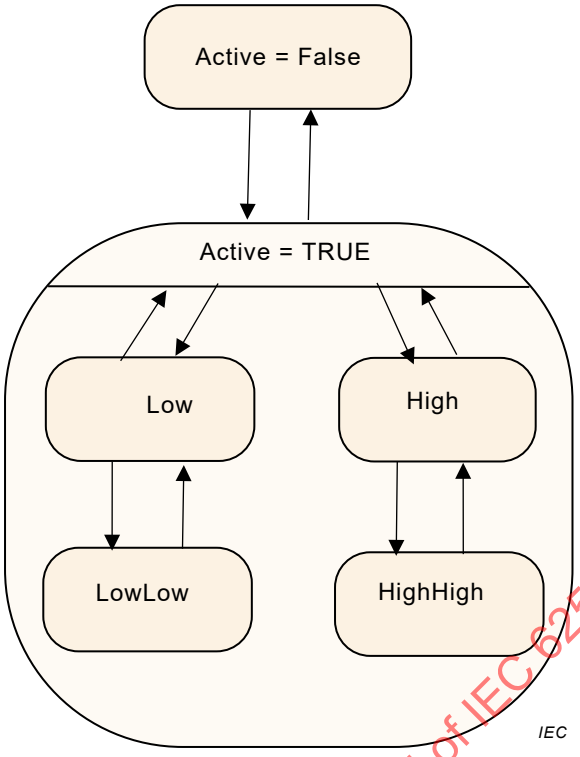


Figure 7 – Multiple active states example

With the multi-state *Alarm* model, state transitions among the substates 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 substate at a time. If for example a temperature exceeds the HighHigh limit the associated exclusive level *Alarm* will be in the HighHigh substate and not in the High substate.

Some *Alarm* systems, however, allow multiple substates 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 substate.

4.10 Condition instances in the AddressSpace

Because *Conditions* always have a state (*Enabled* or *Disabled*) and possibly many substates 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 IEC 62541 series includes 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 document defines a number of subtypes of the *AuditEventTypes* but without adding new fields to them.

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

5 Model

5.1 General

The *Alarm* and *Condition* model extends the OPC UA base *Event* model by defining various *Event Types* based on the *BaseEventType*. All of the *Event Types* defined in this document 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*.

Subclauses 5.5 to 5.8 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. The full *AlarmConditionType* hierarchy can be found in Figure 8.

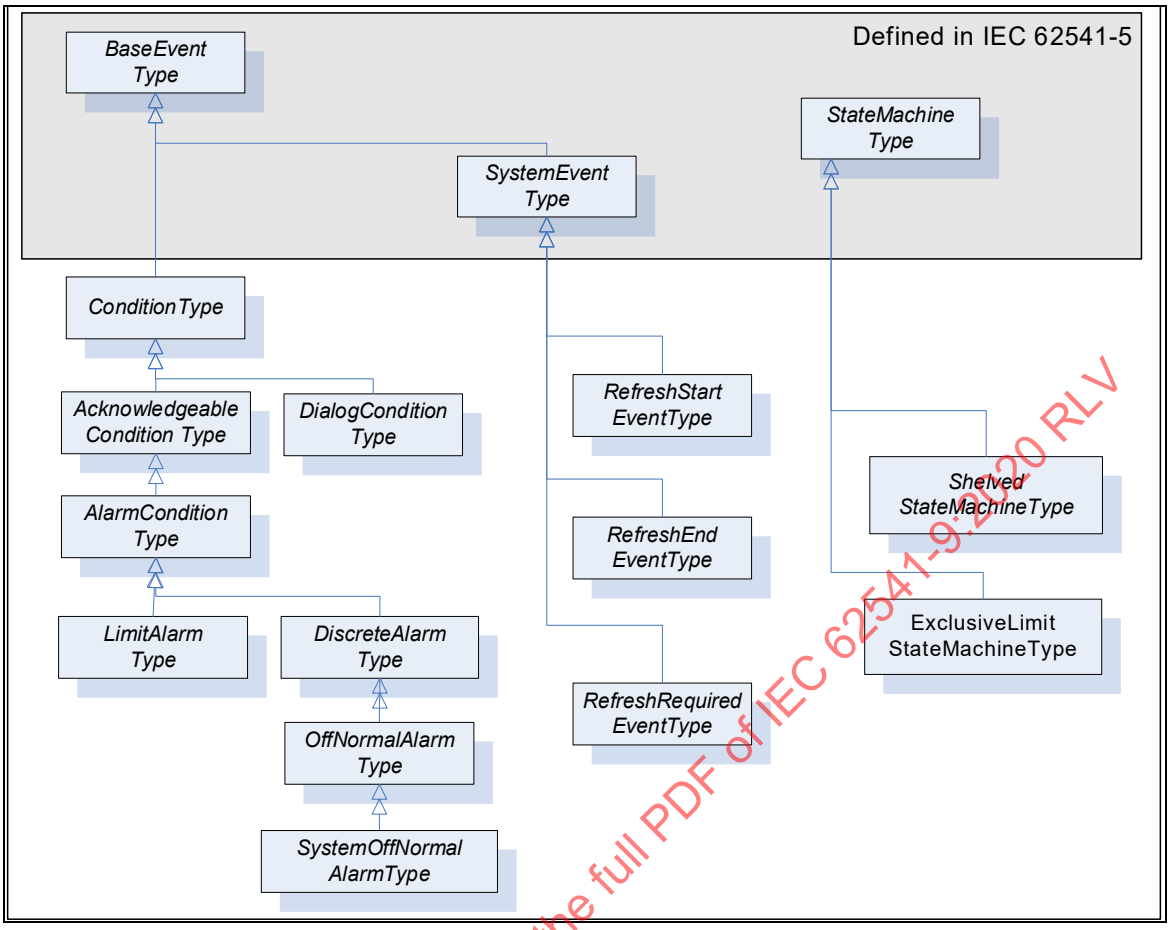


Figure 8 – ConditionType hierarchy

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 and the model described in this document.

5.2 Two-state state machines

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

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

Table 3 – TwoStateVariableType definition

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

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

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

TransitionTime specifies the time when the current state was entered.

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

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

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

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

A *HasTrueSubState Reference* is used to indicate that the True state has substates.

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

5.3 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 *BaseDataVariableType* defined in IEC 62541-5.					
HasProperty	Variable	SourceTimestamp	UtcTime	PropertyType	Mandatory

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

5.4 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 substates and the addition of *Alarm* grouping.

The *TwoStateVariableType References* will only exist when substates are available. For example, if a *TwoStateVariableType* machine is in a False State, then any substates referenced from the True state will not be available. If an Event is generated while in the False state and information from the True state substate is part of the data that is to be reported than this data would be reported as a NULL. With this approach, *TwoStateVariableTypes* can be extended with subordinate state machines in a similar fashion to the *StateMachineType* defined in IEC 62541-5.

5.4.2 HasTrueSubState ReferenceType

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

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

The SourceNode of the Reference shall be an instance of a TwoStateVariableType and the TargetNode shall 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 substate, but it is required that the substate provides the inverse Reference (IsTrueSubStateOf) to its super state.

Table 5 – HasTrueSubState ReferenceType

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

5.4.3 HasFalseSubState ReferenceType

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

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

The SourceNode of the Reference shall be an instance of a TwoStateVariableType and the TargetNode shall 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 substate, but it is required that the substate provides the inverse Reference (IsFalseSubStateOf) to its super state.

Table 6 – HasFalseSubState ReferenceType

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

5.4.4 HasAlarmSuppressionGroup ReferenceType

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

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

The *SourceNode* of the *Reference* shall be an instance of an *AlarmConditionType* or subtype and 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**

5.4.5 AlarmGroupMember ReferenceType

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

This *ReferenceType* is used to indicate the *Alarm* instances that are part of an *Alarm Group*.

The *SourceNode* of the *Reference* shall be an instance of an *AlarmGroupType* or subtype of it and 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**

5.5 Condition Model

5.5.1 General

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

The *Condition* model is illustrated in Figure 9 and formally defined in the subsequent tables. It is worth noting that this figure, like all figures in this document, is not intended to be complete. Rather, the figures only illustrate information provided by the formal definitions.

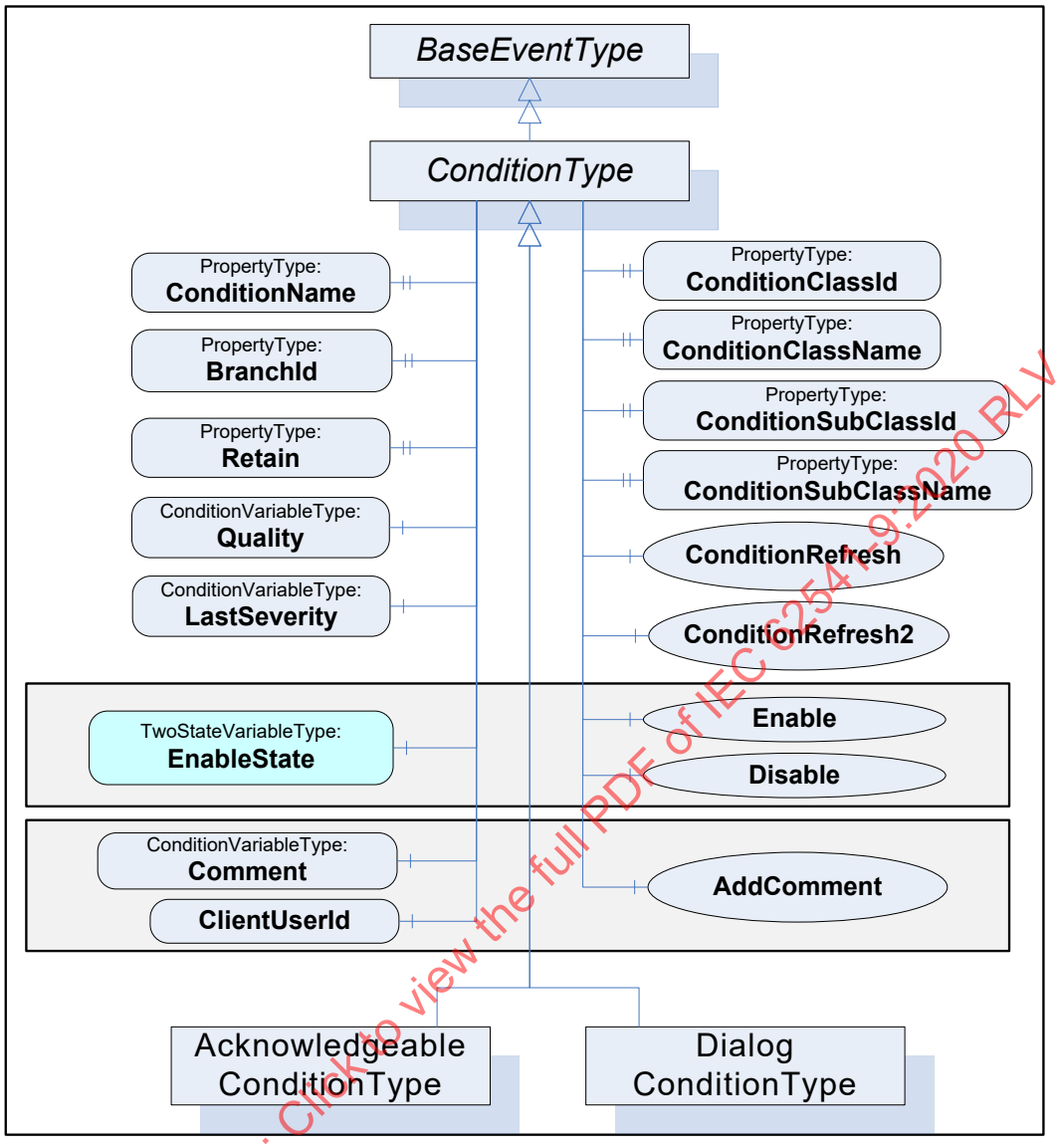


Figure 9 – Condition model

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 9. The False state of the *EnabledState* shall not be extended with a substate machine.

Table 9 – ConditionType definition

Attribute	Value				
BrowseName	ConditionType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseEventType* defined in IEC 62541-5					
HasSubtype	ObjectType	DialogConditionType	Defined in 5.6.2		
HasSubtype	ObjectType	AcknowledgeableConditionType	Defined in 5.7.2		
HasProperty	Variable	ConditionClassId	NodeId	PropertyType	Mandatory
HasProperty	Variable	ConditionClassName	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	ConditionSubClassId	NodeId[]	PropertyType	Optional
HasProperty	Variable	ConditionSubClassName	LocalizedText[]	PropertyType	Optional
HasProperty	Variable	ConditionName	String	PropertyType	Mandatory
HasProperty	Variable	BranchId	NodeId	PropertyType	Mandatory
HasProperty	Variable	Retain	Boolean	PropertyType	Mandatory
HasComponent	Variable	EnabledState	LocalizedText	TwoStateVariableType	Mandatory
HasComponent	Variable	Quality	StatusCode	ConditionVariableType	Mandatory
HasComponent	Variable	LastSeverity	UInt16	ConditionVariableType	Mandatory
HasComponent	Variable	Comment	LocalizedText	ConditionVariableType	Mandatory
HasProperty	Variable	ClientUserId	String	PropertyType	Mandatory
HasComponent	Method	Disable	Defined in 5.5.4		Mandatory
HasComponent	Method	Enable	Defined in 5.5.5		Mandatory
HasComponent	Method	AddComment	Defined in 5.5.6		Mandatory
HasComponent	Method	ConditionRefresh	Defined in 5.5.7		None
HasComponent	Method	ConditionRefresh2	Defined in 5.5.8		None

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* which are related to the motor as well as a temperature sensor associated with the motor. In the former, the *InputNode* for the High RPM *Alarm* is the value of the Motor RPM, while in the later the *InputNode* of the High Alarm would be the value of the temperature sensor that is associated with the motor.

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

ConditionClassName provides the display name of the subtype of *BaseConditionClassType*.

ConditionSubClassId specifies additional class[es] that apply to the *Condition*. It is the *NodeId* of the corresponding subtype of *BaseConditionClassType*. See 5.9.6 for the definition of *ConditionClass* and a set of *ConditionClasses* defined in this document. When using this *Property* for filtering, *Clients* shall specify all individual subtypes of *BaseConditionClassType NodeIds*. The OfType operator cannot be applied. The *Client* specifies a NULL in the filter, to return *Conditions* where no sub class is applied. When returning *Conditions*, if this optional field is not available in a *Condition*, a NULL shall be returned for the field.

ConditionSubClassName provides the display name[s] of the *ConditionClassType[s]* listed in the ConditionSubClassId.

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

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

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

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

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

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

- When the *Condition* instance enters the *Disabled* state, the *Retain Property* of this *Condition* shall be set to False by the *Server* to indicate to the *Client* that the *Condition* instance is currently not of interest to *Client*s. This includes all *ConditionBranches* if any branches exist.
- When the *Condition* instance enters the enabled state, the *Condition* shall be evaluated and all of its *Properties* updated to reflect the current values. If this evaluation causes the *Retain Property* to transition to True for any *ConditionBranch*, then an *Event Notification* shall be generated for that *ConditionBranch*.
- The *Server* may choose to continue to test for a *Condition* instance while it is *Disable*d. However, no *Event Notifications* will be generated while the *Condition* instance is disabled.

- For any *Condition* that exists in the *AddressSpace* the *Attributes* and the following *Variables* will continue to have valid values even in the *Disable*d state; EventId, *Event* Type, Source *Node*, Source Name, Time, and EnabledState. Other *Properties* may no longer provide current valid values. All *Variables* that are no longer provided shall return a status of Bad_ConditionDisabled. The Event that reports the Disabled state should report the *Properties* as NULL or with a status of Bad_ConditionDisabled.

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

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

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

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

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

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

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

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

The *NodeId* of the *Condition* instance is used as *ConditionId*. It is not explicitly modelled as a component of the *ConditionType*. However, it can be requested with the following *SimpleAttributeOperand* (see Table 10) in the *SelectClause* of the *EventFilter*:

Table 10 – SimpleAttributeOperand

Name	Type	Description
SimpleAttributeOperand		
typeId	NodeId	*NodeId* of the *ConditionType Node*
browsePath[]	QualifiedName	empty
attributeId	IntegerId	Id of the *NodeId Attribute*

5.5.3 Condition and branch instances

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

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

The state of a *Condition* instance at any given time is the set values for the *Variable*s 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 document 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*.

Signature

```
Disable();
```

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

Table 11 – Disable result codes

Result Code	Description
Bad_ConditionAlreadyDisabled	See Table 101 for the description of this result code.

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

Table 12 – Disable Method AddressSpace definition

Attribute	Value				
BrowseName	Disable				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionEnableEventType	Defined in 5.10.2		

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.*

Signature

```
Enable();
```

Method result codes in Table 13 (defined in Call Service).

Table 13 – Enable result codes

Result Code	Description
Bad_ConditionAlreadyEnabled	See Table 101 for the description of this result code.

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

Table 14 – Enable Method AddressSpace definition

Attribute	Value				
BrowseName	Enable				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionEnableEventType	Defined in 5.10.2		

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
    [in] LocalizedText Comment
    );
```

The parameters are defined in Table 15.

Table 15 – AddComment arguments

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

Method result codes in Table 16 (defined in Call Service).

Table 16 – AddComment result codes

Result Code	Description
Bad_MethodInvalid	The *MethodId* provided does not correspond to the *ObjectId* provided. See IEC 62541-4 for the general description of this result code.
Bad_EventIdUnknown	See Table 101 for the description of this result code.
Bad_NodeIdInvalid	Used to indicate that the specified *ObjectId* is not valid or that the *Method* was called on the ConditionType *Node*. See IEC 62541-4 for the general description of this result code.

Comments

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

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

Table 17 specifies the *AddressSpace* representation for the *AddComment Method*.

Table 17 – AddComment Method AddressSpace definition

Attribute	Value				
BrowseName	AddComment				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	*Variable*	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGeneratesEvent	ObjectType	AuditConditionCommentEventType	Defined in 5.10.4		

5.5.7 ConditionRefresh Method

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

Signature

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

The parameters are defined in Table 18.

Table 18 – ConditionRefresh parameters

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

Method result codes in Table 19 (defined in Call Service).

Table 19 – ConditionRefresh result codes

Result Code	Description
Bad_SubscriptionIdInvalid	See IEC 62541-4 for the description of this result code
Bad_RefreshInProgress	See Table 101 for the description of this result code
Bad_UserAccessDenied	The *Method* was not called in the context of the *Session* that owns the *Subscription* See IEC 62541-4 for the general description of this result code.

Comments

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

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

1) The *Server* issues an event of *RefreshStartEvent*Type(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* might 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 20 specifies the *AddressSpace* representation for the *ConditionRefresh Method*.

Table 20 – ConditionRefresh Method AddressSpace definition

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

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* or its subtypes. To invoke this *Method*, the call shall pass the well-known *MethodId* of the *Method* on the *ConditionType* and the *ObjectId* shall be the well-known *ObjectId* of the *ConditionType Object*.

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 21.

Table 21 – ConditionRefresh2 parameters

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

Method result codes in Table 22 (defined in Call Service).

Table 22 – 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 101 for the description of this result code
Bad_UserAccessDenied	The *Method* was not called in the context of the Session that owns the *Subscription.* 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

Subclause 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* which 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 23 specifies the *AddressSpace* representation for the *ConditionRefresh2 Method*.

Table 23 – ConditionRefresh2 Method AddressSpace definition

Attribute	Value				
BrowseName	ConditionRefresh2				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	*Variable*	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGeneratesEvent	ObjectType	RefreshStartEvent	Defined in 5.11.2		
AlwaysGeneratesEvent	ObjectType	RefreshEndEvent	Defined in 5.11.3		

5.6 Dialog Model

5.6.1 General

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

5.6.2 DialogConditionType

The *DialogConditionType* is used to represent *Conditions* as dialogs. It is illustrated in Figure 10 and formally defined in Table 24.

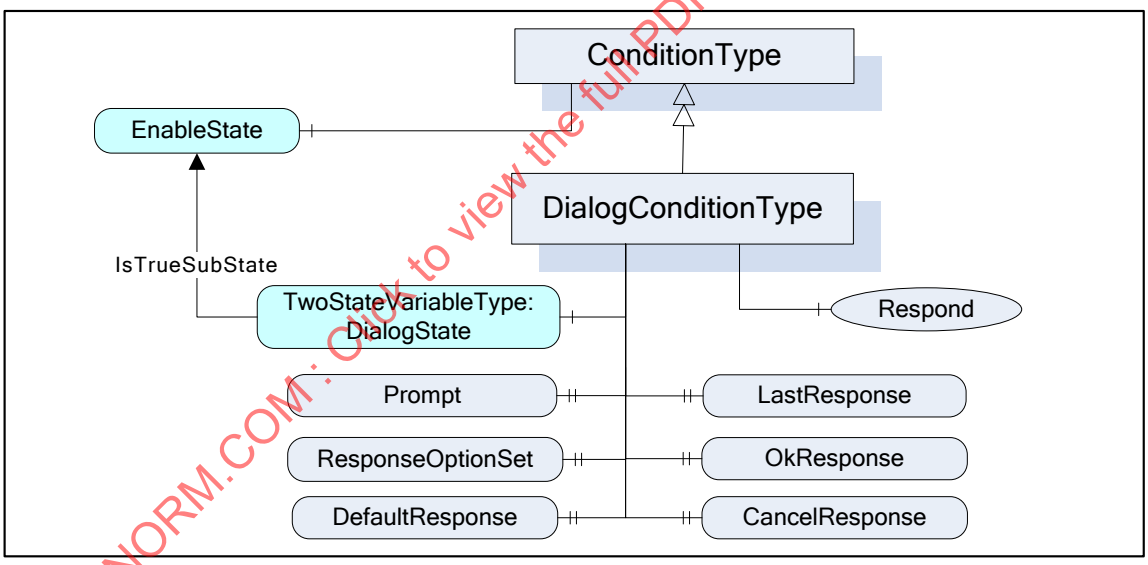


Figure 10 – DialogConditionType overview

Table 24 – DialogConditionType definition

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

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

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

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

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

Typical combinations of response options are

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

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

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

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

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

5.6.3 Respond Method

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

Signature

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

The parameters are defined in Table 25.

Table 25 – Respond parameters

Argument	Description
SelectedResponse	Selected index of the *ResponseOptionSet* array.

Method result codes in Table 26 (defined in Call Service).

Table 26 – Respond Result Codes

Result Code	Description
Bad_DialogNotActive	See Table 101 for the description of this result code.
Bad_DialogResponseInvalid	See Table 101 for the description of this result code.

Table 27 specifies the *AddressSpace* representation for the *Respond Method*.

Table 27 – Respond Method AddressSpace definition

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

5.7 Acknowledgeable Condition Model

5.7.1 General

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

AcknowledgeableConditions are represented by the *AcknowledgeableConditionType* which is a subtype of the *ConditionType*. The model is formally defined in 5.7.2 to 5.7.4.

5.7.2 AcknowledgeableConditionType

The *AcknowledgeableConditionType* extends the *ConditionType* by defining acknowledgement characteristics. It is an abstract type. The *AcknowledgeableConditionType* is illustrated in Figure 11 and formally defined in Table 28.

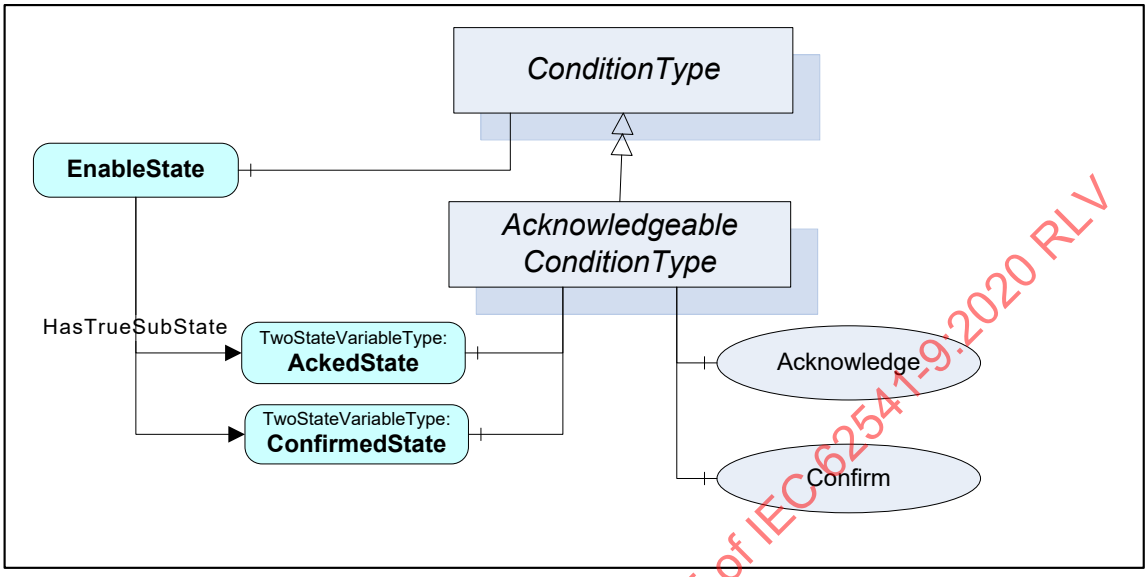


Figure 11 – AcknowledgeableConditionType overview

Table 28 – AcknowledgeableConditionType definition

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

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

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

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

5.7.3 Acknowledge Method

The *Acknowledge Method* is used to acknowledge an *Event Notification* for a *Condition* instance state where *AckedState* is False. 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, *Servers* shall allow *Clients* to call the *Acknowledge Method* by specifying *ConditionId* as the *ObjectId.* The *Method* cannot be called with an *ObjectId* of the *AcknowledgeableConditionType Node*.

Signature

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

The parameters are defined in Table 29.

Table 29 – Acknowledge parameters

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

Method result codes in Table 30 (defined in Call Service).

Table 30 – Acknowledge result codes

Result Code	Description
Bad_ConditionBranchAlreadyAcked	See Table 101 for the description of this result code.
Bad_MethodInvalid	The method id does not refer to a method for the specified object or ConditionId.
Bad_EventIdUnknown	See Table 101 for the description of this result code.
Bad_NodeIdInvalid	Used to indicate that the specified *ObjectId* is not valid or that the *Method* was called on the ConditionType *Node*. See IEC 62541-4 for the general description of this result code.

Comments

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

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

A valid *EventId* will result in an *Event Notification* where *AckedState*/Id is set to True and the *Comment Property* contains the text of the optional comment argument. If a previous state is acknowledged, the *BranchId* and all *Condition* values of this branch will be reported. Table 31 specifies the *AddressSpace* representation for the *Acknowledge Method*.

Table 31 – Acknowledge Method AddressSpace definition

Attribute	Value				
BrowseName	Acknowledge				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	*Variable*	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGenerates Event	ObjectType	AuditConditionAcknowledge EventType	Defined in 5.10.5		

5.7.4 Confirm Method

The *Confirm Method* is used to confirm an *Event Notifications* for a *Condition* instance state where *ConfirmedState* is False. 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, *Servers* shall allow *Clients* to call the *Confirm Method* by specifying *ConditionId* as the *ObjectId*. The *Method* cannot be called with an *ObjectId* of the *AcknowledgeableConditionType Node*.

Signature

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

The parameters are defined in Table 32.

Table 32 – Confirm Method parameters

Argument	Description
EventId	*EventId* identifying a particular *Event Notification*. Only *Event Notifications* where the Id property of the *ConfirmedState* is False can be confirmed.
Comment	A localized text to be applied to the *Conditions*.

Method result codes in Table 33 (defined in Call Service).

Table 33 – Confirm result codes

Result Code	Description
Bad_ConditionBranchAlreadyConfirmed	See Table 101 for the description of this result code.
Bad_MethodInvalid	The method id does not refer to a method for the specified object or *ConditionId*. See IEC 62541-4 for the general description of this result code.
Bad_EventIdUnknown	See Table 101 for the description of this result code.
Bad_NodeIdUnknown	Used to indicate that the specified *ObjectId* is not valid or that the *Method* was called on the ConditionType *Node*. See IEC 62541-4 for the general description of this result code.

Comments

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

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

A valid *EventId* will result in an *Event Notification* where *ConfirmedState*/Id is set to True and the *Comment Property* contains the text of the optional comment argument. If a previous state is confirmed, the *BranchId* and all *Condition* values of this branch will be reported. A *Client* can confirm only events that have a *ConfirmedState*/Id set to False. The logic for setting *ConfirmedState*/Id to False is *Server* specific and may even be event or condition specific.

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

Table 34 – Confirm Method AddressSpace definition

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

5.8 Alarm model

5.8.1 General

Figure 12 informally describes the AlarmConditionType, its subtypes and where it is in the hierarchy of *Event Types*.

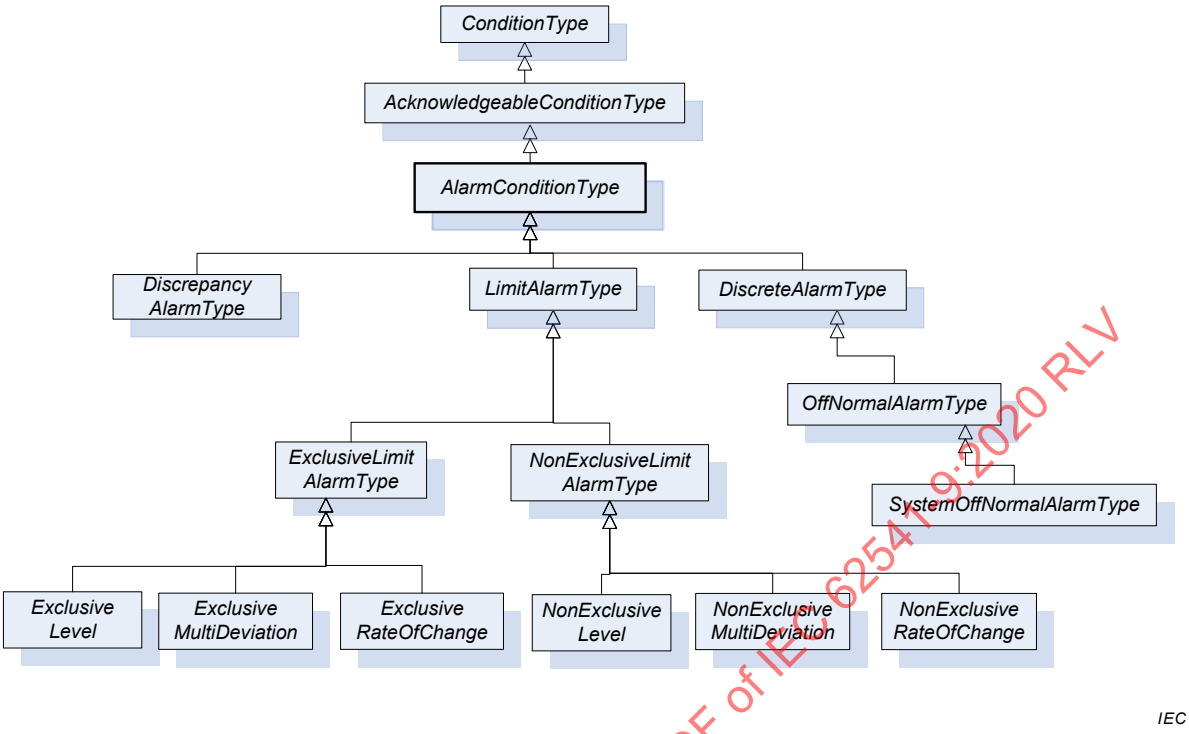


Figure 12 – AlarmConditionType Hierarchy Model

5.8.2 AlarmConditionType

The *AlarmConditionType* is an abstract type that extends the *AcknowledgeableConditionType* by introducing an *ActiveState*, *SuppressedState* and *ShelvingState*. It also adds the ability to set a delay time, re-alarm time, *Alarm* groups and audible *Alarm* settings The *Alarm* model is illustrated in Figure 13. This illustration is not intended to be a complete definition. It is formally defined in Table 35.

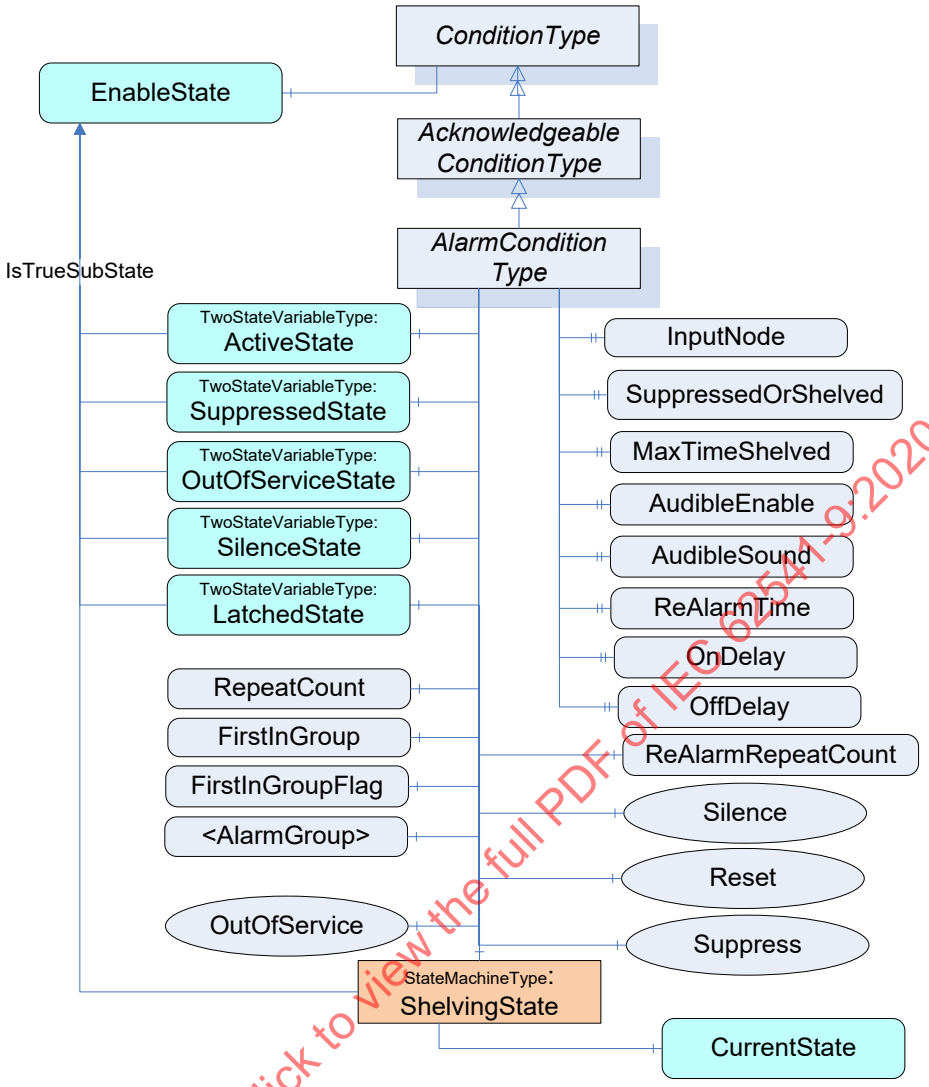


Figure 13 – Alarm Model

Table 35 – AlarmConditionType definition

Attribute	Value				
BrowseName	AlarmConditionType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the AcknowledgeableConditionType defined in clause 5.7.2					
HasComponent	Variable	ActiveState	LocalizedText	TwoStateVariableType	Mandatory
HasProperty	Variable	InputNode	NodeId	PropertyType	Mandatory
HasComponent	Variable	SuppressedState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Variable	OutOfServiceState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Object	ShelvingState		ShelvedStateMachineType	Optional
HasProperty	Variable	SuppressedOrShelved	Boolean	PropertyType	Mandatory
HasProperty	Variable	MaxTimeShelved	Duration	PropertyType	Optional
HasProperty	Variable	AudibleEnabled	Boolean	PropertyType	Optional
HasComponent	Variable	AudibleSound	AudioDataType	AudioVariableType	Optional
HasComponent	Variable	SilenceState	LocalizedText	TwoStateVariableType	Optional
HasProperty	Variable	OnDelay	Duration	PropertyType	Optional
HasProperty	Variable	OffDelay	Duration	PropertyType	Optional
HasComponent	Variable	FirstInGroupFlag	Boolean	BaseDataVariableType	Optional
HasComponent	Object	FirstInGroup		AlarmGroupType	Optional
HasComponent	Object	LatchedState	LocalizedText	TwoStateVariableType	Optional
HasAlarmSuppressionGroup	Object	<AlarmGroup>		AlarmGroupType	OptionalPlaceholder
HasProperty	Variable	ReAlarmTime	Duration	PropertyType	Optional
HasComponent	Variable	ReAlarmRepeatCount	Int16	BaseDataVariableType	Optional
HasComponent	Method	Silence	Defined in 5.8.5		Optional
HasComponent	Method	Suppress	Defined in 5.8.6		Optional
HasComponent	Method	Unsuppress	Defined in 5.8.7		Optional
HasComponent	Method	RemoveFromService	Defined in 5.8.8		Optional
HasComponent	Method	PlaceInService	Defined in 5.8.9		Optional
HasComponent	Method	Reset	Defined in 5.8.4		Optional
HasSubtype	Object	DiscreteAlarmType			
HasSubtype	Object	LimitAlarmType			
HasSubtype	Object	DiscrepancyAlarmType			

The *AlarmConditionType* inherits all *Properties* of the *AcknowledgeableConditionType*. The following states are substates of the True *EnabledState*.

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

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

SuppressedState, *OutOfServiceState* and *ShelvingState* together allow the suppression of *Alarms* on display systems. These three suppressions are generally used by different personnel or systems at a plant, i.e. automatic systems, maintenance personnel and *Operators*.

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

OutOfServiceState is used by maintenance personnel to suppress *Alarms* due to a maintenance issue. For example, if an instrument is taken out of service for maintenance or is removed temporarily while it is being replaced or serviced, the item would have the *OutOfServiceState* set. Recommended state names are described in Annex A.

ShelvingState suggests whether an *Alarm* shall (temporarily) be prevented from being displayed to the user. It is quite often used by *Operators* to block nuisance *Alarms*. The *ShelvingState* is defined in 5.8.10.

When an *Alarm* has any or all of the *SuppressedState, OutOfServiceState* or *ShelvingState* set to True, the *SuppressedOrShelved* property shall be set True and this *Alarm* is then typically not displayed by the *Client*. State transitions associated with the *Alarm* do occur, but they are not typically displayed by the *Client*s as long as the *Alarm* remains in any of the *SuppressedState, OutOfServiceState* or *Shelved* state.

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

The optional *Property AudibleEnabled* is a Boolean that indicates if the current state of this *Alarm* includes an audible *Alarm*.

The optional *Property AudibleSound* contains the sound file that is to be played if an audible *Alarm* is to be generated. This file would be play/generated as long as the *Alarm* is active and unacknowledged, unless the silence *StateMachine* is included, in which case it may also be silenced by this *StateMachine*.

The *SilenceState* is used to suppress the generation of audible *Alarms*. Typically, it is used when an *Operator* silences all *Alarms* on a screen, but needs to acknowledge the *Alarms* individually. Silencing an *Alarm* shall silence the *Alarm* on all systems (screens) that it is being reported on. Not all *Clients* will make use of this *StateMachine*, but it allows multiple *Clients* to synchronize audible *Alarm* states. Acknowledging an *Alarm* shall automatically silence an *Alarm*.

The *OnDelay* and *OffDelay Properties* can be used to eliminate nuisance *Alarms*. The *OnDelay* is used to avoid unnecessary Alarms when a signal temporarily overshoots its setpoint, thus preventing the *Alarm* from being triggered until the signal remains in the *Alarm* state continuously for a specified length of time (*OnDelay* time). The *OffDelay* is used to reduce chattering *Alarms* by locking the *Alarm* indication for a certain holding period after the condition has returned to normal, i.e. the *Alarm* shall stay active for the *OffDelay* time and shall not regenerate if it returns to active in that period. If the *Alarm* remains in the inactive zone for *OffDelay*, it will then become inactive.

The optional variable *FirstInGroupFlag* is used together with the *FirstInGroup* object. The FirstInGroup Object is an instance of an AlarmGroupType that groups a number of related *Alarm*s. The *FirstInGroupFlag* is set on the *Alarm* instance that was the first *Alarm* to trigger in a *FirstInGroup*. If this variable is present, then the *FirstInGroup* shall also be present. These two nodes allow an alarming system to determine which *Alarm* in the list was the trigger. It is commonly used in situations where *Alarm*s are interrelated, and usually multiple *Alarm*s occur. For example, usually all vibration sensors in a turbine trigger if any one of them triggers, but what is important for an *Operator* is the first sensor that triggered.

The *LatchedState Object*, if present, indicates that this *Alarm* supports being latched. The *Alarm* will remain with a retain bit of True until it is no longer active, is acknowledge and is reset. The *Reset Method,* if called while active has no effect on the *Alarm* and is ignored and an error of Bad_InvalidState is return on the call. The *Object* indicates the current state, latched or not latched. Recommended state names are described in Annex A. If this *Object* is provided, the *Reset Method* shall also be provided.

An *Alarm* instance may contain HasAlarmSuppressionGroup reference(s) to instance(s) of *AlarmGroupType*. Each instance is an *AlarmSuppressionGroup.* When an *AlarmSuppressionGroup* goes active, the *Server* shall set the *SuppressedState* of the *Alarm* to True. When all of referenced *AlarmSuppressionGroups* are no longer active, then the Server shall set *SuppressedState* to False. A single *AlarmSuppressionGroup* can be assigned to multiple *Alarms*. *AlarmSuppressionGroups* are used to control *AlarmFloods* and to help manage *Alarms*.

ReAlarmTime if present sets a time that is used to bring an *Alarm* back to the top of an *Alarm* list. If an *Alarm* has not returned to normal within the provided time (from when it last was alarmed), the *Server* will generate a new *Alarm* for it (as if it just went into alarm). If it has been silenced it shall return to an un-silenced state, if it has been acknowledged it shall return to unacknowledged. The *Alarm* active time is set to the time of the re-alarm.

ReAlarmRepeatCount, if present, counts the number times an *Alarm* was re-alarmed. Some smart alarming system would use this count to raise the priority or otherwise generate additional or different annunciations for the given *Alarm*. The count is reset when an *Alarm* returns to normal.

Silence Method may be used to silence an instance of an *Alarm*. It is defined in 5.8.5.

Suppress Method may be used to suppress an instance of an *Alarm*. Most *Alarm* suppression occurs via advanced alarming, but this method allows additional access to suppress a particular *Alarm* instance. Additional details are provided in the definition in 5.8.6.

Unsuppress Method may be used to remove an instance of an *Alarm* from SuppressedState. Additional details are provided in the definition in 5.8.7.

PlaceInService Method may be used to remove an instance of an *Alarm* from *OutOfServiceState*. It is defined in 5.8.9.

RemoveFromService Method may be used to place an instance of an *Alarm* in *OutOfServiceState*. It is defined in 5.8.8.

Reset Method is used to clear a latched *Alarm*. It is defined in 5.8.4. If this *Object* is provided, the *LatchedState Object* shall also be provided.

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

5.8.3 AlarmGroupType

The *AlarmGroupType* provides a simple manner of grouping *Alarms*. This grouping may be used for *Alarm* suppression or for identifying related *Alarms*. The actual usage of the *AlarmGroupType* is specified where it is used.

The *AlarmGroupType* is formally defined in Table 36.

Table 36 – AlarmGroupType definition

Attribute	Value				
BrowseName	AlarmGroupType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the FolderType defined in IEC 62541-5.					
AlarmGroupMember	Object	<AlarmConditionInstance>		AlarmConditionType	OptionalPlaceholder

The instance of an *AlarmGroupType* should be given a name and description that describes the purpose of the *Alarm* group.

The *AlarmGroupType* instance will contain a list of instances of *AlarmConditionType* or subtype of *AlarmConditionType* referenced by *AlarmGroupMember* references. At least one *Alarm* shall be present in an instance of an *AlarmGroupType*.

5.8.4 Reset Method

The *Reset Method* is used reset a latched *Alarm* instance. It is only available on an instance of an *AlarmConditionType* that exposes the *LatchedState*. 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, *Servers* shall allow *Clients* to call the *Reset Method* by specifying *ConditionId* as the *ObjectId*. The *Method* cannot be called with an *ObjectId* of the *AlarmConditionType Node*.

Signature

```
Reset();
```

Method result codes are given in Table 37 (defined in *Call* service).

Table 37 – Silence result codes

Result Code	Description
Bad_MethodInvalid	The *MethodId* provided does not correspond to the *ObjectId* provided. See IEC 62541-4 for the general description of this result code.
Bad_NodeIdInvalid	Used to indicate that the specified *ObjectId* is not valid or that the *Method* was called on the *ConditionType Node*. See IEC 62541-4 for the general description of this result code.
Bad_InvalidState	The *Alarm* instance was not latched or still active or still required acknowledgement. For an *Alarm* Instance to be reset, it must have been in *Alarm*, and returned to normal and have been acknowledged prior to being reset.

Table 38 specifies the *AddressSpace* representation for the *Reset Method*.

Table 38 – Reset Method AddressSpace definition

Attribute	Value				
BrowseName	Reset				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	Ob jectType	AuditConditionResetEventType	Defined in 5.10.11		

5.8.5 Silence Method

The *Silence Method* is used to silence a specific *Alarm* instance. It is only available on an instance of an *AlarmConditionType* that also exposes the *SilenceState*. 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, *Servers* shall allow *Clients* to call the *Silence Method* by specifying *ConditionId* as the *ObjectId*. The *Method* cannot be called with an *ObjectId* of the *AlarmConditionType Node*.

Signature

```
Silence();
```

Method result codes in Table 39 (defined in *Call* service).

Table 39 – Silence result codes

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

Comments

If the instance is not currently in an audible state, the command is ignored.

Table 40 specifies the *AddressSpace* representation for the *Silence Method*.

Table 40 – Silence Method AddressSpace definition

Attribute	Value				
BrowseName	Silence				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
AlwaysGeneratesEvent	ObjectType	AuditConditionSilenceEventType	Defined in 5.10.10		

5.8.6 Suppress Method

The *Suppress Method* is used to suppress a specific *Alarm* instance. It is only available on an instance of an *AlarmConditionType* that also exposes the *SuppressedState*. This *Method* may be used to change the *SuppressedState* of an *Alarm* and overwrite any suppression caused by an associated *AlarmSuppressionGroup*. This *Method* works in parallel with any suppression triggered by an *AlarmSupressionGroup*, in that if the *Method* is used to suppress an *Alarm*, an *AlarmSuppressionGroup* might clear the suppression.

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, *Servers* shall allow *Clients* to call the *Suppress Method* by specifying *ConditionId* as the *ObjectId.* The *Method* may not be called with an *ObjectId* of the *AlarmConditionType Node*.

Signature

```
Suppress();
```

Method Result Codes in Table 41 (defined in Call Service).

Table 41 – Suppress result codes

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

Comments

Suppress Method applies to an *Alarm* instance, even if it is not currently active.

Table 42 specifies the *AddressSpace* representation for the *Suppress Method*.

Table 42 – Suppress Method AddressSpace definition

Attribute	Value				
BrowseName	Suppress				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
AlwaysGeneratesEvent	ObjectType	AuditConditionSuppressionEventType	Defined in 5.10.4		

5.8.7 Unsuppress Method

The *Unsuppress Method* is used to clear the *SuppressedState* of a specific *Alarm* instance. It is only available on an instance of an *AlarmConditionType* that also exposes the *SuppressedState*. This *Method* may be used to overwrite any suppression cause by an associated AlarmSuppressionGroup. This *Method* works in parallel with any suppression triggered by an AlarmSuppressionGroup, in that if the *Method* is used to clear the *SuppressedState* of an *Alarm*, any change in an AlarmSuppressionGroup might again suppress the *Alarm*.

Normally, the *NodeId* of the ObjectInstance is passed as the *ObjectId* to the *Call Service*. However, some *Servers* do not expose *Condition* instances in the *AddressSpace.* Therefore, *Servers* shall allow *Clients* to call the *Unsuppress Method* by specifying *ConditionId* as the *ObjectId.* The *Method* may not be called with an *ObjectId* of the *AlarmConditionType Node*.

Signature

```
Unsuppress();
```

Method Result Codes in Table 43 (defined in Call Service).

Table 43 – Unsuppress result codes

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

Comments

Unsuppress Method applies to an *Alarm* instance, even if it is not currently active.

Table 44 specifies the *AddressSpace* representation for the *Suppress Method*.

Table 44 – Unsuppress Method AddressSpace definition

Attribute	Value				
BrowseName	Unsuppress				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionSuppressionEventType	Defined in 5.10.4		

5.8.8 RemoveFromService Method

The *RemoveFromService Method* is used to suppress a specific *Alarm* instance. It is only available on an instance of an AlarmConditionType that also exposes the *OutOfServiceState*. 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, *Servers* shall allow *Clients* to call the *RemoveFromService Method* by specifying *ConditionId* as the *ObjectId.* The *Method* may not be called with an *ObjectId* of the *AlarmConditionType Node*.

Signature

```
RemoveFromService ();
```

Method result codes in Table 45 (defined in *Call Service*).

Table 45 – RemoveFromService result codes

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

Comments

Instances that do not expose the *OutOfService State* shall reject *RemoveFromService* calls. *RemoveFromService Method* applies to an *Alarm* instance, even if it is not currently in the *Active State*.

Table 46 specifies the *AddressSpace* representation for the *RemoveFromService Method*.

Table 46 – RemoveFromService Method AddressSpace definition

Attribute	Value				
BrowseName	RemoveFromService				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
AlwaysGeneratesEvent	ObjectType	AuditConditionOutOfServiceEventType	Defined in 5.10.12		

5.8.9 PlaceInService Method

The *PlaceInService Method* is used to set the OutOfServiceState to False of a specific *Alarm* instance. It is only available on an instance of an *AlarmConditionType* that also exposes the *OutOfServiceState*. Normally, the *NodeId* of the ObjectInstance is passed as the *ObjectId* to the *Call Service*. However, some *Servers* do not expose *Condition* instances in the *AddressSpace.* Therefore, *Servers* shall allow *Clients* to call the *PlaceInService Method* by specifying *ConditionId* as the *ObjectId.* The *Method* may not be called with an *ObjectId* of the *AlarmConditionType Node*.

Signature

```
PlaceInService ();
```

Method result codes in Table 47 (defined in *Call Service*).

Table 47 – PlaceInService result codes

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

Comments

The *PlaceInService Method* applies to an *Alarm* instance, even if it is not currently in the *Active State*.

Table 48 specifies the *AddressSpace* representation for the *PlaceInService Method*.

Table 48 – PlaceInService Method AddressSpace definition

Attribute	**Value**				
BrowseName	PlaceInService				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionOutOfServiceEventType	Defined in 5.10.12		

5.8.10 ShelvedStateMachineType

5.8.10.1 Overview

The *ShelvedStateMachineType* defines a substate machine that represents an advanced *Alarm* filtering model. This model is illustrated in Figure 15.

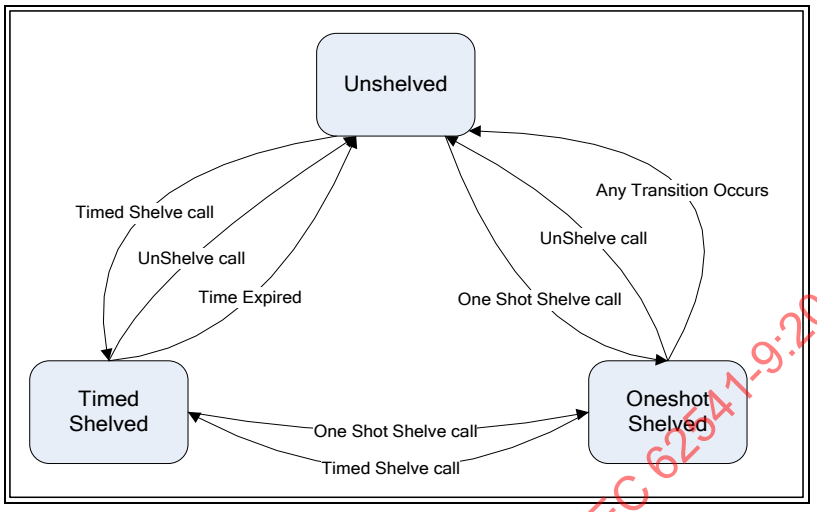
The state model supports two types of *Shelving*: *OneShotShelving* and *TimedShelving*. They are illustrated in Figure 14. The illustration includes the allowed transitions between the various substates. *Shelving* is an *Operator* initiated activity.

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

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

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

All but two transitions are caused by *Method* calls as illustrated in Figure 14. The "Time Expired" transition is simply a system generated transition that occurs when the time value defined as part of the "Timed Shelved Call" has expired. The "Any Transition Occurs" transition is also a system generated transition; this transition is generated when the *Condition* goes to an inactive state.

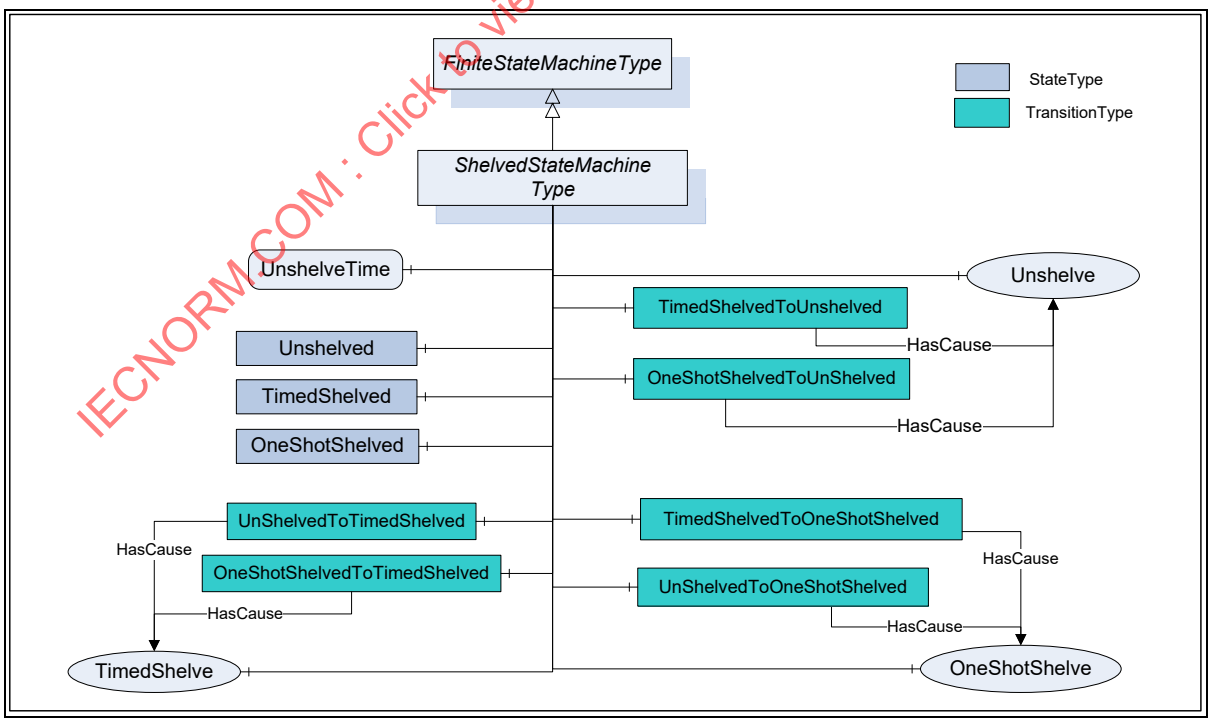


IEC

Figure 14 – Shelve state transitions

The *ShelvedStateMachineType* includes a hierarchy of substates. It supports all transitions between *Unshelved*, *OneShotShelved* and *TimedShelved*.

The state machine is illustrated in Figure 15 and formally defined in Table 49.



IEC

Figure 15 – ShelvedStateMachineType model

Table 49 –ShelvedStateMachineType definition

Attribute	Value				
BrowseName	ShelvedStateMachineType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the Finite*StateMachineType* defined in IEC 62541-5					
HasProperty	Variable	UnshelveTime	Duration	PropertyType	Mandatory
HasComponent	Object	Unshelved		StateType	
HasComponent	Object	TimedShelved		StateType	
HasComponent	Object	OneShotShelved		StateType	
HasComponent	Object	UnshelvedToTimedShelved		TransitionType	
HasComponent	Object	TimedShelvedToUnshelved		TransitionType	
HasComponent	Object	TimedShelvedToOneShotShelved		TransitionType	
HasComponent	Object	UnshelvedToOneShotShelved		TransitionType	
HasComponent	Object	OneShotShelvedToUnshelved		TransitionType	
HasComponent	Object	OneShotShelvedToTimedShelved		TransitionType	
HasComponent	Method	TimedShelve	Defined in 5.8.10.3		Mandatory
HasComponent	Method	OneShotShelve	Defined in 5.8.10.4		Mandatory
HasComponent	Method	Unshelve	Defined in 5.8.10.2		Mandatory

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

This *FiniteStateMachine* supports three *Active* states; *Unshelved*, *TimedShelved* and *OneShotShelved*. It also supports six transitions. The states and transitions are described in Table 50. This *FiniteStateMachine* also supports three *Methods*; *TimedShelve*, *OneShotShelve* and *Unshelve*.

Table 50 – ShelvedStateMachineType transitions

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

5.8.10.2 Unshelve Method

The *Unshelve Method* sets the instance of *AlarmConditionType* to the *Unshelved* state. Normally, the *MethodId* found in the *Shelving* child of the *Condition* instance and the *NodeId* of the *Shelving* object as the *ObjectId* are passed 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 *Unshelve Method* by specifying *ConditionId* as the *ObjectId.* The *Method* may not be called with an *ObjectId* of the *ShelvedStateMachineType Node*.

Signature

```
Unshelve( );
```

Method Result Codes in Table 51 (defined in Call Service).

Table 51 – Unshelve result codes

Result Code	Description
Bad_ConditionNotShelved	See Table 101 for the description of this result code.

Table 52 specifies the *AddressSpace* representation for the *Unshelve Method*.

Table 52 – Unshelve Method AddressSpace definition

Attribute	Value				
BrowseName	Unshelve				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionShelvingEventType	Defined in 5.10.7		

5.8.10.3 TimedShelve Method

The *TimedShelve Method* sets the instance of *AlarmConditionType* to the *TimedShelved* state (parameters are defined in Table 53 and result codes are described in Table 54). Normally, the *MethodId* found in the *Shelving* child of the *Condition* instance and the *NodeId* of the *Shelving* object as the *ObjectId* are passed 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 *TimedShelve Method* by specifying *ConditionId* as the *ObjectId.* The *Method* may not be called with an *ObjectId* of the *ShelvedStateMachineType Node*.

Signature

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

Table 53 – TimedShelve parameters

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

Method Result Codes (defined in Call Service).

Table 54 – TimedShelve result codes

Result Code	Description
Bad_ConditionAlreadyShelved	See Table 101 for the description of this result code. The *Alarm* is already in *TimedShelved* state and the system does not allow a reset of the shelved timer.
Bad_ShelvingTimeOutOfRange	See Table 101 for the description of this result code.

Comments

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

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

Table 55 specifies the *AddressSpace* representation for the *TimedShelve Method*.

Table 55 – TimedShelve Method AddressSpace definition

Attribute	Value				
BrowseName	TimedShelve				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
AlwaysGeneratesEvent	ObjectType	AuditConditionShelvingEventType	Defined in 5.10.7		

5.8.10.4 OneShotShelve Method

The *OneShotShelve Method* sets the instance of *AlarmConditionType* to the *OneShotShelved* state. Normally, the *MethodId* found in the *Shelving* child of the *Condition* instance and the *NodeId* of the *Shelving* object as the *ObjectId* are passed 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 *OneShotShelve Method* by specifying *ConditionId* as the *ObjectId*. The *Method* may not be called with an *ObjectId* of the *ShelvedStateMachineType Node*.

Signature

```
OneShotShelve( );
```

Method Result Codes are defined in Table 56 (status code field is defined in *Call* Service).

Table 56 – OneShotShelve result codes

Result Code	Description
Bad_ConditionAlreadyShelved	See Table 101 for the description of this result code. The *Alarm* is already in *OneShotShelved* state.

Table 57 specifies the *AddressSpace* representation for the *OneShotShelve Method*.

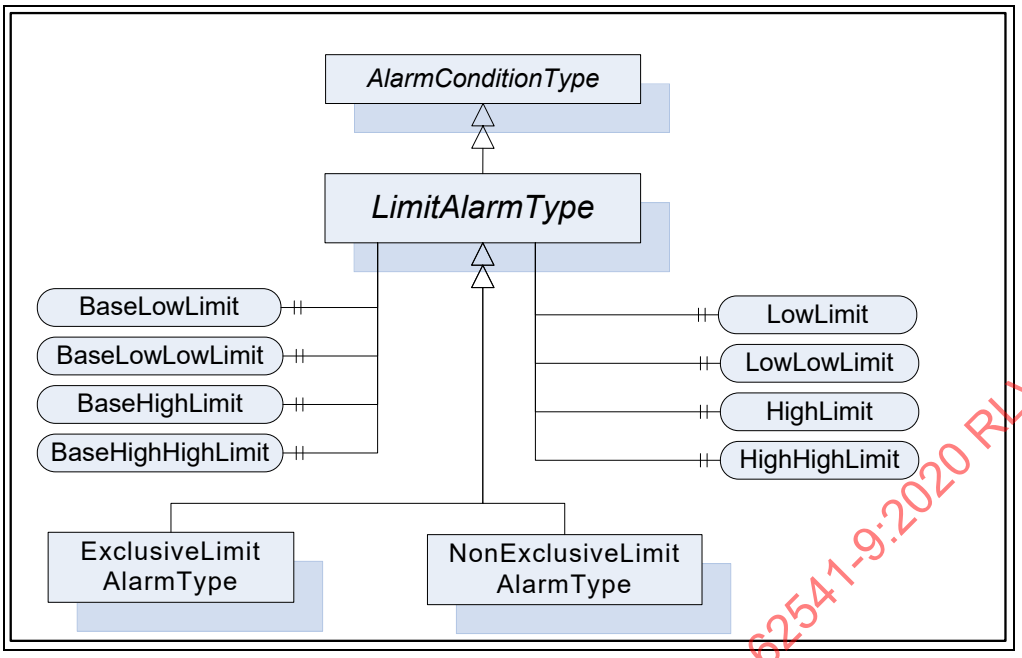
Table 57 – OneShotShelve Method AddressSpace definition

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

5.8.11 LimitAlarmType

Alarms may be modelled with multiple exclusive substates and assigned limits or they may be modelled with nonexclusive limits that may be used to group multiple states together.

The *LimitAlarmType* is an abstract type used to provide a base *Type* for *AlarmConditionTypes* with multiple limits. The *LimitAlarmType* is illustrated in Figure 16.



IEC

Figure 16 – LimitAlarmType

The *LimitAlarmType* is formally defined in Table 58.

Table 58 – LimitAlarmType definition

Attribute	Value				
BrowseName	LimitAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the AlarmConditionType defined in 5.8.2.					
HasSubtype	ObjectType	ExclusiveLimitAlarmType	Defined in 5.8.12.3		
HasSubtype	ObjectType	NonExclusiveLimitAlarmType	Defined in 5.8.13		
HasProperty	Variable	HighHighLimit	Double	PropertyType	Optional
HasProperty	Variable	HighLimit	Double	PropertyType	Optional
HasProperty	Variable	LowLimit	Double	PropertyType	Optional
HasProperty	Variable	LowLowLimit	Double	PropertyType	Optional
HasProperty	Variable	BaseHighHighLimit	Double	PropertyType	Optional
HasProperty	Variable	BaseHighLimit	Double	PropertyType	Optional
HasProperty	Variable	BaseLowLimit	Double	PropertyType	Optional
HasProperty	Variable	BaseLowLowLimit	Double	PropertyType	Optional

Four optional limits are defined which configure the states of the derived limit *Alarm* Types. These *Properties* shall be set for any *Alarm* limits that are exposed by the derived limit *Alarm* types. These *Properties* are listed as optional but at least one is required. For cases where an underlying system cannot provide the actual value of a limit, the limit *Property* shall still be provided, but will have its *AccessLevel* set to not readable. It is assumed that the limits are described using the same Engineering Unit that is assigned to the variable that is the source of the *Alarm*. For Rate of change limit *Alarms*, it is assumed this rate is units per second unless otherwise specified.

Four optional base limits are defined which are used for *AdaptiveAlarming*. They contain the configured *Alarm* limit. If a *Server* supports *AdaptiveAlarming* for *Alarm* limits, the corresponding base *Alarm* limit shall be provided for any limits that are exposed by the derived limit Alarm types. The value of this property is the value of the limit to which an *AdaptiveAlarm* can be reset if any algorithmic changes need to be discarded.

The *Alarm* limits listed may cause an *Alarm* to be generated when a value equals the limit or it may generate the *Alarm* when the limit is exceeded, (i.e. the Value is above the limit for *HighLimit* and below the limit for *LowLimit*). The exact behaviour when the value is equal to the limit is *Server*-specific.

The *Variable* that is the source of the *LimitAlarmType Alarm* shall be a scalar. This *LimitAlarmType* can be subtyped if the *Variable* that is the source is an array. The subtype shall describe the expected behaviour with respect to limits and the array values. Some possible options:

- if any element of the array exceeds the limit, an *Alarm* is generated,
- if all elements exceed the limit, an *Alarm* is generated,
- the limits may also be an array, in which case if any array limit is exceeded by the corresponding source array element, an *Alarm* is generated.

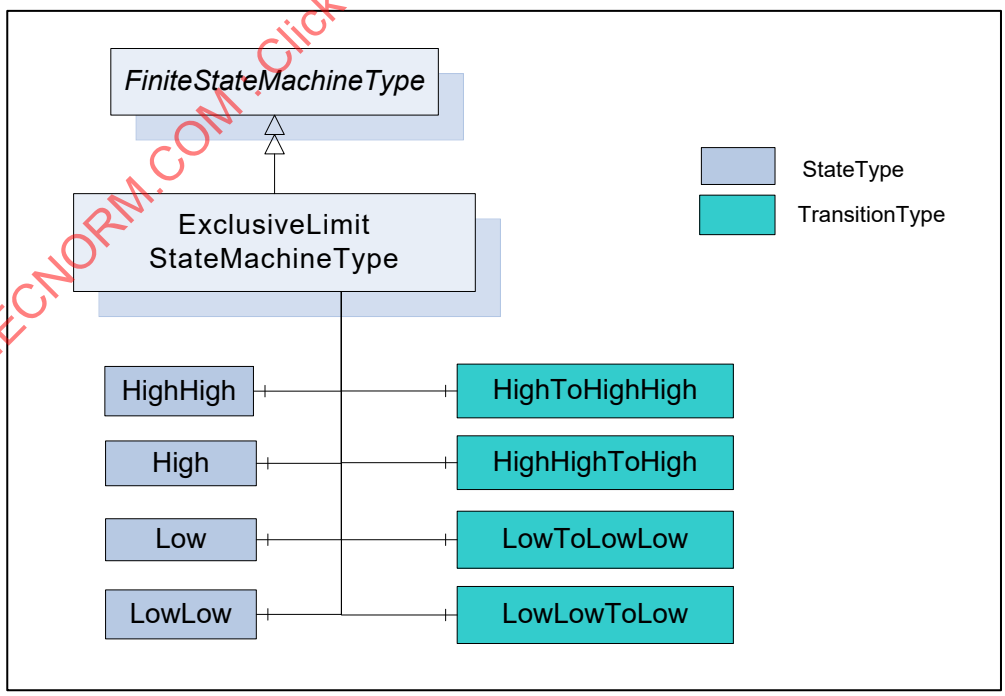
5.8.12 Exclusive limit types

5.8.12.1 Overview

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

5.8.12.2 ExclusiveLimitStateMachineType

The *ExclusiveLimitStateMachineType* defines the state machine used by *AlarmConditionTypes* that handle multiple mutually exclusive limits. It is illustrated in Figure 17.



IEC

Figure 17 – ExclusiveLimitStateMachineType

It is created by extending the *FiniteStateMachineType*. It is formally defined in Table 59 and the state transitions are described in Table 60.

Table 59 – ExclusiveLimitStateMachineType definition

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

Table 60 – ExclusiveLimitStateMachineType transitions

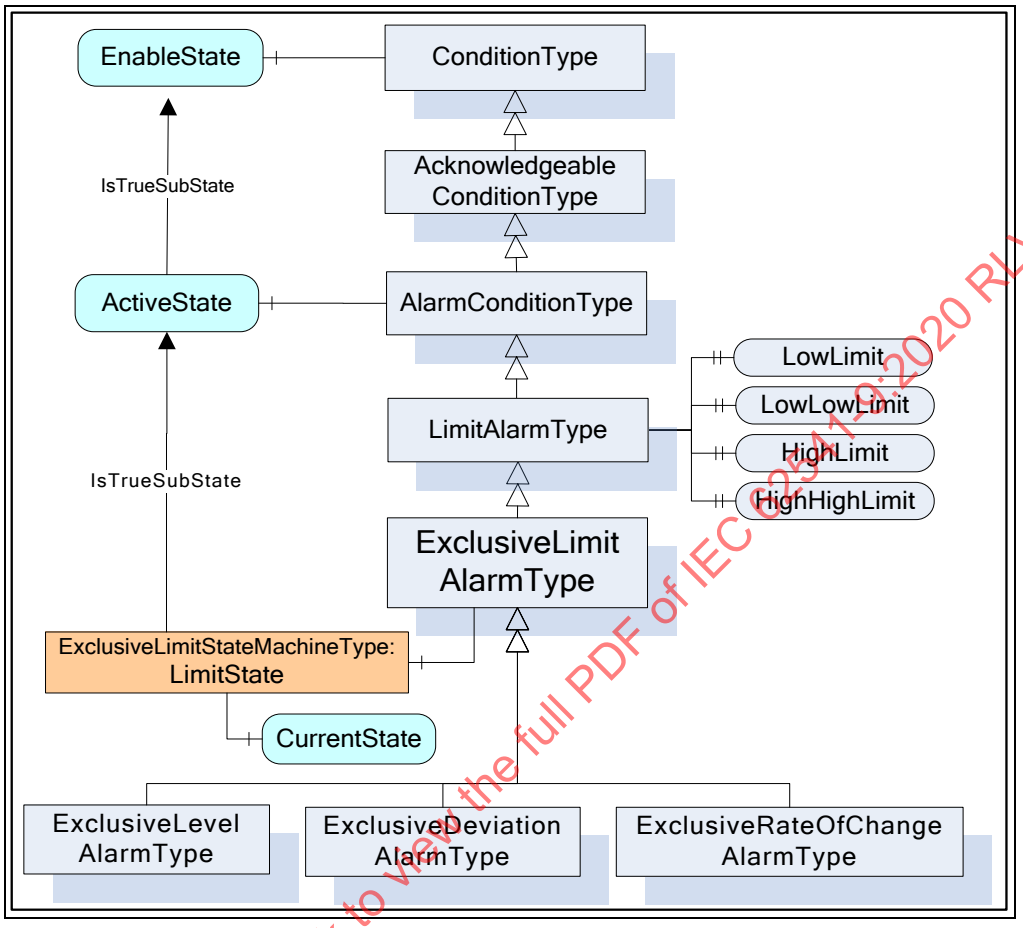
BrowseName	References	BrowseName	TypeDefinition
Transitions			
HighHighToHigh	FromState	HighHigh	StateType
	ToState	High	StateType
	HasEffect	AlarmConditionType	
HighToHighHigh	FromState	High	StateType
	ToState	HighHigh	StateType
	HasEffect	AlarmConditionType	
LowLowToLow	FromState	LowLow	StateType
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

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

The *LowLow*, *Low*, *High*, *HighHigh* are typical for many industries. Vendors may introduce substate models that include additional limits; they may also omit limits in an instance. If a model omits states or transitions in the *StateMachine*, it is recommended that they provide the optional *Property AvailableStates* and/or *AvailableTransitions* (see IEC 62541-5).

5.8.12.3 ExclusiveLimitAlarmType

The *ExclusiveLimitAlarmType* is used to specify the common behaviour for *Alarm Types* with multiple mutually exclusive limits. The *ExclusiveLimitAlarmType* is illustrated in Figure 18.



IEC

Figure 18 – ExclusiveLimitAlarmType

The *ExclusiveLimitAlarmType* is formally defined in Table 61.

Table 61 – ExclusiveLimitAlarmType definition

Attribute	Value				
BrowseName	ExclusiveLimitAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the LimitAlarmType defined in 5.8.11.					
HasSubtype	ObjectType	ExclusiveLevelAlarmType	Defined in 5.8.14.3		
HasSubtype	ObjectType	ExclusiveDeviationAlarmType Type	Defined in 5.8.15.3		
HasSubtype	ObjectType	ExclusiveRateOfChangeAlarm Type	Defined in 5.8.16.3		
HasComponent	Object	LimitState		*ExclusiveLimitStateMachineType*	Mandatory

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

5.8.13 NonExclusiveLimitAlarmType

The *NonExclusiveLimitAlarmType* is used to specify the common behaviour for *Alarm Types* with multiple non-exclusive limits. The *NonExclusiveLimitAlarmType* is illustrated in Figure 19.

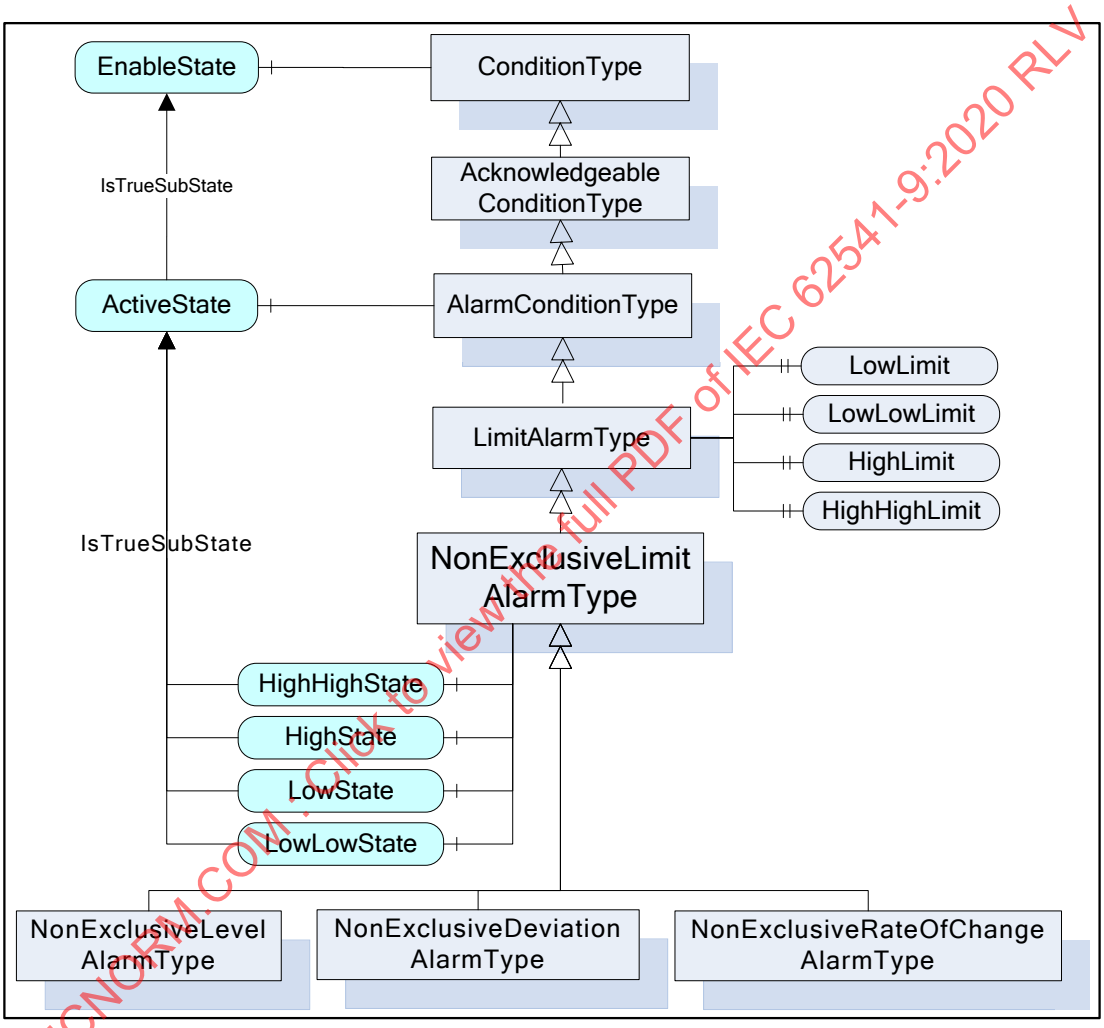


Figure 19 – NonExclusiveLimitAlarmType

The *NonExclusiveLimitAlarmType* is formally defined in Table 62.

Table 62 – NonExclusiveLimitAlarmType definition

Attribute	Value				
BrowseName	NonExclusiveLimitAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the LimitAlarmType defined in 5.8.11.					
HasSubtype	ObjectType	NonExclusiveLevelAlarmType	Defined in 5.8.14.2		
HasSubtype	ObjectType	NonExclusiveDeviationAlarmType	Defined in 5.8.15.2		
HasSubtype	ObjectType	NonExclusiveRateOfChangeAlarmType	Defined in 5.8.16.2		
HasComponent	Variable	HighHighState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Variable	HighState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Variable	LowState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Variable	LowLowState	LocalizedText	TwoStateVariableType	Optional

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

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

5.8.14 Level Alarm

5.8.14.1 Overview

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

5.8.14.2 NonExclusiveLevelAlarmType

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

The *NonExclusiveLevelAlarmType* is based on the *NonExclusiveLimitAlarmType*. It is formally defined in Table 63.

Table 63 – NonExclusiveLevelAlarmType definition

Attribute	Value				
BrowseName	NonExclusiveLevelAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the NonExclusiveLimitAlarmType defined in 5.8.13.					

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

5.8.14.3 ExclusiveLevelAlarmType

The *ExclusiveLevelAlarmType* is a special level *Alarm* utilized with multiple mutually exclusive limits. It is formally defined in Table 64.

Table 64 – ExclusiveLevelAlarmType definition

Attribute	Value				
BrowseName	ExclusiveLevelAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherits the Properties of the ExclusiveLimitAlarmType defined in 5.8.12.3.					

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

5.8.15 Deviation Alarm

5.8.15.1 Overview

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

For example, if a set point had a value of 10, a high deviation *Alarm* limit of 2 and a low deviation *Alarm* limit of −1, then the low substate is entered if the process value drops below 9; the high substate is entered if the process value raises above 12. If the set point were changed to 11 then the new deviation values would be 10 and 13 respectively. The set point may be fixed by a configuration, adjusted by an *Operator* or it may be adjusted by an algorithm, the actual functionality exposed by the set point is application specific. The deviation *Alarm* may also be used to report a problem between a redundant data source where the difference between the primary source and the secondary source exceeds the included limit. In this case, the *SetpointNode* would point to the secondary source.

5.8.15.2 NonExclusiveDeviationAlarmType

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

The *NonExclusiveDeviationAlarmType* is based on the *NonExclusiveLimitAlarmType*. It is formally defined in Table 65.

Table 65 – NonExclusiveDeviationAlarmType definition

Attribute	Value				
BrowseName	NonExclusiveDeviationAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the NonExclusiveLimitAlarmType defined in 5.8.13.					
HasProperty	Variable	SetpointNode	NodeId	PropertyType	Mandatory
HasProperty	Variable	BaseSetpointNode	NodeId	PropertyType	Optional

The *SetpointNode Property* provides the *NodeId* of the set point used in the deviation calculation. In cases where the *Alarm* is generated by an underlying system and if the *Variable* is not in the *AddressSpace*, a NULL *NodeId* shall be provided.

The *BaseSetpointNode Property* provides the *NodeId* of the original or base setpoint. The value of this node is the value of the setpoint to which an *AdaptiveAlarm* may be reset if any algorithmic changes need to be discarded. The value of this node usually contains the originally configured set point.

5.8.15.3 ExclusiveDeviationAlarmType

The *ExclusiveDeviationAlarmType* is utilized with multiple mutually exclusive limits. It is formally defined in Table 66.

Table 66 – ExclusiveDeviationAlarmType definition

Attribute	Value				
BrowseName	ExclusiveDeviationAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Inherits the *Properties* of the *ExclusiveLimitAlarmType* defined in 5.8.12.3.					
HasProperty	Variable	SetpointNode	NodeId	PropertyType	Mandatory
HasProperty	Variable	BaseSetpointNode	NodeId	PropertyType	Optional

The *SetpointNode Property* provides the *NodeId* of the set point used in the *Deviation* calculation. If this *Variable* is not in the *AddressSpace*, a NULL *NodeId* shall be provided.

The *BaseSetpointNode Property* provides the *NodeId* of the original or base setpoint. The value of this node is the value of the set point to which an *AdaptiveAlarm* may be reset if any algorithmic changes need to be discarded. The value of this node usually contains the originally configured set point.

5.8.16 Rate of change Alarms

5.8.16.1 Overview

A *Rate* of *Change Alarm* is commonly used to report an unusual change or lack of change in a measured value related to the speed at which the value has changed. The *Rate* of *Change Alarm* becomes active when the rate at which the value changes exceeds or drops below a defined limit.

A *Rate* of *Change* is measured in some time unit, such as seconds or minutes and some unit of measure, such as percent or metre. For example, a tank may have a High limit for the *Rate* of *Change* of its level (measured in metres) which would be 4 metres per minute. If the tank level changes at a rate that is greater than 4 metres per minute, then the High substate is entered.

5.8.16.2 NonExclusiveRateOfChangeAlarmType

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

The *NonExclusiveRateOfChangeAlarmType* is based on the *NonExclusiveLimitAlarmType*. It is formally defined in Table 67.

Table 67 – NonExclusiveRateOfChangeAlarmType definition

Attribute	Value				
BrowseName	NonExclusiveRateOfChangeAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the NonExclusiveLimitAlarmType defined in clause 5.8.13.					
HasProperty	Variable	EngineeringUnits	EUInformation	PropertyType	Optional

EngineeringUnits provides the engineering units associated with the limits values. If this is not provided, the assumed Engineering Unit is the same as the EU associated with the parent variable per second e.g. if parent is meters, this unit is meters/second.

5.8.16.3 ExclusiveRateOfChangeAlarmType

ExclusiveRateOfChangeAlarmType is utilized with multiple mutually exclusive limits. It is formally defined in Table 68.

Table 68 – ExclusiveRateOfChangeAlarmType definition

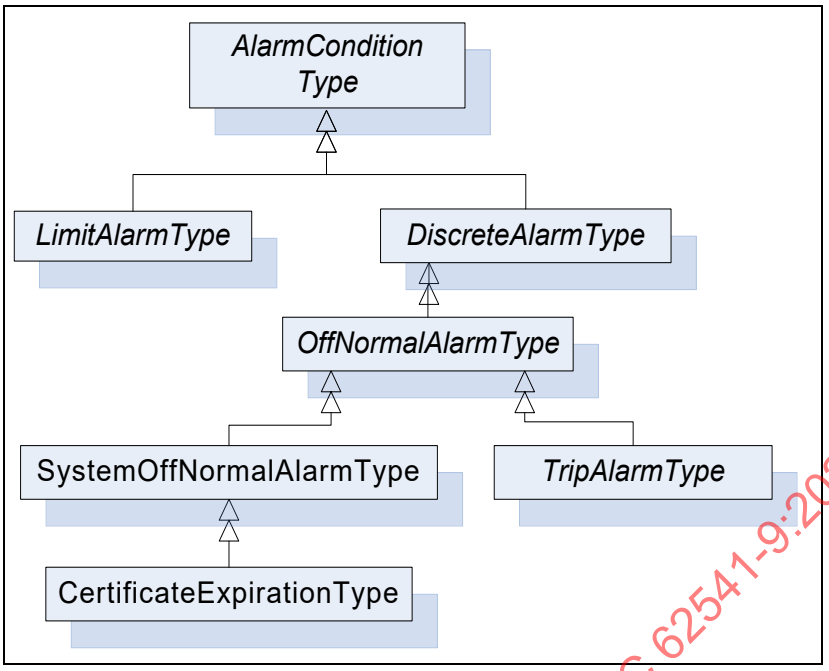
Attribute	Value				
BrowseName	ExclusiveRateOfChangeAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherits the *Properties* of the *ExclusiveLimitAlarmType* defined in 5.8.12.3.					
HasProperty	Variable	EngineeringUnits	EUInformation	PropertyType	Optional

EngineeringUnits provides the engineering units associated with the limits values. If this is not provided, the assumed Engineering Unit is the same as the EU associated with the parent variable per second; e.g. if parent is metres, this unit is metres/second.

5.8.17 Discrete Alarms

5.8.17.1 DiscreteAlarmType

The *DiscreteAlarmType* is used to classify *Types* into *Alarm Conditions* where the input for the *Alarm* may take on only a certain number of possible values (e.g. True/False, running/stopped/terminating). The *DiscreteAlarmType* with subtypes defined in this document is illustrated in Figure 20. It is formally defined in Table 69.



IEC

Figure 20 – DiscreteAlarmType Hierarchy

Table 69 – DiscreteAlarmType definition

Attribute	Value				
BrowseName	DiscreteAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the AlarmConditionType defined in 5.8.2.					
HasSubtype	*ObjectType*	OffNormalAlarmType	Defined in 5.8.15		

5.8.17.2 OffNormalAlarmType

The *OffNormalAlarmType* is a specialization of the *DiscreteAlarmType* intended to represent a discrete *Condition* that is considered to be not normal. It is formally defined in Table 70. This subtype is usually used to indicate that a discrete value is in an *Alarm* state, it is active as long as a non-normal value is present.

Table 70 – OffNormalAlarmType Definition

Attribute	Value				
BrowseName	OffNormalAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the DiscreteAlarmType defined in 5.8.17.1					
HasSubtype	*ObjectType*	TripAlarmType	Defined in 5.8.17.4		
HasSubtype	*ObjectType*	SystemOffNormalAlarmType	Defined in 5.8.17.3		
HasProperty	*Variable*	NormalState	NodeId	PropertyType	Mandatory

The *NormalState Property* is a *Property* that points to a *Variable* which has a value that corresponds to one of the possible values of the *Variable* pointed to by the *InputNode Property* where the *NormalState Property Variable* value is the value that is considered to be the normal state of the *Variable* pointed to by the *InputNode Property*. When the value of the *Variable* referenced by the *InputNode Property* is not equal to the value of the *NormalState Property* the *Alarm* is *Active*. If this *Variable* is not in the *AddressSpace*, a NULL *NodeId* shall be provided.

5.8.17.3 SystemOffNormalAlarmType

This *Condition* is used by a *Server* to indicate that an underlying system that is providing *Alarm* information is having a communication problem and that the *Server* may have invalid or incomplete *Condition* state in the *Subscription*. Its representation in the *AddressSpace* is formally defined in Table 71.

Table 71 – SystemOffNormalAlarmType definition

Attribute	Value				
BrowseName	SystemOffNormalAlarmType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasSubtype	*ObjectType*	CertificateExpirationAlarmType	Defined in 5.8.17.7		
Subtype of the *OffNormalAlarmType*, i.e. it has HasProperty *References* to the same *Nodes*.					

5.8.17.4 TripAlarmType

The *TripAlarmType* is a specialization of the *OffNormalAlarmType* intended to represent an equipment trip *Condition*. The *Alarm* becomes active when the monitored piece of equipment experiences some abnormal fault such as a motor shutting down due to an overload condition. It is formally defined in Table 72. This *Type* is mainly used for categorization.

Table 72 – TripAlarmType definition

Attribute	Value				
BrowseName	TripAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the OffNormalAlarmType defined in 5.8.17.2.					

5.8.17.5 InstrumentDiagnosticAlarmType

The *InstrumentDiagnosticAlarmType* is a specialization of the *OffNormalAlarmType* intended to represent a fault in a field device. The *Alarm* becomes active when the monitored device experiences a fault such as a sensor failure. It is formally defined in Table 73. This *Type* is mainly used for categorization.

Table 73 – InstrumentDiagnosticAlarmType definition

Attribute	Value				
BrowseName	InstrumentDiagnosticAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the OffNormalAlarmType defined in clause 5.8.17.2.					

5.8.17.6 SystemDiagnosticAlarmType

The *SystemDiagnosticAlarmType* is a specialization of the *OffNormalAlarmType* intended to represent a fault in a system or sub-system. The *Alarm* becomes active when the monitored system experiences a fault. It is formally defined in Table 74. This *Type* is mainly used for categorization.

Table 74 – SystemDiagnosticAlarmType definition

Attribute	Value				
BrowseName	SystemDiagnosticAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the OffNormalAlarmType defined in 5.8.17.2.					

5.8.17.7 CertificateExpirationAlarmType

This *SystemOffNormalAlarmType* is raised by the *Server* when the *Server's* Certificate is within the *ExpirationLimit* of expiration. This *Alarm* automatically returns to normal when the certificate is updated.

The *SystemOffNormalAlarmType* is formally defined in Table 75.

Table 75 – CertificateExpirationAlarmType definition

Attribute	Value				
BrowseName	CertificateExpirationAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the SystemOffNormalAlarmType defined in 5.8.17.3					
HasProperty	Variable	ExpirationDate	DateTime	PropertyType	Mandatory
HasProperty	Variable	ExpirationLimit	Duration	PropertyType	Optional
HasProperty	Variable	CertificateType	NodeId	PropertyType	Mandatory
HasProperty	Variable	Certificate	ByteString	PropertyType	Mandatory

ExpirationDate is the date and time this certificate will expire.

ExpirationLimit is the time interval before the *ExpirationDate* at which this *Alarm* will trigger. This shall be a positive number. If the property is not provided, a default of 2 weeks shall be used.

CertificateType – See Part 12 for definition of *CertificateType*.

Certificate is the certificate that is about to expire.

5.8.18 DiscrepancyAlarmType

The *DiscrepancyAlarmType* is commonly used to report an action that did not occur within an expected time range.

The *DiscrepancyAlarmType* is based on the *AlarmConditionType*. It is formally defined in Table 76.

Table 76 – DiscrepancyAlarmType definition

Attribute	Value				
BrowseName	DiscrepancyAlarmType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AlarmConditionType* defined in 5.8.2.					
HasProperty	Variable	TargetValueNode	NodeId	PropertyType	Mandatory
HasProperty	Variable	ExpectedTime	Duration	PropertyType	Mandatory
HasProperty	Variable	Tolerance	Double	PropertyType	Optional

The *TargetValueNode Property* provides the *NodeId* of the *Variable* that is used for the target value.

The *ExpectedTime Property* provides the *Duration* within which the value pointed to by the InputNode shall equal the value specified by the *TargetValueNode* (or be within the *Tolerance* range, if specified).

The *Tolerance Property* is a value that may be added to or subtracted from the *TargetValueNode's* value, providing a range that the value can be in without generating the *Alarm*.

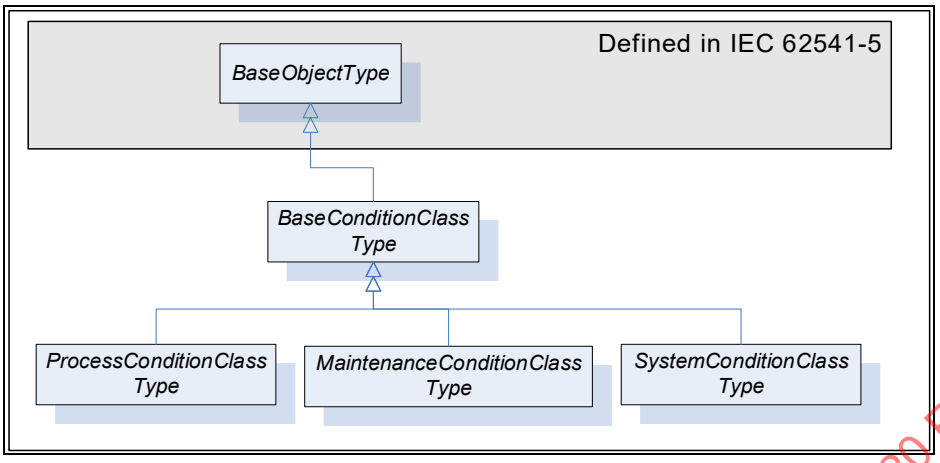
A *DiscrepancyAlarmType* may be used to indicate a motor has not responded to a start request within a given time, or that a process value has not reached a given value after a setpoint change within a given time interval.

The *DiscrepancyAlarmType* shall return to normal when the value has reached the target value.

5.9 ConditionClasses

5.9.1 Overview

Conditions are used in specific application domains like Maintenance, System or Process. The *ConditionClass* hierarchy is used to specify domains and is orthogonal to the *ConditionType* hierarchy. The *ConditionClassId Property* of the *ConditionType* is used to assign a *Condition* to a *ConditionClass*. *Clients* may use this *Property* to filter out essential classes. OPC UA defines the base *ObjectType* for all *ConditionClasses* and a set of common classes used across many industries. Figure 21 informally describes the hierarchy of *ConditionClass Types* defined in this document.



IEC

Figure 21 – ConditionClass type hierarchy

ConditionClasses are not representations of *Objects* in the underlying system and, therefore, only exist as *Type Nodes* in the *Address Space*.

5.9.2 BaseConditionClassType

BaseConditionClassType is used as class whenever a *Condition* cannot be assigned to a more concrete class. *Servers* should use a more specific *ConditionClass*, if possible. All *ConditionClass Types* derive from *BaseConditionClassType*. It is formally defined in Table 77.

Table 77 – BaseConditionClassType definition

Attribute	Value				
BrowseName	BaseConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the BaseObjectType defined in IEC 62541-5.					

5.9.3 ProcessConditionClassType

The *ProcessConditionClassType* is used to classify *Conditions* related to the process itself. Examples of a process would be a control system in a boiler, or the instrumentation associated with a chemical plant or paper machine. The *ProcessConditionClassType* is formally defined in Table 78.

Table 78 – ProcessConditionClassType definition

Attribute	Value				
BrowseName	ProcessConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the BaseConditionClassType defined in 5.9.2.					

5.9.4 MaintenanceConditionClassType

The *MaintenanceConditionClassType* is used to classify *Conditions* related to maintenance. Examples of maintenance would be Asset Management systems or conditions, which occur in process control systems, which are related to calibration of equipment. The *MaintenanceConditionClassType* is formally defined in Table 79. No further definition is provided here. It is expected that other standards development groups will define domain-specific subtypes.

Table 79 – MaintenanceConditionClassType definition

Attribute	Value				
BrowseName	MaintenanceConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the BaseConditionClassType defined in 5.9.2.					

5.9.5 SystemConditionClassType

The *SystemConditionClassType* is used to classify *Conditions* related to the System. It is formally defined in Table 80. System *Conditions* occur in the controlling or monitoring system process. Examples of System related items could include available disk space on a computer, Archive media availability, network loading issues or a controller error. No further definition is provided here. It is expected that other standards development groups or vendors will define domain-specific subtypes.

Table 80 – SystemConditionClassType definition

Attribute	Value				
BrowseName	SystemConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseConditionClassType* defined in 5.9.2.					

5.9.6 SafetyConditionClassType

The *SafetyConditionClassType* is used to classify *Conditions* related to safety. It is formally defined in Table 81.

Safety *Conditions* occur in the controlling or monitoring system process. Examples of safety related items could include emergency shutdown systems or fire suppression systems.

Table 81 – SafetyConditionClassType definition

Attribute	Value				
BrowseName	SafetyConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseConditionClassType* defined in 5.9.2.					

5.9.7 HighlyManagedAlarmConditionClassType

In *Alarm* systems some *Alarms* may be classified as highly managed *Alarms*. This class of *Alarm* requires special handling that varies according to the individual requirements. It might require individual acknowledgement or not allow suppression or any of a number of other special behaviours. The *HighlyManagedAlarmConditionClassType* is used to classify *Conditions* as highly managed *Alarms*. It is formally defined in Table 82.

Table 82 – HighlyManagedAlarmConditionClassType definition

Attribute	Value				
BrowseName	HighlyManagedAlarmConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseConditionClassType* defined in 5.9.2.					

5.9.8 TrainingConditionClassType

The *TrainingConditionClassType* is used to classify *Conditions* related to training system or training exercises. It is formally defined in Table 83. These *Conditions* typically occur in a training system or are generated as part of a simulation for a training exercise. Training *Conditions* might be process or system conditions. It is expected that other standards development groups or vendors will define domain-specific subtypes.

Table 83 – TrainingConditionClassType definition

Attribute	Value				
BrowseName	TrainingConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseConditionClassType* defined in 5.9.2.					

5.9.9 StatisticalConditionClassType

The *StatisticalConditionClassType* is used to classify *Conditions* related that are based on statistical calculations. It is formally defined in Table 84. These *Conditions* are generated as part of a statistical analysis. They might be any of an *Alarm* number of types.

Table 84 – StatisticalConditionClassType definition

Attribute	Value				
BrowseName	StatisticalConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseConditionClassType* defined in 5.9.2.					

5.9.10 TestingConditionSubClassType

The *TestingConditionSubClassType* is used to classify *Conditions* related to testing of an *Alarm* system or *Alarm* function. It is formally defined in Table 85. Testing *Conditions* might include a condition to test an alarm annunciation such as a horn or other panel. It might also be used to temporarily reclassify a *Condition* to check response times or suppression logic. It is expected that other standards development groups or vendors will define domain-specific subtypes.

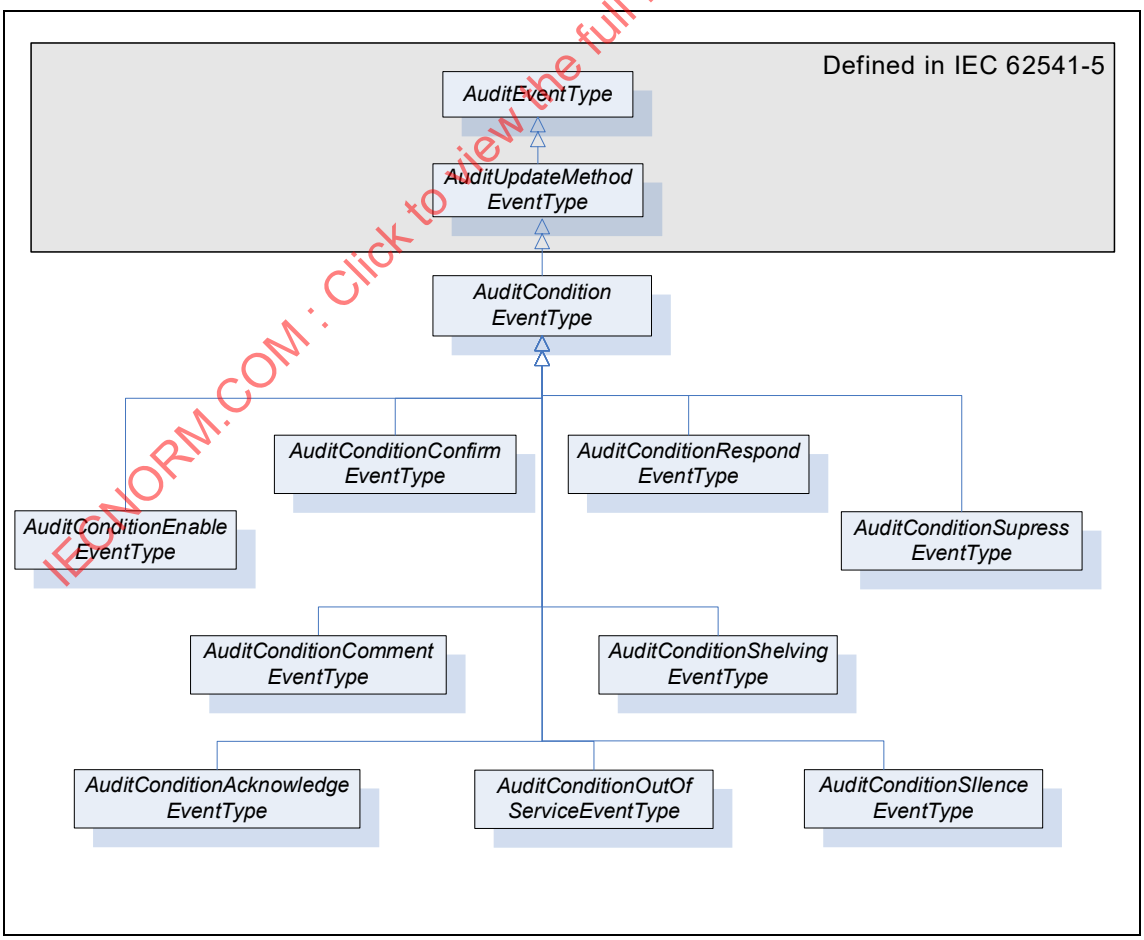
Table 85 – TestingConditionSubClassType definition

Attribute	Value				
BrowseName	TestingConditionSubClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseConditionClassType* defined in 5.9.2.					

5.10 Audit Events

5.10.1 Overview

Following are subtypes of *AuditUpdateMethodEventType* that will be generated in response to the *Methods* defined in this document. They are illustrated in Figure 22.



IEC

Figure 22 – AuditEvent hierarchy

AuditConditionEventTypes are normally used in response to a *Method* call. However, these *Events* shall also be notified if the functionality of such a *Method* is performed by some other *Server*-specific means. In this case, the *SourceName Property* shall contain a proper description of this internal means and the other *Properties* should be filled in as described for the given *EventType*.

5.10.2 AuditConditionEventType

This *EventType* is used to subsume all *AuditConditionEventTypes*. It is formally defined in Table 86.

Table 86 – AuditConditionEventType definition

Attribute	Value				
BrowseName	AuditConditionEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditUpdateMethodEventType* defined in IEC 62541-5					

AuditConditionEventTypes inherit all *Properties* of the *AuditUpdateMethodEventType* defined in IEC 62541-5. Unless a subtype overrides the definition, the inherited *Properties* of the *Condition* will be used as defined.

- The inherited *Property SourceNode* shall be filled with the *ConditionId*.
- The *SourceName* shall be "Method/" and the name of the *Service* that generated the *Event* (e.g. *Disable*, Enable, *Acknowledge,* etc.*)*.

This *EventType* can be further customized to reflect particular *Condition* related actions.

5.10.3 AuditConditionEnableEventType

This *EventType* is used to indicate a change in the enabled state of a *Condition* instance. It is formally defined in Table 87.

Table 87 – AuditConditionEnableEventType definition

Attribute	Value				
BrowseName	AuditConditionEnableEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

The SourceName shall indicate Method/Enable or Method/Disable. If the audit *Event* is not the result of a *Method* call, but due to an internal action of the *Server,* the *SourceName* shall reflect Enable or Disable, it may be preceded by an appropriate description such as "Internal/Enable" or "Remote/Enable".

5.10.4 AuditConditionCommentEventType

This *EventType* is used to report an *AddComment* action. It is formally defined in Table 88.

Table 88 – AuditConditionCommentEventType definition

Attribute		Value			
BrowseName		AuditConditionCommentEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	ConditionEventId	ByteString	PropertyType	Mandatory
HasProperty	Variable	Comment	LocalizedText	PropertyType	Mandatory
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

The *ConditionEventId* field shall contain the id of the event for which the comment was added.

The *Comment* contains the actual comment that was added.

5.10.5 AuditConditionRespondEventType

This *EventType* is used to report a *Respond* action (see 5.6). It is formally defined in Table 89.

Table 89 – AuditConditionRespondEventType definition

Attribute		Value			
BrowseName		AuditConditionRespondEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	SelectedResponse	UInt32	PropertyType	Mandatory
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

The SelectedResponse field shall contain the response that was selected.

5.10.6 AuditConditionAcknowledgeEventType

This *EventType* is used to indicate acknowledgement or confirmation of one or more *Conditions*. It is formally defined in Table 90.

Table 90 – AuditConditionAcknowledgeEventType definition

Attribute		Value			
BrowseName		AuditConditionAcknowledgeEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	ConditionEventId	ByteString	PropertyType	Mandatory
HasProperty	Variable	Comment	LocalizedText	PropertyType	Mandatory
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

The *ConditionEventId* field shall contain the id of the *Event* that was acknowledged.

The *Comment* contains the actual comment that was added; it may be a blank comment or a NULL.

5.10.7 AuditConditionConfirmEventType

This *EventType* is used to report a *Confirm* action. It is formally defined in Table 91.

Table 91 – AuditConditionConfirmEventType definition

Attribute		Value			
BrowseName		AuditConditionConfirmEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	ConditionEventId	ByteString	PropertyType	Mandatory
HasProperty	Variable	Comment	LocalizedText	PropertyType	Mandatory
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

The *ConditionEventId* field shall contain the id of the *Event* that was confirmed.

The *Comment* contains the actual comment that was added; it may be a blank comment or a NULL.

5.10.8 AuditConditionShelvingEventType

This *EventType* is used to indicate a change to the *Shelving* state of a *Condition* instance. It is formally defined in Table 92.

Table 92 – AuditConditionShelvingEventType definition

Attribute		Value			
BrowseName		AuditConditionShelvingEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	ShelvingTime	Duration	PropertyType	Optional
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

If the *Method* indicates a *TimedShelve* operation, the *ShelvingTime* field shall contain duration for which the *Alarm* is to be shelved. For other *Shelving Methods*, this parameter may be omitted or NULL.

5.10.9 AuditConditionSuppressionEventType

This *EventType* is used to indicate a change to the *Suppression* state of a *Condition* instance. It is formally defined in Table 93.

Table 93 – AuditConditionSuppressionEventType definition

Attribute		Value			
BrowseName		AuditConditionSuppressionEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

This *Event* indicates an *Alarm* suppression operation. An audit *Event* of this type shall be generated, if audit events are supported for any suppression action, including automatic system-based suppression.

5.10.10 AuditConditionSilenceEventType

This *EventType* is used to indicate a change to the *Silence* state of a *Condition* instance. It is formally defined in Table 94.

Table 94 – AuditConditionSilenceEventType definition

Attribute		Value			
BrowseName		AuditConditionSilenceEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

This event indicates that an *Alarm* was silenced, but not acknowledged. An audit event of this type shall be generated, if Audit events are supported for any silence action, including automatic system-based silence.

5.10.11 AuditConditionResetEventType

This *EventType* is used to indicate a change to the *Latched* state of a *Condition* instance. It is formally defined in Table 95.

Table 95 – AuditConditionResetEventType definition

Attribute		Value			
BrowseName		AuditConditionResetEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

This event indicates that an *Alarm* was reset. An audit event of this type shall be generated, if Audit events are supported for any *Alarm* action.

5.10.12 AuditConditionOutOfServiceEventType

This *EventType* is used to indicate a change to the *OutOfService State* of a *Condition* instance. It is formally defined in Table 96.

Table 96 – AuditConditionOutOfServiceEventType definition

Attribute		Value			
BrowseName		AuditConditionOutOfServiceEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditConditionEventType* defined in 5.10.2 that is, inheriting the InstanceDeclarations of that Node.					

An audit *Event* of this type shall be generated if audit *Events* are supported.

5.11 Condition Refresh related Events

5.11.1 Overview

Following are subtypes of *SystemEventType* that will be generated in response to a *Refresh Methods* call. They are illustrated in Figure 23.

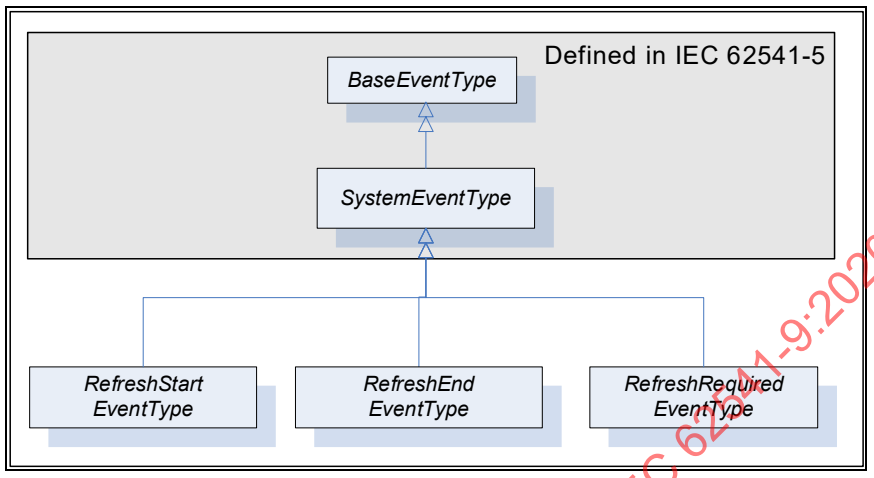


Figure 23 – Refresh Related Event Hierarchy

5.11.2 RefreshStartEventType

This *EventType* is used by a *Server* to mark the beginning of a *Refresh Notification* cycle. Its representation in the *AddressSpace* is formally defined in Table 97.

Table 97 – RefreshStartEventType definition

Attribute	Value				
BrowseName	RefreshStartEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *SystemEventType* defined in IEC 62541-5, i.e. it has HasProperty *References* to the same *Nodes*.					

5.11.3 RefreshEndEventType

This *EventType* is used by a *Server* to mark the end of a *Refresh Notification* cycle. Its representation in the *AddressSpace* is formally defined in Table 98.

Table 98 – RefreshEndEventType definition

Attribute	Value				
BrowseName	RefreshEndEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *SystemEventType* defined in IEC 62541-5, i.e. it has HasProperty *References* to the same *Nodes*.					

5.11.4 RefreshRequiredEventType

This *EventType* is used by a *Server* to indicate that a significant change has occurred in the *Server* or in the subsystem below the *Server* that may or does invalidate the *Condition* state of a *Subscription*. Its representation in the *AddressSpace* is formally defined in Table 99.

Table 99 – RefreshRequiredEventType definition

Attribute	**Value**				
BrowseName	RefreshRequiredEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *SystemEventType* defined in IEC 62541-5, i.e. it has HasProperty *References* to the same *Nodes*.					

When a *Server* detects an *Event* queue overflow, it shall track if any *Condition Event*s have been lost, if any *Condition Event*s were lost, it shall issue a *RefreshRequiredEventType Event* to the *Client* after the *Event* queue is no longer in an overflow state.

5.12 HasCondition Reference type

The *HasCondition ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *NonHierarchicalReferences*. The representation in the *AddressSpace* is specified in Table 100.

The semantic of this *ReferenceType* is to specify the relationship between a *ConditionSource* and its *Conditions*. Each *ConditionSource* shall be the target of a *HasEventSource Reference* or a subtype of *HasEventSource*. The *AddressSpace* organisation that shall be provided for *Clients* to detect *Conditions* and *ConditionSources* is defined in Clause 6. Various examples for the use of this *ReferenceType* may be found in Clause B.2.

HasCondition References can be used in the *Type* definition of an *Object* or a *Variable*. In this case, the *SourceNode* of this *ReferenceType* shall be an *ObjectType* or *VariableType Node* or one of their *InstanceDeclaration Nodes*. The *TargetNode* shall be a *Condition* instance declaration or a *ConditionType*. The following rules for instantiation apply:

- all *HasCondition References* used in a *Type* shall exist in instances of these *Types* as well;
- if the *TargetNode* in the *Type* definition is a *ConditionType*, the same *TargetNode* will be referenced on the instance.

HasCondition References may be used solely in the instance space when they are not available in *Type* definitions. In this case, the *SourceNode* of this *ReferenceType* shall be an *Object*, *Variable* or *Method Node*. The *TargetNode* shall be a *Condition* instance or a *ConditionType*.

Table 100 – HasCondition *ReferenceType*

Attributes	**Value**		
BrowseName	HasCondition		
InverseName	IsConditionOf		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**

5.13 Alarm and Condition status codes

Table 101 defines the *StatusCodes* defined for *Alarm* and *Conditions* (A&C).

Table 101 – Alarm & Condition result codes

Symbolic Id	Description
Bad_ConditionAlreadyEnabled	The addressed Condition is already enabled.
Bad_ConditionAlreadyDisabled	The addressed Condition is already disabled.
Bad_ConditionAlreadyShelved	The Alarm is already in a shelved state.
Bad_ConditionBranchAlreadyAcked	The *EventId* does not refer to a state that needs acknowledgement.
Bad_ConditionBranchAlreadyConfirmed	The *EventId* does not refer to a state that needs confirmation.
Bad_ConditionNotShelved	The Alarm is not in the requested shelved state.
Bad_DialogNotActive	The *DialogConditionType* instance is not in *Active* state.
Bad_DialogResponseInvalid	The selected option is not a valid index in the *ResponseOptionSet* array.
Bad_EventIdUnknown	The specified *EventId* is not known to the *Server*.
Bad_RefreshInProgress	A ConditionRefresh operation is already in progress.
Bad_ShelvingTimeOutOfRange	The provided *Shelving* time is outside the range allowed by the *Server* for *Shelving*

5.14 Expected A&C server behaviours

5.14.1 General

This subclause describes behaviour that is expected from an OPC UA *Server* that is implementing the *A&C Information Model*. In particular this subclause describes specific behaviours that apply to various aspect of the A&C *Information Model*.

5.14.2 Communication problems

In some implementation of an OPC UA A&C *Server*, the *Alarms* and *Condition* are provided by an underlying system. The expected behaviour of an A&C *Server* when it is encountering communication problems with the underlying system is:

- If communication fails to the underlying system,
  - For any *Event* field related information that is exposed in the address space, the Value/*StatusCode* obtained when reading the *Event* fields that are associated with the communication failure shall have a value of NULL and a *StatusCode* of Bad_CommunicationError.
  - For *Subscriptions* that contain *Conditions* for which the failure applies, the effected *Conditions* generate an *Event,* if the *Retain* field is set to True. These *Events* shall have their *Event* fields that are associated with the communication failure contain a *StatusCode* of Bad_CommunicationError for the value.
  - A *Condition* of the SystemOffNormalAlarmType shall be used to report the communication failure to *Alarm Clients*. The *NormalState* field shall contain the *NodeId* of the *Variable* that indicates the status of the underlying system.
- For start-up of an A&C *Server* that is obtaining A&C information from an already running underlying system:
  - If a value is unavailable for an *Event* field that is being reported due to a start-up of the UA *Server* (i.e. the information is just not available for the *Event*) the *Event* field shall contain a *StatusCode* set to Bad_WaitingForInitialData for the value.
  - If the "Time" field is normally provided by the underlying system and is unavailable, the Time will be reported as a *StatusCode* with a value of Bad_WaitingForInitialData.

5.14.3 Redundant A&C servers

In an OPC UA *Server* that is implementing the A&C *Information Model* and that is configured to be a redundant OPC UA *Server*, the following behaviour is expected:

- The EventId is used to uniquely identify an *Event*. For an *Event* that is in each of the redundant *Servers*, it shall be identical. This applies to all standard *Events*, *Alarms* and *Conditions*. This may be accomplished by sharing of information between redundant *Server* (such as actual *Events*) or it may be accomplished by providing a strict *EventId* generating algorithm that will generate an identical *EventId* for each *Event*.
- It is expected that for cold or warm failovers of redundant *Servers*, *Subscription* for *Events* shall require a Refresh operation. The *Client* shall initiate this *Refresh* operation.
- It is expected that for hot failovers of redundant *Servers*, *Subscriptions* for *Events* may require a *Refresh* operation. The *Server* shall issue a *RefreshRequiredEventType Event* if it is required.
- For transparent redundancy, a *Server* shall not require any action be performed by a *Client*.

6 AddressSpace organisation

6.1 General

The *AddressSpace* organisation described in this clause allows *Clients* to detect *Conditions* and *ConditionSources*. An additional hierarchy of *Object Nodes* that are notifiers may be established to define one or more areas; the *Client* can subscribe to specific areas to limit the *Event Notifications* sent by the *Server*. Additional examples can be found in Clause B.2.

6.2 EventNotifier and source hierarchy

HasNotifier and *HasEventSource References* are used to expose the hierarchical organization of *Event* notifying *Objects* and *ConditionSources*. An *Event* notifying *Object* represents typically an area of *Operator* responsibility. The definition of such an area configuration is outside the scope of this document. If areas are available, they shall be linked together and with the included *ConditionSources* using the *HasNotifier* and the *HasEventSource Reference Types*. The *Server Object* shall be the root of this hierarchy.

Figure 24 shows such a hierarchy. Note that *HasNotifier* is a subtype of *HasEventSource*. I.e. the target *Node* of a *HasNotifier Reference* (an *Event* notifying *Object*) can also be a *ConditionSource*. The *HasEventSource Reference* is used if the target *Node* is a *ConditionSource* but cannot be used as *Event* notifier. See IEC 62541-3 for the formal definition of these *Reference Types*.

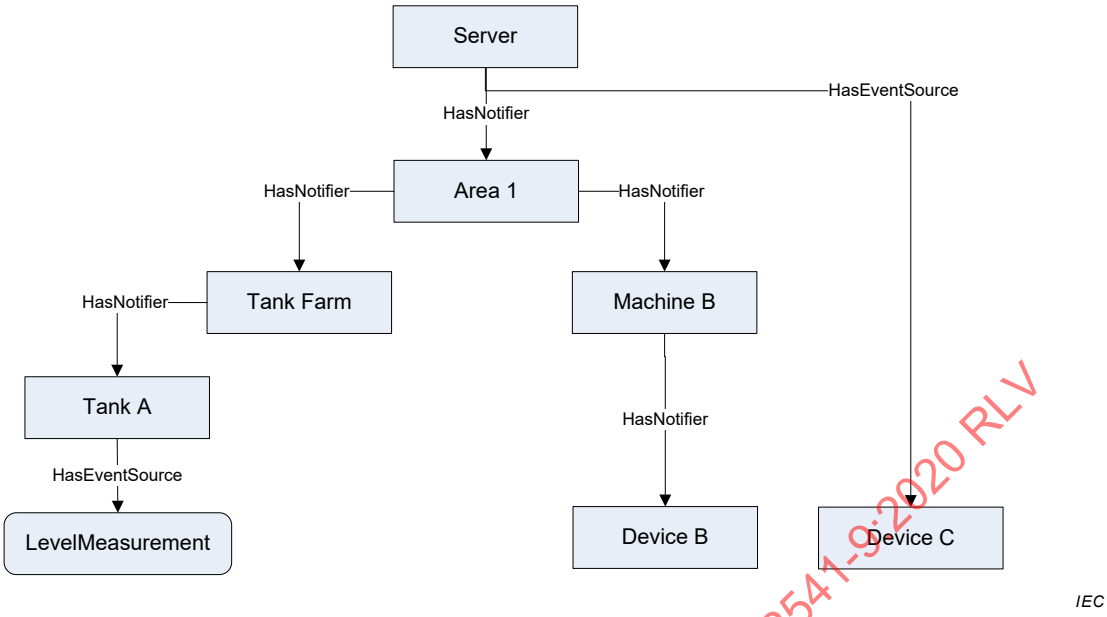


Figure 24 – Typical HasNotifier Hierarchy

6.3 Adding Conditions to the hierarchy

HasCondition is used to reference *Conditions*. The *Reference* is from a *ConditionSource* to a *Condition* instance or – if no instance is exposed by the *Server* – to the *ConditionType*.

Clients can locate *Conditions* by first browsing for *ConditionSources* following *HasEventSource References* (including subtypes like the *HasNotifier Reference*) and then browsing for *HasCondition References* from all target *Nodes* of the discovered *References*.

Figure 25 shows the application of the *HasCondition Reference* in a *HasNotifier* hierarchy. The *Variable* LevelMeasurement and the *Object* "Device B" *Reference Condition* instances. The *Object* "Tank A" *References* a *ConditionType* (MySystemAlarmType) indicating that a *Condition* exists but is not exposed in the *AddressSpace*.

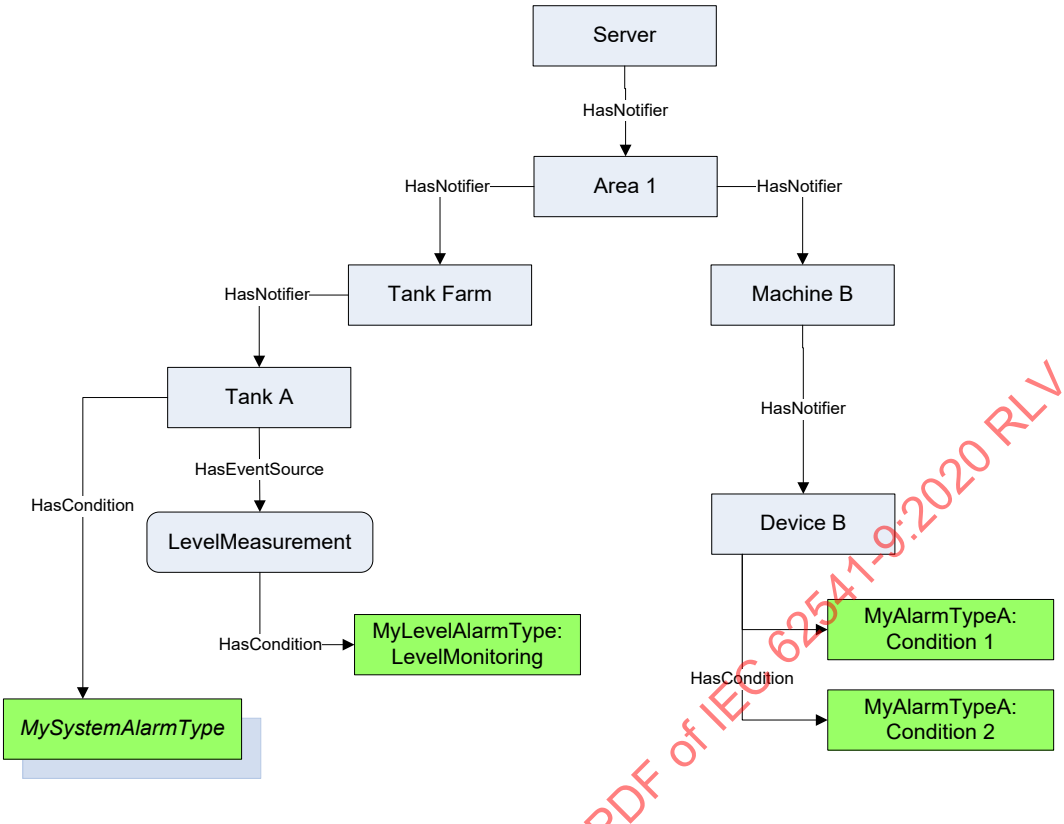


Figure 25 – Use of HasCondition in a HasNotifier hierarchy

6.4 Conditions in InstanceDeclarations

Figure 26 shows the use of the *HasCondition Reference* and the *HasEventSource Reference* in an *InstanceDeclaration*. They are used to indicate what *References* and *Conditions* are available on the instance of the *ObjectType*.

The use of the *HasEventSource Reference* in the context of *InstanceDeclarations* and *TypeDefinition Nodes* has no effect for *Event* generation.

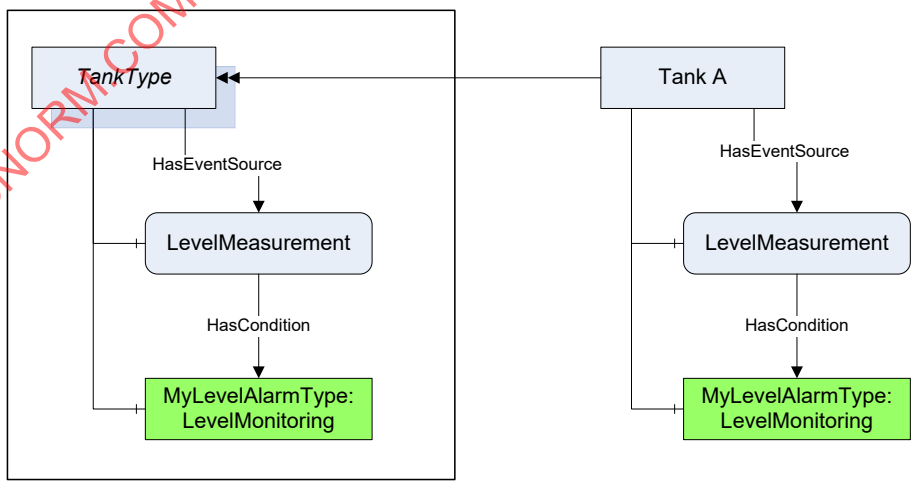


Figure 26 – Use of HasCondition in an InstanceDeclaration

6.5 Conditions in a VariableType

Use of *HasCondition* in a *VariableType* is a special use case since *Variables* (and *VariableTypes*) may not have *Conditions* as components. Figure 27 provides an example of this use case. Note that there is no component relationship for the "LevelMonitoring" *Alarm*. It is *Server*-specific whether and where they assign a *HasComponent Reference*.

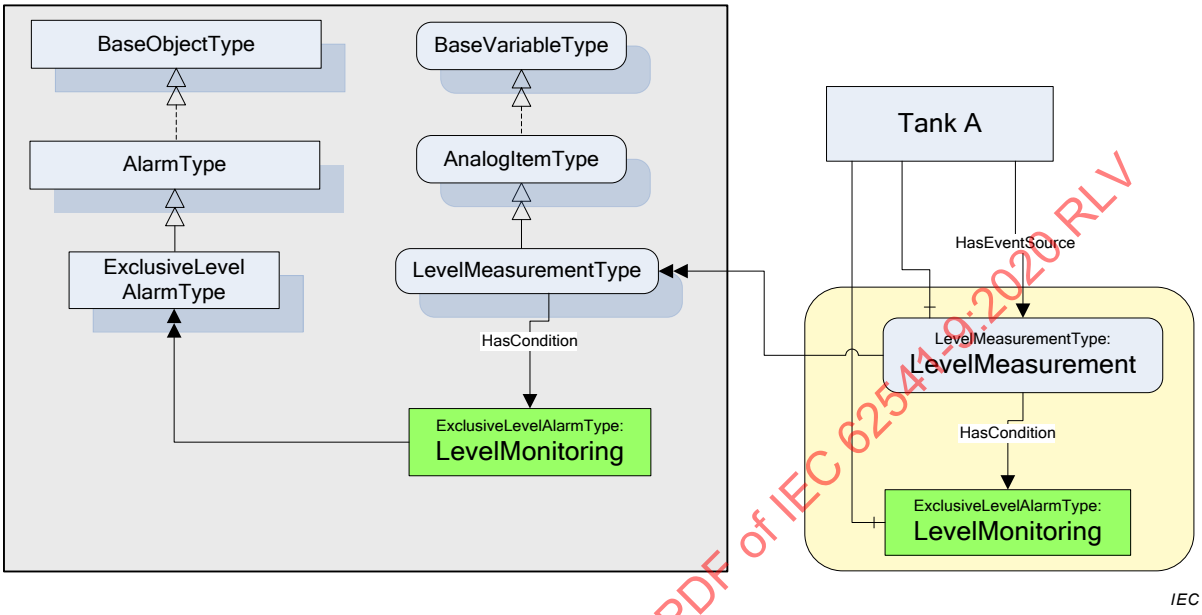


Figure 27 – Use of HasCondition in a VariableType

7 System State and alarms

7.1 Overview

The state of alarms is affected by the state of the process, equipment, system or plant. For example, when a tank is taken out of service, the level alarms associated with the tank would be no longer used, until the tank is returned to service. This clause describes *ReferenceTypes* that can be used by a *StateMachine* to indicate that a specific *Effect* on *Alarms* caused by the transition of a *StateMachine*. *StateMachines* that describe the state of a process, system or equipment can vary, but an example *StateMachine* is provided in Annex F.

7.2 HasEffectDisable

The *HasEffectDisable ReferenceType* is a concrete *ReferenceType* and can be used directly. It is a subtype of *HasEffect*.

The semantic of this *ReferenceType* is to point form a *Transition* to an *Alarm* that will be disabled.

- If the *Reference* is to an *Object* then all *Alarms* in the *HasNotifier* hierarchy below that *Object* are disabled,
- If the target is an *AlarmType* then all instances of that *AlarmType* in the *HasNotifier* hierarchy below the *Object* containing the *StateMachine* are disabled,
- If the target is an *Alarm* instance then the given *Alarm* instance is disabled.

The *SourceNode* of this *ReferenceType* shall be an *Object* of the *ObjectType TransitionType* or one of its subtypes. The *TargetNode* can be of an *Object* or *AlarmType*.

The representation of the *HasEffectDisable ReferenceType* in the *AddressSpace* is specified in Table 102.

Table 102 – HasEffectDisable ReferenceType

Attributes	Value		
BrowseName	*HasEffectDisable*		
InverseName	MayBeDisabledBy		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**

7.3 HasEffectEnable

The *HasEffectEnable ReferenceType* is a concrete *ReferenceType* and may be used directly. It is a subtype of *HasEffect*.

The semantic of this *ReferenceType* is to point form a *Transition* to an *Alarm* that will be enabled.

- If the *Reference* is to an *Object* then all *Alarms* in the *HasNotifier* hierarchy below that *Object* are enabled.
- If the target is an *AlarmType* then all instances of that *AlarmType* in the *HasNotifier* hierarchy below the *Object* containing the *StateMachine* are enabled.
- If the target is an *Alarm* instance then the given *Alarm* instance is enabled.

The *SourceNode* of this *ReferenceType* shall be an *Object* of the *ObjectType TransitionType* or one of its subtypes. The *TargetNode* can be of any *NodeClass*.

The representation of the *HasEffectenable ReferenceType* in the *AddressSpace* is specified in Table 103.

Table 103 – HasEffectEnable ReferenceType

Attributes	Value		
BrowseName	HasEffectEnable		
InverseName	MayBeEnabledBy		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**

7.4 HasEffectSuppress

The *HasEffectSuppress ReferenceType* is a concrete *ReferenceType* and may be used directly. It is a subtype of *HasEffect*.

The semantic of this *ReferenceType* is to point form a *Transition* to an *Alarm* that will be suppressed.

- If the reference is to an *Object* then all *Alarms* in the *EventNotifer* hierarchy below that *Object* are suppressed.
- If the target is an *AlarmType* then all instance of that AlarmType in the *HasNotifier* hierarchy below the *Object* containing the *StateMachine* are suppressed.
- If the target is an *Alarm* instance then the given *Alarm* instance is suppressed.

The *SourceNode* of this *ReferenceType* shall be an *Object* of the *ObjectType TransitionType* or one of its subtypes. The *TargetNode* can be of any *NodeClass*.

The representation of the *HasEffectSuppress ReferenceType* in the *AddressSpace* is specified in Table 104.

Table 104 – HasEffectSuppress ReferenceType

Attributes	Value		
BrowseName	HasEffectSuppress		
InverseName	MayBeSuppressedBy		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**

7.5 HasEffectUnsuppressed

The *HasEffectUnsuppressed ReferenceType* is a concrete *ReferenceType* and may be used directly. It is a subtype of *HasEffect*.

The semantic of this *ReferenceType* is to point form a *Transition* to an *Alarm* that will no longer be suppressed.

- If the *Reference* is to an *Object* then all *Alarms* in the *HasNotifier* hierarchy below that *Object* are removed from being suppressed.
- If the target is an *AlarmType* then all instance of that AlarmType are no longer suppressed below the *Object* containing the *StateMachine*.
- if the target is an *Alarm* instance then the given *Alarm* instance is no longer suppressed. No errors are logged if the *Alarm* was not suppressed.

The *SourceNode* of this *ReferenceType* shall be an *Object* of the *ObjectType TransitionType* or one of its subtypes. The *TargetNode* can be of any *NodeClass*.

The representation of the *HasEffectUnsuppress ReferenceType* in the *AddressSpace* is specified in Table 105.

Table 105 – HasEffectUnsuppress ReferenceType

Attributes	Value		
BrowseName	HasEffectUnsuppress		
InverseName	MayBeUnsuppressedBy		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**

8 Alarm metrics

8.1 Overview

The goal of a well-designed alarm system is to ensure that an *Operator* is made aware of issues, both critical and non-critical, but is not overwhelmed by alarms/alerts or other messages. When designing an alarm system, criteria are defined for alarm rates and general performance of the system at various levels (*Operator* station, plant area, overall system etc.). Evaluating the performance of an alarm system with regard to these design criteria requires the collection of alarm metrics. These metrics provide summaries of alarm rates and other alarm-related information.

This clause defines a standard structure for metrics. This structure may be implemented at multiple levels allowing a *Server* to collect metrics as needed. For example, an *Object* of this type might be added to the *Server Object* providing a summary of the *Alarm* performance for the entire *Server*. An instance might also be provided on an *Object* that includes a *HasNotifier* hierarchy, such as a tank *Object*. In this case, it would provide the summary of all of the *Alarms* that are part of the tank *HasNotifier* hierarchy.

8.2 AlarmMetricsType

This Object*Type* is used for metric information. The *ObjectType* is formally defined in Table 106.

Table 106 – AlarmMetricsType Definition

Attribute		Value			
BrowseName		AlarmMetricsType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the BaseObjectType defined in IEC 62541-5.					
HasComponent	Variable	AlarmCount	UInt32	BaseDataVariableType	Mandatory
HasComponent	Variable	StartTime	UtcTime	BaseDataVariableType	Mandatory
HasComponent	Variable	MaximumActiveState	Duration	BaseDataVariableType	Mandatory
HasComponent	Variable	MaximumUnAck	Duration	BaseDataVariableType	Mandatory
HasComponent	Variable	CurrentAlarmRate	Double	AlarmRateVariableType	Mandatory
HasComponent	Variable	MaximumAlarmRate	Double	AlarmRateVariableType	Mandatory
HasComponent	Variable	MaximumReAlarmCount	UInt32	BaseDataVariableType	Mandatory
HasComponent	Variable	AverageAlarmRate	Double	AlarmRateVariableType	Mandatory
HasComponent	Method	Reset			Mandatory

An instance of *AlarmMetricsType* can be added, with a *HasComponent* reference, to any *Object* that has its "SubscribeToEvents" bit set within the *EventNotifier Attribute*. It will collect the *Alarm* metrics for all *Alarm* sources assigned to this notifier *Object*. For example, if *Alarm* metrics are desired for Tank A *Object* (see Figure B.3) that is in the *HasNotifier* hierarchy than an instance of this object would be referenced by the Tank A object. When this object is associated with the *Server Object* it will report *Alarm* metrics for the entire *Server*.

AlarmCount is the total count of *Alarm*s since the last restart of the system or reset of this counter.

StartTime is the time at which the *Server* started or the time of the last *Reset* Method invocation, whichever is later.

MaximumActiveState is the maximum time for which an *Alarm* was in the active state.

MaximumUnAck is the maximum time for which an *Alarm* was in the unacknowledged state.

CurrentAlarmRate is the sum of *Alarms* that occurred in the last *Rate* number of minutes (see 8.3). This sum should not include nuisance Alarms (i.e. chattering alarms). It is updated every *Rate* number of minutes.

MaximumAlarmRate is the maximum *Alarm* rate detected since the start of the *Server*, where the rate is calculated as for *CurrentAlarmRate*.

MaximumReAlarmCount is the maximum *ReAlarmCount* for any *Alarm*.

AverageAlarmRate is the average *Alarm* rate since the start of the *Server* or the last invocation of *Reset Method*, where the rate is calculated as for *CurrentAlarmRate*.

Reset is a *Method* that will reset all of the counters, rates or times in this *Object*

8.3 AlarmRateVariableType

This variable type provides a unit field for the rate for which the *Alarm* diagnostic applies.

Table 107 – AlarmRateVariableType definition

Attribute		Value			
BrowseName		AlarmRateVariableType			
IsAbstract		False			
ValueRank		Scalar			
DataType		Double			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	Rate	UInt16	PropertyType	Mandatory

Rate is the number of minutes over which the item is calculated.

8.4 Reset Method

The *Reset Method* is used to reset all of the counters, rates and time in the *Object.*

Signature

```
Reset();
```

Method Result Codes in Table 108 (defined in Call Service).

Table 108 – Suppress result codes

Result Code	Description
Bad_MethodInvalid	The MethodId provided does not correspond to the ObjectId provided. See IEC 62541-4 for the general description of this result code.
Bad_NodeIdInvalid	Used to indicate that the specified *ObjectId* is not valid. See IEC 62541-4 for the general description of this result code.

Comments

The Reset Method will clear all setting in the diagnostic object and initialize them to zero.

Table 109 specifies the *AddressSpace* representation for the *Reset Method*.

Table 109 – Reset Method AddressSpace definition

Attribute	Value				
BrowseName	Reset				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditUpdateMethodEventType	Defined in IEC 62541-5.		

Annex A
(informative)

Recommended localized names

A.1 Recommended state names for TwoState variables

A.1.1 LocaleId "en"

The recommended state display names for the LocaleId "en" are listed in Table A.1 and Table A.2.

Table A.1 – Recommended state names for LocaleId "en"

Condition Type	State Variable	False State Name	True State Name
ConditionType	EnabledState	Disabled	Enabled
DialogConditionType	DialogState	Inactive	Active
AcknowledgeableConditionType	AckedState	Unacknowledged	Acknowledged
	ConfirmedState	Unconfirmed	Confirmed
AlarmConditionType	ActiveState	Inactive	Active
	SuppressedState	Unsuppressed	Suppressed
	OutOfServiceState	In Service	Out of Service
	SilenceState	Silenced	Not Silenced
	LatchedState	Latched	Unlatched
NonExclusiveLimitAlarmType	HighHighState	HighHigh inactive	HighHigh active
	HighState	High inactive	High active
	LowState	Low inactive	Low active
	LowLowState	LowLow inactive	LowLow active

Table A.2 – Recommended display names for LocaleId "en"

Condition Type	Browse Name	display name
Shelved	Unshelved	Unshelved
	TimedShelved	Timed Shelved
	OneShotShelved	One Shot Shelved
Exclusive	HighHigh	HighHigh
	High	High
	Low	Low
	LowLow	LowLow

A.1.2 LocaleId "de"

The recommended state display names for the LocaleId "de" are listed in Table A.3 and Table A.4.

Table A.3 – Recommended state names for LocaleId "de"

Condition Type	State Variable	False State Name	True State Name
ConditionType	EnabledState	Ausgeschaltet	Eingeschaltet
DialogConditionType	DialogState	Inaktiv	Aktiv
AcknowledgeableConditionType	AckedState	Unquittiert	Quittiert
	ConfirmedState	Unbestätigt	Bestätigt
AlarmConditionType	ActiveState	Inaktiv	Aktiv
	SuppressedState	Nicht unterdrückt	Unterdrückt
	OutOfServiceState	In Betrieb	Außer Betrieb
	SilenceState	Stumm	Nicht Stumm
	LatchedState	Verriegelt	Entriegelt
NonExclusiveLimitAlarmType	HighHighState	HighHigh inaktiv	HighHigh aktiv
	HighState	High inaktiv	High aktiv
	LowState	Low inaktiv	Low aktiv
	LowLowState	LowLow inaktiv	LowLow aktiv

Table A.4 – Recommended display names for LocaleId "de"

Condition Type	Browse Name	display name
Shelved	Unshelved	Nicht zurückgestellt
	TimedShelved	Befristet zurückgestellt
	OneShotShelved	Einmalig zurückgestellt
Exclusive	HighHigh	HighHigh
	High	High
	Low	Low
	LowLow	LowLow

A.1.3 LocaleId "fr"

The recommended state display names for the LocaleId "fr" are listed in Table A.5 and Table A.6.

Table A.5 – Recommended state names for LocaleId "fr"

Condition Type	State Variable	False State Name	True State Name
ConditionType	EnabledState	Hors Service	En Service
DialogConditionType	DialogState	Inactive	Active
AcknowledgeableConditionType	AckedState	Non-acquitté	Acquitté
	ConfirmedState	Non-Confirmé	Confirmé
AlarmConditionType	ActiveState	Inactive	Active
	SuppressedState	Présent	Supprimé
	OutOfServiceState	En Fonction	Hors Fonction
	SilenceState	Muette	Non-Muette
	LatchedState		
NonExclusiveLimitAlarmType	HighHighState	Très Haute Inactive	Très Haute Active
	HighState	Haute inactive	Haute active
	LowState	Basse inactive	Basse active
	LowLowState	Très basse inactive	Très basse active

Table A.6 – Recommended display names for LocaleId "fr"

Condition Type	Browse Name	display name
Shelved	Unshelved	Surveillée
	TimedShelved	Mise de coté temporelle
	OneShotShelved	Mise de coté unique
Exclusive	HighHigh	Très haute
	High	Haute
	Low	Basse
	LowLow	Très basse

A.2 Recommended dialog response options

The recommended *Dialog* response option names in different locales are listed in Table A.7.

Table A.7 – Recommended dialog response options

Locale "en"	Locale "de"	Locale "fr"
Ok	OK	Ok
Cancel	Abbrechen	Annuler
Yes	Ja	Oui
No	Nein	Non
Abort	Abbrechen	Abandonner
Retry	Wiederholen	Réessayer
Ignore	Ignorieren	Ignorer
Next	Nächster	Prochain
Previous	Vorheriger	Precedent

Annex B
(informative)

Examples

B.1 Examples for Event sequences from Condition instances

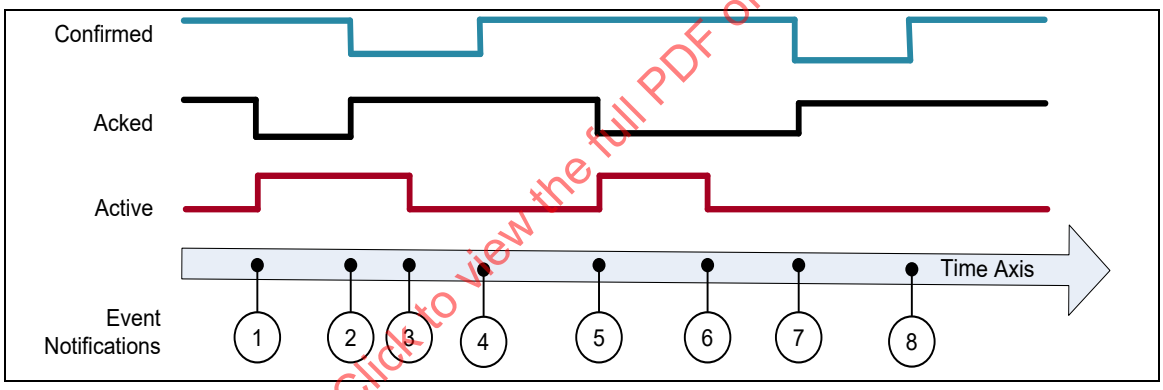
B.1.1 Overview

The following examples show the *Event* flow for typical *Alarm* situations. Table B.1 and Table B.2 list the value of state *Variables* for each *Event Notification*.

B.1.2 Server maintains current state only

This example is for *Servers* that do not support previous states and therefore do not create and maintain *Branches* of a single *Condition*.

Figure B.1 shows an *Alarm* as it becomes active and then inactive and also the acknowledgement and confirmation cycles. Table B.1 lists the values of the state *Variables*. All *Events* are coming from the same *Condition* instance and therefore have the same *ConditionId*.



IEC

Figure B.1 – Single state example

Table B.1 – Example of a Condition that only keeps the latest state

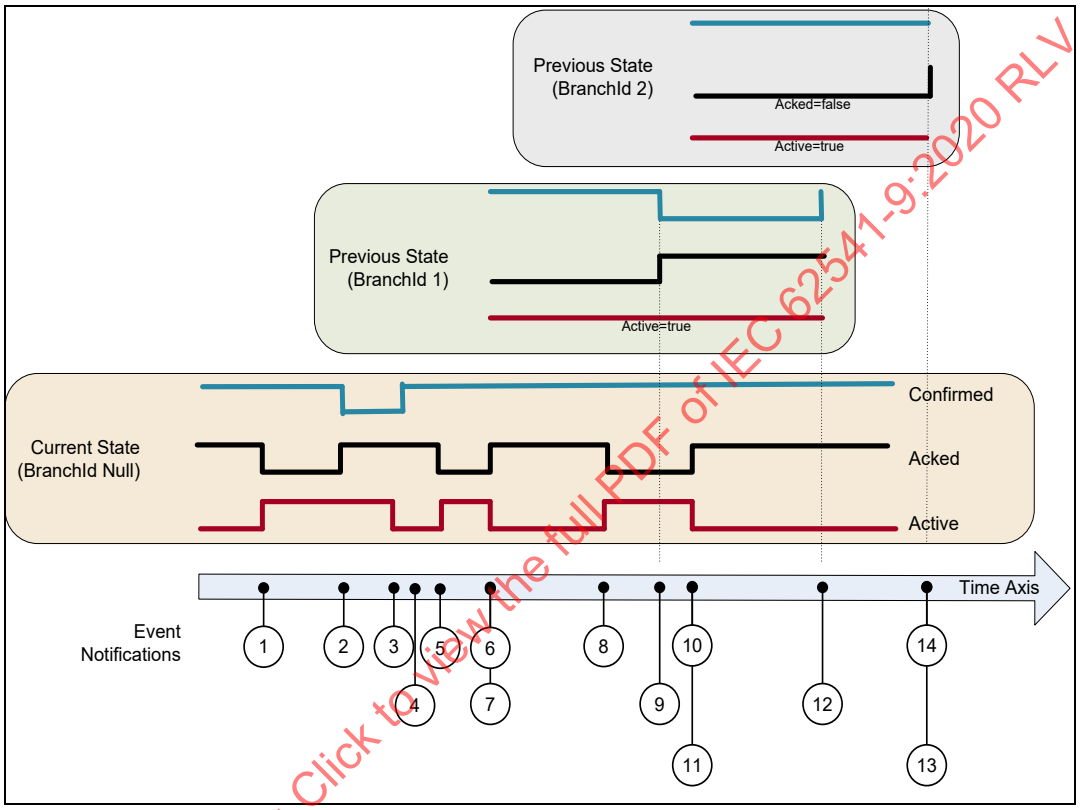
EventId	BranchId	Active	Acked	Confirmed	Retain	Description
-*)	NULL	False	True	True	False	Initial state of *Condition*.
1	NULL	True	False	True	True	*Alarm* goes active.
2	NULL	True	True	False	True	*Condition* acknowledged Confirm required
3	NULL	False	True	False	True	*Alarm* goes inactive.
4	NULL	False	True	True	False	*Condition* confirmed
5	NULL	True	False	True	True	*Alarm* goes active.
6	NULL	False	False	True	True	*Alarm* goes inactive.
7	NULL	False	True	False	True	*Condition* acknowledged, Confirm required.
8	NULL	False	True	True	False	*Condition* confirmed.

*) The first row is included to illustrate the initial state of the *Condition*. This state will not be reported by an *Event*.

B.1.3 Server maintains previous states

This example is for *Servers* that are able to maintain previous states of a *Condition* and therefore create and maintain *Branches* of a single *Condition*.

Figure B.2 illustrates the use of branches by a *Server* requiring acknowledgement of all transitions into *Active* state, not just the most recent transition. In this example no acknowledgement is required on a transition into an inactive state. Table B.2 lists the values of the state *Variables*. All *Events* are coming from the same *Condition* instance and have therefore the same *ConditionId*.



IEC

Figure B.2 – Previous state example

Table B.2 – Example of a *Condition* that maintains previous states via branches

EventId	BranchId	Active	Acked	Confirmed	Retain	Description
[a]	NULL	False	True	True	False	Initial state of *Condition*.
1	NULL	True	False	True	True	Alarm goes active.
2	NULL	True	True	True	True	Condition acknowledged requires Confirm
3	NULL	False	True	False	True	*Alarm* goes inactive.
4	NULL	False	True	True	False	Confirmed
5	NULL	True	False	True	True	*Alarm* goes active.
6	NULL	False	True	True	True	*Alarm* goes inactive.
7	1	True	False	True	True [b]	Prior state needs acknowledgment. Branch #1 created.
8	NULL	True	False	True	True	*Alarm* goes active again.
9	1	True	True	False	True	Prior state acknowledged. Confirm required.
10	NULL	False	True	True	True [b]	*Alarm* goes inactive again.
11	2	True	False	True	True	Prior state needs acknowledgment. Branch #2 created.
12	1	True	True	True	False	Prior state confirmed. Branch #1 deleted.
13	2	True	True	True	False	Prior state acknowledged, Auto Confirmed by system. Branch #2 deleted. The confirmation of the previous transition allows the system to auto confirm this transition
14	NULL	False	True	True	False	No longer of interest.

[a] *The first row* is included to illustrate the initial state of the *Condition*. This state will not be reported by an *Event*.

Notes on specific situations shown with this example:

If the current state of the *Condition* is acknowledged then the *Acked* flag is set and the new state is reported (*Event* #2). If the *Condition* state changes before it can be acknowledged (*Event* #6) then a branch state is reported (*Event* #7). Timestamps for the *Event*s #6 and #7 is identical.

The branch state can be updated several times (*Event*s #9) before it is cleared (*Event* #12).

A single *Condition* can have many branch states active (*Event*s #11).

[b] *It is recommended* as in this table to leave Retain=True as long as there exist previous states (branches).

B.2 AddressSpace examples

This clause provides additional examples for the use of *HasNotifier*, *HasEventSource* and *HasCondition References* to expose the organization of areas and sources with their associated *Conditions*. This hierarchy is additional to a hierarchy provided with *Organizes* and *Aggregates References*.

Figure B.3 illustrates the use of the *HasCondition Reference* with *Condition* instances.

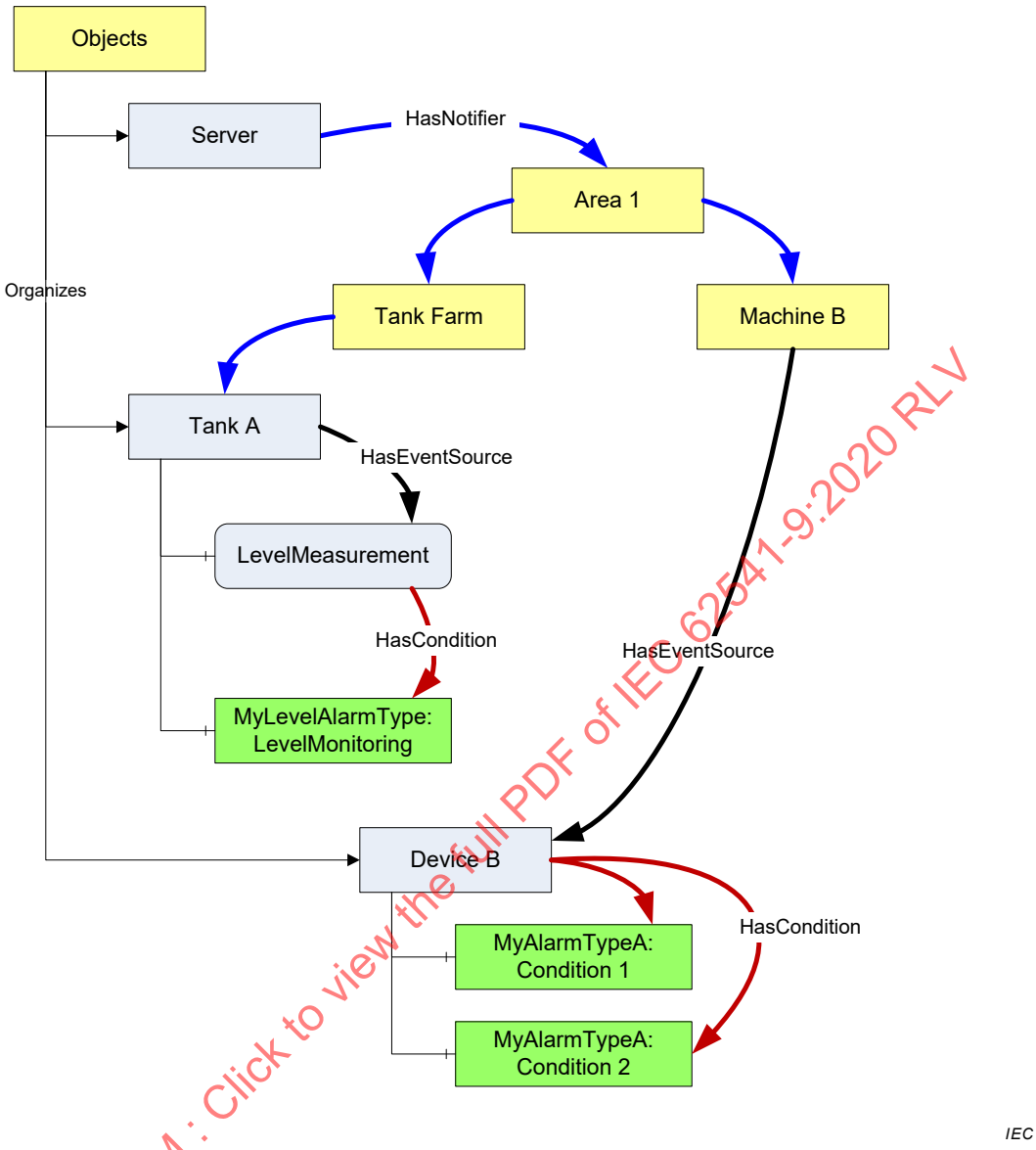


Figure B.3 – HasCondition used with Condition instances

In systems where *Conditions* are not available as instances, the *ConditionSource* may reference the *ConditionTypes* instead. This is illustrated with the example in Figure B.4.

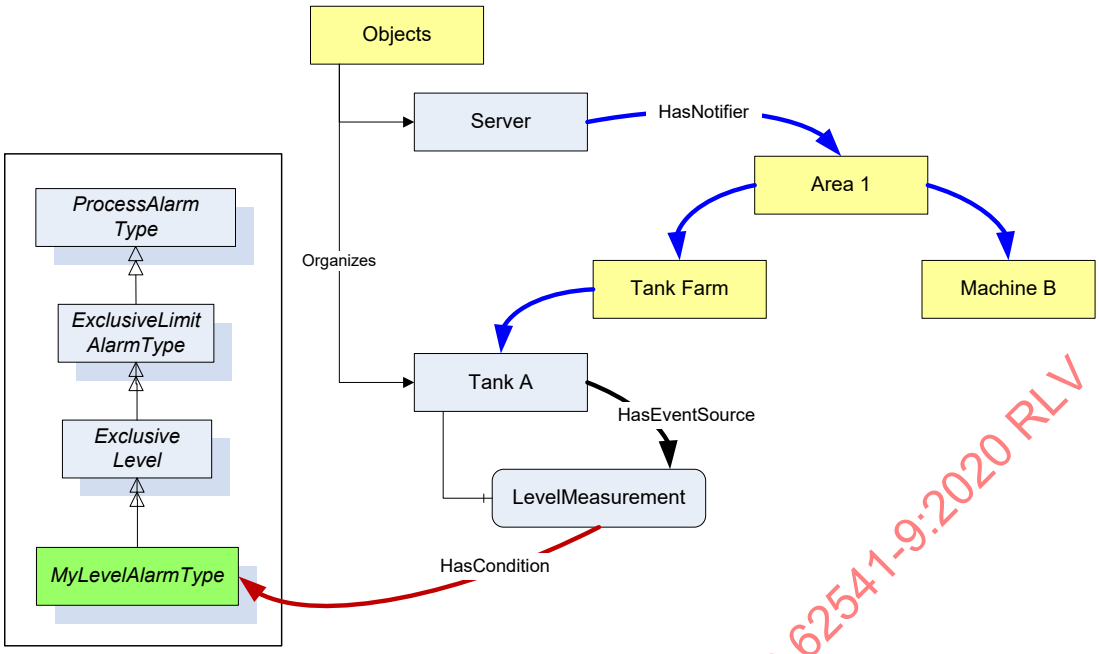


Figure B.4 – HasCondition reference to a Condition type

Figure B.5 provides an example where the *HasCondition Reference* is already defined in the *Type* system. The *Reference* may point to a *Condition Type* or to an instance. Both variants are shown in this example. A *Reference* to a *Condition Type* in the *Type* system will result in a *Reference* to the same *Type Node* in the instance.

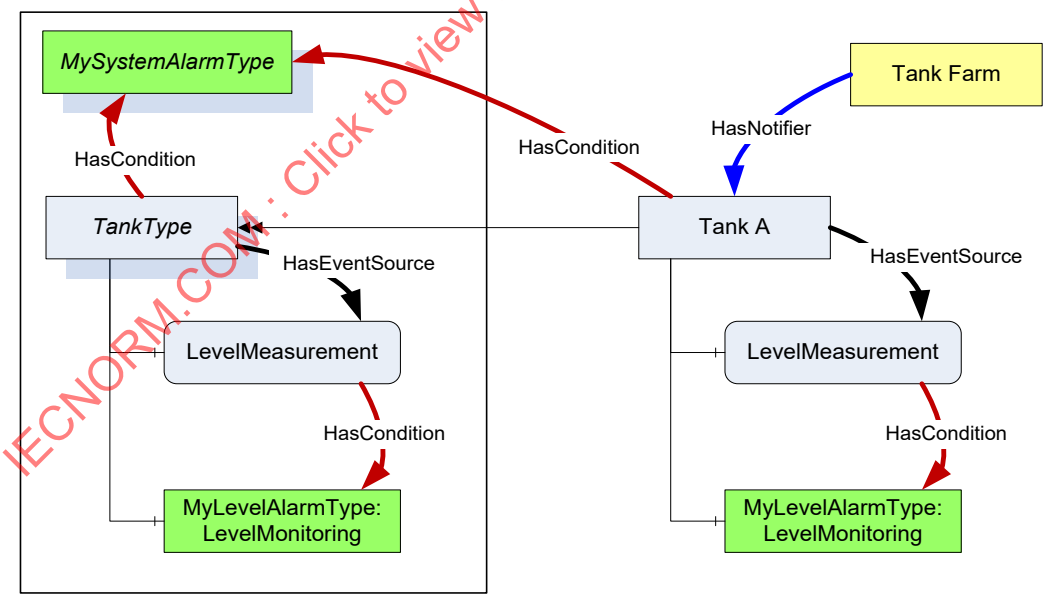


Figure B.5 – HasCondition used with an instance declaration

Annex C
(informative)

Mapping to EEMUA

Table C.1 lists EEMUA terms and how OPC UA terms maps to them.

Table C.1 – EEMUA Terms

EEMUA Term	OPC UA Term	EEMUA Definition
Accepted	Acknowledged=True	An *Alarm* is accepted when the *Operator* has indicated awareness of its presence. In OPC UA, this may be accomplished with the *Acknowledge Method*.
Active Alarm	Active = True	An *Alarm Condition* which is on (i.e. limit has been exceeded and *Condition* continues to exist).
Alarm Message	Message Property (defined in IEC 62541-5.)	Test information presented to the *Operator* that describes the *Alarm Condition*.
Alarm Priority	Severity Property (defined in IEC 62541-5.)	The ranking of *Alarms* by severity and response time.
Alert	-	A lower priority *Notification* than an *Alarm* that has no serious consequence if ignored or missed. In some Industries also referred to as a "Prompt" or "Warning". No direct mapping! In UA the concept of *Alerts* may be accomplished by the use of severity. E.g., *Alarms* that have a severity below 50 may be considered as *Alerts*.
Cleared	Active = False	An *Alarm* state that indicates the *Condition* has returned to normal.
Disable	Enabled = False	An *Alarm* is disabled when the system is configured such that the *Alarm* will not be generated even though the base *Alarm Condition* is present.
Prompt	Dialog	A request from the control system that the *Operator* perform some process action that the system cannot perform or that requires *Operator* authority to perform.
Raised	Active = True	An *Alarm* is *Raised* or initiated when the *Condition* creating the *Alarm* has occurred.
Release	OneShotShelving	A "release" is a facility that may be applied to a standing (UA = active) *Alarm* in a similar way to which *Shelving* is applied. A released *Alarm* is temporarily removed from the *Alarm* list and put on the shelf. There is no indication to the *Operator* when the *Alarm* clears, but it is taken off the shelf. Hence, when the *Alarm* is raised again it appears on the *Alarm* list in the normal way.
Reset	Retain=False	An *Alarm* is Reset when it is in a state that can be removed from the Display list. OPC UA includes *Retain* flag which as part of its definition states: "when a *Client* receives an *Event* with the *Retain* flag set to False, the *Client* should consider this as a *Condition*/Branch that is no longer of interest, in the case of a "current *Alarm* display" the *Condition*/Branch would be removed from the display"
Shelving	Shelving	*Shelving* is a 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. A Shelved *Alarm* will be removed from the list and will not re-annunciate until un-shelved.
Standing	Active = True	An *Alarm* is *Standing* whilst the *Condition* persists (*Raised* and *Standing* are often used interchangeably).
Suppress	Suppress	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.
Unaccepted	Acknowledged = False	An *Alarm* is accepted when the *Operator* has indicated awareness of its presence. It is unaccepted until this has been done.

Annex D
(informative)

Mapping from OPC A&E to OPC UA A&C

D.1 Overview

Serving as a bridge between COM and OPC UA components, the Alarm and *Event*s proxy and wrapper enable existing A&E COM *Clients* and *Servers* to connect to UA *Alarms* and *Conditions* components.

Simply stated, there are two aspects to the migration strategy. The first aspect enables a UA *Alarms* and *Conditions Client* to connect to an existing Alarms and *Event*s COM *Server* via a UA *Server* wrapper. This wrapper is notated from this point forward as the A&E COM UA Wrapper. The second aspect enables an existing Alarms and *Event*s COM *Client* to connect to a UA *Alarms* and *Conditions Server* via a COM proxy. This proxy is notated from this point forward as the A&E COM UA Proxy.

An Alarms and *Event*s COM *Client* is notated from this point forward as A&E COM *Client*.

A UA *Alarms* and *Conditions Server* is notated from this point forward as UA A&C *Server*.

The mappings describe generic A&E COM interoperability components. It is recommended that vendors use this mapping if they develop their own components; however, some applications may benefit from vendor-specific mappings.

D.2 Alarms and Events COM UA wrapper

D.2.1 Event Areas

Event Areas in the A&E COM *Server* are represented in the A&E COM UA Wrapper as *Objects* with a *TypeDefinition* of *BaseObjectType*. The EventNotifier *Attribute* for these *Objects* always has the SubscribeToEvents flag set to True.

The root Area is represented by an *Object* with a *BrowseName* that depends on the UA *Server*. It is always the target of a HasNotifier *Reference* from the *Server Node*. The root Area allows multiple A&E COM *Server*s to be wrapped within a single UA *Server*.

The Area hierarchy is discovered with the BrowseOPCAreas and the GetQualifiedAreaName *Method*s. The Area name returned by BrowseOPCAreas is used as the *BrowseName* and *DisplayName* for each Area *Node*. The QualifiedAreaName is used to construct the *NodeId*. The NamespaceURI qualifying the *NodeId* and *BrowseName* is a unique URI assigned to the combination of machine and COM *Server*.

Each Area is the target of *HasNotifier Reference* from its parent Area. It may be the source of one or more HasNotifier *Reference*s to its child Areas. It may also be a source of a *HasEventSource Reference* to any sources in the Area.

The A&E COM *Server* may not support filtering by Areas. If this is the case, then no Area *Node*s are shown in the UA *Server* address space. Some implementations could use the AREAS *Attribute* to provide filtering by Areas within the A&E COM UA Wrapper.

D.2.2 Event sources

Event Sources in the A&E COM *Server* are represented in the A&E COM UA Wrapper as *Objects* with a *TypeDefinition* of *BaseObjectType*. If the A&E COM *Server* supports source filtering then the SubscribeToEvents flag is True and the Source is a target of a HasNotifier *Reference*. If source filtering is not supported the SubscribeToEvents flag is False and the Source is a target of a HasEventSource *Reference*.

The Sources are discovered by calling BrowseOPCAreas and the GetQualifiedSourceName *Method*s. The Source name returned by BrowseOPCAreas is used as the BrowseName and DisplayName. The QualifiedSourceName is used to construct the *NodeId*. *Event* Source *Node*s are always targets of a HasEventSource *Reference* from an Area.

D.2.3 Event categories

Event Categories in the A&E COM *Server* are represented in the UA *Server* as *ObjectTypes* which are subtypes of *BaseEventType*. The *BrowseName* and *DisplayName* of the *ObjectType Node* for Simple and Tracking *Event* Types are constructed by appending the text 'EventType' to the Description of the *Event* Category. For *Condition Event* Types the text 'AlarmType' is appended to the *Condition* Name.

These *ObjectType Node*s have a super type which depends on the A&E *Event* Type, the *Event* Category Description and the *Condition* Name; however, the best mapping requires knowledge of the semantics associated with the *Event* Categories and *Condition* Names. If an A&E COM UA Wrapper does not know these semantics then Simple *Event* Types are subtypes of *BaseEventType*, Tracking *Event* Types are subtypes of *AuditEventType* and *Condition Event* Types are subtypes of the *AlarmType*. Table D.1 defines mappings for a set of "well known" Category description and *Condition* Names to a standard super type.

Table D.1 – Mapping from standard Event categories to OPC UA Event types

COM A&E Event Type	Category Description	Condition Name	OPC UA EventType
Simple	---	---	BaseEventType
Simple	Device Failure	---	DeviceFailureEventType
Simple	System Message	---	SystemEventType
Tracking	---	---	AuditEventType
Condition	---	---	AlarmType
Condition	Level	---	LimitAlarmType
Condition	Level	PVLEVEL	ExclusiveLevelAlarmType
Condition	Level	SPLEVEL	ExclusiveLevelAlarmType
Condition	Level	HI HI	NonExclusiveLevelAlarmType
Condition	Level	HI	NonExclusiveLevelAlarmType
Condition	Level	LO	NonExclusiveLevelAlarmType
Condition	Level	LO LO	NonExclusiveLevelAlarmType
Condition	Deviation	---	NonExclusiveDeviationAlarmType
Condition	Discrete	---	DiscreteAlarmType
Condition	Discrete	CFN	OffNormalAlarmType
Condition	Discrete	TRIP	TripAlarmType

There is no generic mapping defined for A&E COM sub-*Conditions*. If an *Event* Category is mapped to a *LimitAlarmType* then the sub *Condition* name in the *Event* shall be used to set the state of a suitable State *Variable*. For example, if the sub-*Condition* name is "HI HI" then that means the *HighHigh* state for the *LimitAlarmType* is active

For *Condition Event* Types, the *Event* Category is also used to define subtypes of *BaseConditionClassType*.

Figure D.1 illustrates how *ObjectType Nodes* created from the *Event* Categories and *Condition* Names are placed in the standard OPC UA *HasNotifier* hierarchy.

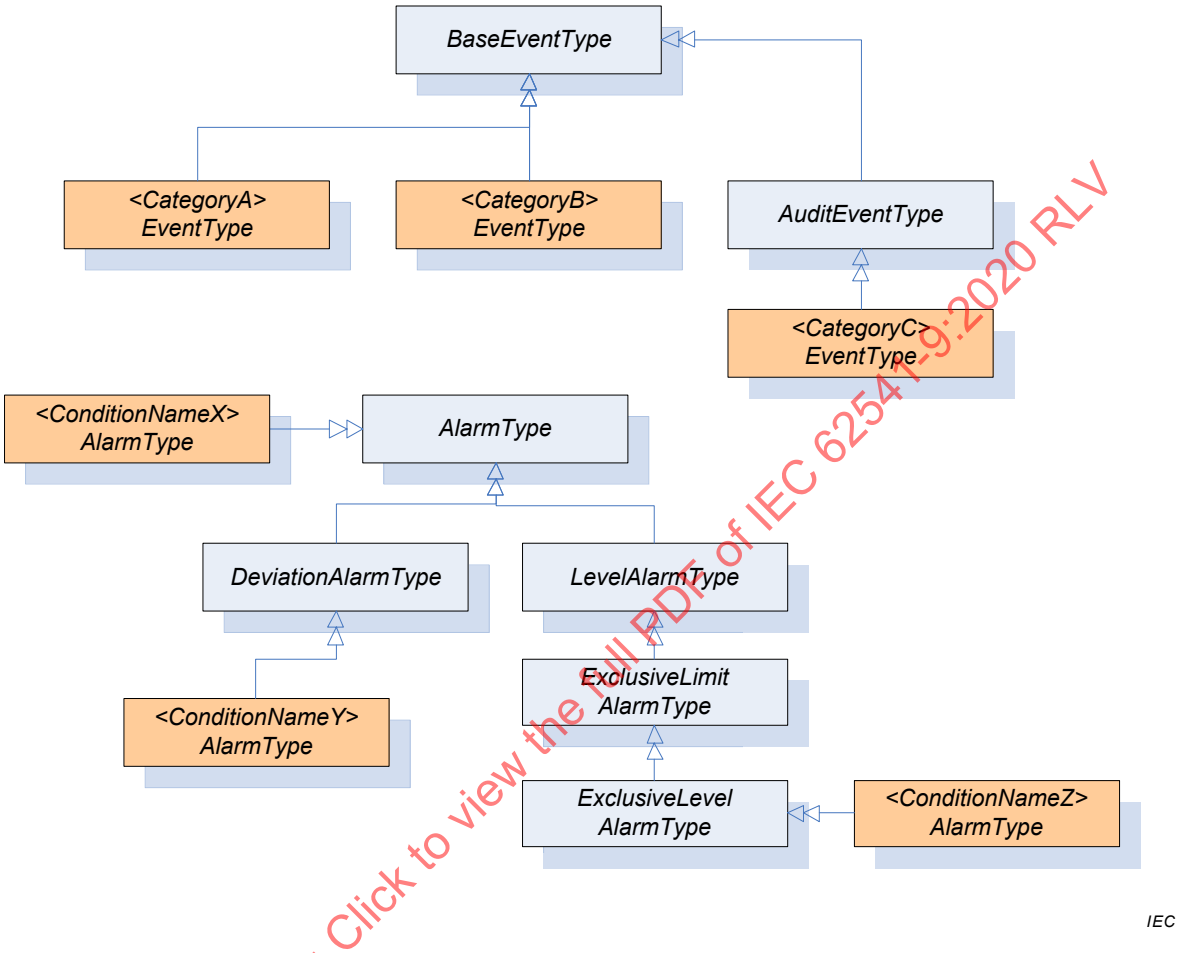


Figure D.1 – The type model of a wrapped COM A&E server

D.2.4 Event attributes

Event Attributes in the A&E COM *Server* are represented in the UA *Server* as *Variables* which are targets of *HasProperty References* from the *ObjectTypes* which represent the *Event* Categories. The *BrowseName* and *DisplayName* are the description for the *Event Attribute*. The data type of the *Event Attribute* is used to set *DataType* and *ValueRank*. The *NodeId* is constructed from the EventCategoryId, ConditionName and the AttributeId.

D.2.5 Event subscriptions

The A&E COM UA Wrapper creates a *Subscription* with the COM AE *Server* the first time a *MonitoredItem* is created for the *Server Object* or one of the *Nodes* representing Areas. The Area filter is set based on the *Node* being monitored. No other filters are specified.

If all *MonitoredItems* for an Area are disabled then the *Subscription* will be deactivated.

The *Subscription* is deleted when the last *MonitoredItem* for the *Node* is deleted.

When filtering by Area, the A&E COM UA Wrapper needs to add two Area filters: one based on the QualifiedAreaName which forms the *NodeId* and one with the text '/*' appended to it. This ensures that *Events* from sub areas are correctly reported by the COM AE *Server*.

A simple A&E COM UA Wrapper will always request all *Attributes* for all *Event* Categories when creating the *Subscription*. A more sophisticated wrapper may look at the *EventFilter* to determine which *Attributes* are actually used and only request those.

Table D.2 lists how the fields in the ONEVENTSTRUCT which are used by the A&E COM UA Wrapper are mapped to UA BaseEventType *Variable*s.

Table D.2 – Mapping from ONEVENTSTRUCT fields to UA BaseEventType Variables

UA Event Variable	ONEVENTSTRUCT Field	Notes
EventId	szSource szConditionName ftTime ftActiveTime dwCookie	A ByteString constructed by appending the fields together.
EventType	dwEventType dwEventCategory szConditionName	The NodeId for the corresponding *ObjectType Node*. The szConditionName maybe omitted by some implementations.
SourceNode	szSource	The *NodeId* of the corresponding Source *Object Node*.
SourceName	szSource	-
Time	ftTime	-
ReceiveTime	-	Set when the *Notification* is received by the wrapper.
LocalTime	-	Set based on the clock of the machine running the wrapper.
Message	szMessage	"Locale" is the default locale for the COM AE *Server*.
Severity	dwSeverity	-

Table D.3 lists how the fields in the ONEVENTSTRUCT which are used by the A&E COM UA Wrapper are mapped to UA *AuditEventType Variable*s.

Table D.3 – Mapping from ONEVENTSTRUCT fields to UA AuditEventType Variables

UA Event Variable	ONEVENTSTRUCT Field	Notes
ActionTimeStamp	ftTime	Only set for tracking *Event*s.
Status	-	Always set to True.
ServerId	-	Set to the COM AE *Server* NamespaceURI
ClientAuditEntryId	-	Not set.
ClientUserId	szActorID	-

Table D.4 lists how the fields in the ONEVENTSTRUCT which are used by the A&E COM UA Wrapper are mapped to UA AlarmType Variables.

Table D.4 – Mapping from ONEVENTSTRUCT fields to UA AlarmType Variables

UA Event Variable	ONEVENTSTRUCT Field	Notes
ConditionClassId	dwEventType	Set to the *NodeId* of the *ConditionClassType* for the *Event* Category of a *Condition Event* Type. Set to the *NodeId* of *BaseConditionClassType Node* for non-*Condition Event* Types.
ConditionClassName	dwEventType	Set to the *BrowseName* of the *ConditionClassType* for the *Event* Category of Condition *Event* Type. To set "BaseConditionClass" non-*Condition Event* Types.
ConditionName	szConditionName	-
BranchId	-	Always set to NULL.
Retain	wNewState	Set to True if the OPC_CONDITION_ACKED bit is not set or OPC_CONDITION_ACTIVE bit is set.
EnabledState	wNewState	Set to "Enabled" or "Disabled"
EnabledState.Id	wNewState	Set to True if OPC_CONDITION_ENABLED is set
EnabledState. EffectiveDisplayName	wNewState	A string constructed from the bits in the wNewState flag. The following rules are applied in order to select the string: "Disabled" if OPC_CONDITION_ENABLED is not set. "Unacknowledged" if OPC_CONDITION_ACKED is not set. "Active" if OPC_CONDITION_ACKED is set. "Enabled" if OPC_CONDITION_ENABLED is set.
Quality	wQuality	The COM DA Quality converted to a UA StatusCode.
Severity	dwSeverity	Set based on the last *Event* received for the *Condition* instance. Set to the current value if the last *Event* is not available.
Comment	-	The value of the ACK_COMMENT *Attribute*
ClientUserId	szActorID	-
AckedState	wNewState	Set to "Acknowledged" or "Unacknowledged "
AckedState.Id	wNewState	Set to True if OPC_CONDITION_ACKED is set
ActiveState	wNewState	Set to "Active" or "Inactive "
ActiveState.Id	wNewState	Set to True if OPC_CONDITION_ACTIVE is set
ActiveState.TransitionTime	ftActiveTime	This time is set when the *ActiveState* transitions from False to True. NOTE Additional logic applies to exclusive limit alarms, in that the LimitState.TransitionTime also needs to be set, but this is set each time a limit is crossed (multiple limits might exist). For the initial transition to True the ftActiveTime is used for both LimitState.TransitionTime and ActiveState.TransitionTime. For subsequent transition the ActiveState.Transition time does not change, but the LimitState.TransitionTime will be updated with the new ftActiveTime. For example, if an alarm has Hi and HiHi limits, when the Hi limit is crossed and the alarm goes active the FTActiveTime is used for both times, but when the HiHi limit is later crossed, the FtActiveTime is only be used for the LimitState.TransitionTime. NOTE The ftActiveTime is part of the key for identifying the unique event in the A&E server and needs to be saved for processing any commands back to the A&E Server.

The A&C *Condition* Model defines other optional *Variable*s which are not needed in the A&E COM UA Wrapper. Any additional fields associated with *Event Attributes* are also reported.

D.2.6 Condition instances

Condition instances do not appear in the UA *Server* address space. *Condition*s may be acknowledged by passing the EventId to the *Acknowledge Method* defined on the *AcknowledgeableConditionType*.

*Condition*s may not be enabled or disabled via the COM A&E Wrapper.

D.2.7 Condition Refresh

The COM A&E Wrapper does not store the state of *Condition*s. When *ConditionRefresh* is called the *Refresh Method* is called on all COM AE *Subscription*s associated with the *ConditionRefresh* call. The wrapper needs to wait until it receives the call back with the bLastRefresh flag set to True in the OnEvent call before it can tell the UA *Client* that the *Refresh* has completed.

D.3 Alarms and Events COM UA proxy

D.3.1 General

As illustrated in the figure below, the A&E COM UA Proxy is a COM *Server* combined with a UA *Client*. It maps the *Alarms* and *Conditions* address space of UA A&C *Server* into the appropriate COM Alarms and *Event Object*s.

Subclauses D.3.2 through D.3.9 identify the design guidelines and constraints used to develop the A&E COM UA Proxy provided by the OPC Foundation. In order to maintain a high degree of consistency and interoperability, it is strongly recommended that vendors, who choose to implement their own version of the A&E COM UA Proxy, follow these same guidelines and constraints.

The A&E COM *Client* simply needs to address how to connect to the UA A&C *Server*. Connectivity approaches include the one where A&E COM *Client*s connect to a UA A&C *Server* with a CLSID just as if the target *Server* were an A&E COM *Server*. However, the CLSID may be considered virtual since it is defined to connect to intermediary components that ultimately connect to the UA A&C *Server*. Using this approach, the A&E COM *Client* calls co-create instance with a virtual CLSID as described above. This connects to the A&E COM UA Proxy components. The A&E COM UA Proxy then establishes a secure channel and session with the UA A&C *Server*. As a result, the A&E COM *Client* gets a COM *Event Server* interface pointer.

D.3.2 Server status mapping

The A&E COM UA Proxy reads the UA A&C *Server* status from the *Server Object Variable Node*. Status enumeration values that are returned in *ServerStatusDataType* structure may be mapped 1 for 1 to the A&E COM *Server* status values with the exception of UA A&C *Server* status values *Unknown* and *Communication Fault.* These both map to the A&E COM *Server* status value of *Failed*.

The VendorInfo string of the A&E COM *Server* status is mapped from *ManufacturerName*.

D.3.3 Event Type mapping

Since all *Alarms* and *Conditions Event*s belong to a subtype of *BaseEventType*, the A&E COM UA Proxy maps the subtype as received from the UA A&C *Server* to one of the three A&E *Event* types: Simple, Tracking and *Condition*. Figure D.2 shows the mapping as follows:

- those A&C *Event*s which are of subtype *AuditEventType* are marked as A&E *Event* type Tracking;
- those A&C *Event*s which are *ConditionType* are marked as A&E *Event* type *Condition*;
- those A&C *Event*s which are of any subtype except *AuditEventType* or *ConditionType* are marked as A&E *Event* type Simple.

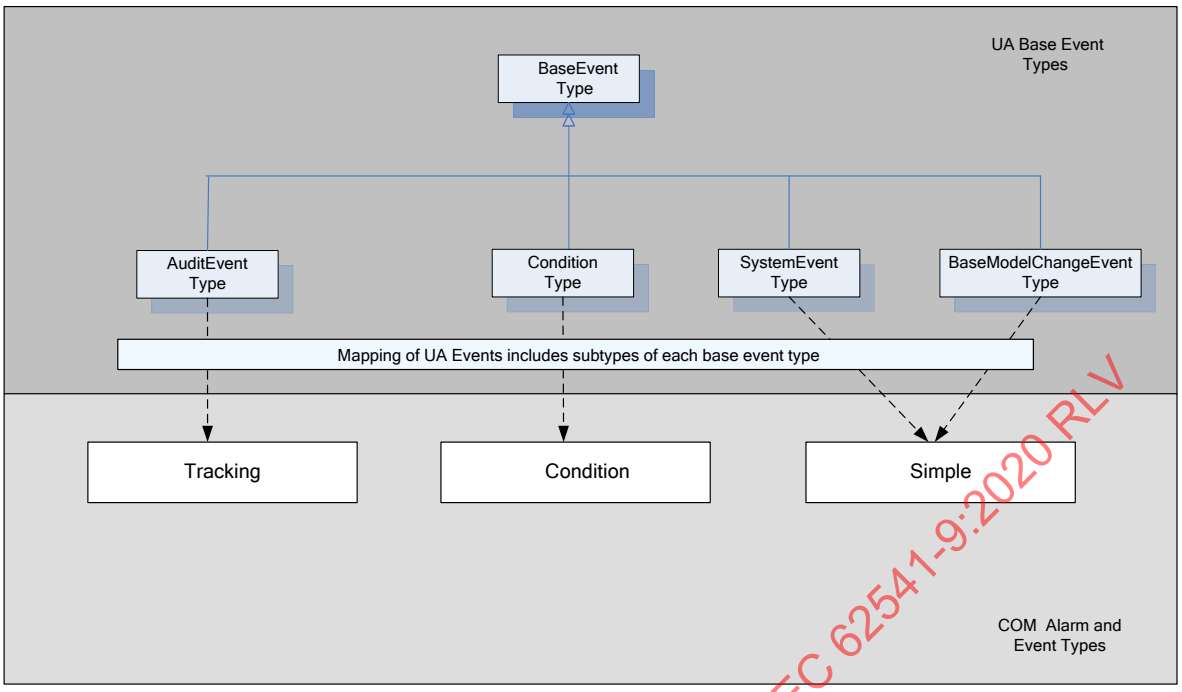


Figure D.2 – Mapping UA Event Types to COM A&E Event Types

Note that the *Event* type mapping described above also applies to the children of each subtype.

D.3.4 Event category mapping

Each A&E *Event* type (e.g. Simple, Tracking, *Condition*) has an associated set of *Event* categories which are intended to define groupings of A&E *Event*s. For example, Level and Deviation are possible *Event* categories of the *Condition Event* type for an A&E COM *Server*. However, since A&C does not explicitly support *Event* categories, the A&E COM UA Proxy uses A&C *Event* types to return A&E *Event* categories to the A&E COM *Client*. The A&E COM UA Proxy builds the collection of supported categories by traversing the type definitions in the address space of the UA A&C *Server*. Figure D.3 shows the mapping as follows:

- A&E Tracking categories consist of the set of all *Event* types defined in the hierarchy of subtypes of AuditEventType and TransitionEventType, including AuditEventType itself and TransitionEventType itself.
- A&E *Condition* categories consist of the set of all *Event* types defined in the hierarchy of subtypes of *ConditionType*, including *ConditionType* itself.
- A&E Simple categories consist of the set of *Event* types defined in the hierarchy of subtypes of *BaseEventType* excluding *AuditEventType* and *ConditionType* and their respective subtypes.

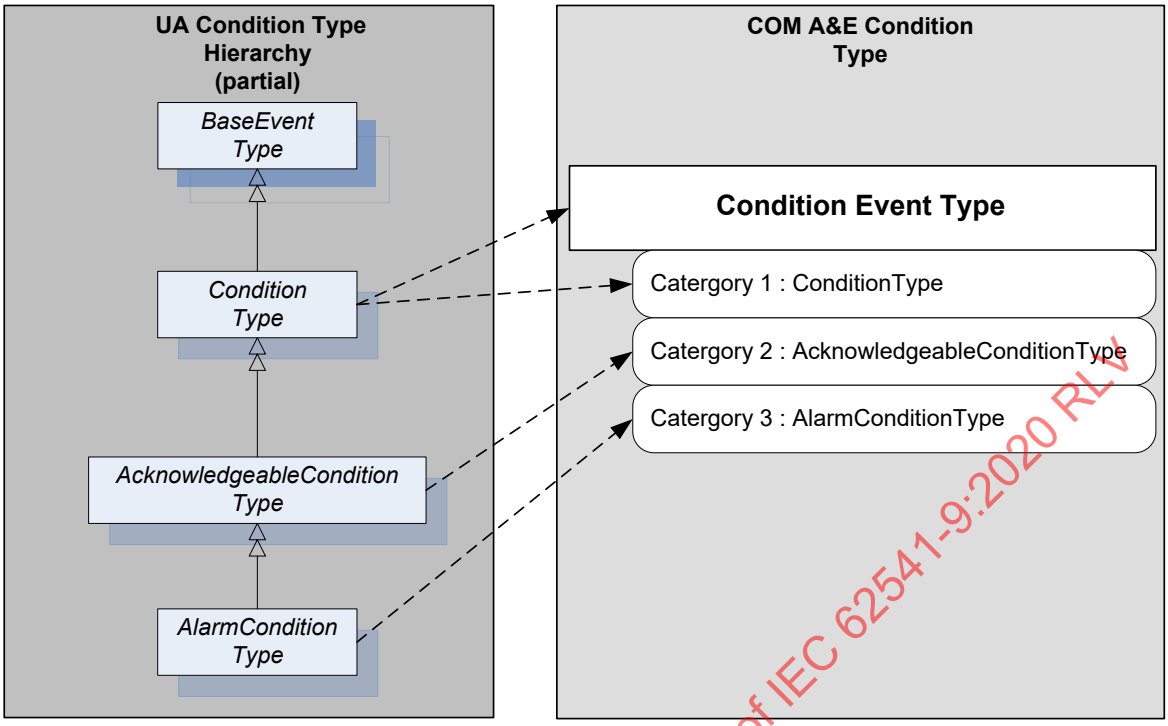


Figure D.3 – Example mapping of UA Event Types to COM A&E categories

Category name is derived from the display name *Attribute* of the *Node* type as discovered in the type hierarchy of the UA A&C *Server*.

Category description is derived from the description *Attribute* of the *Node* type as discovered in the type hierarchy of the UA A&C *Server*.

The A&E COM UA Proxy assigns Category IDs.

D.3.5 Event Category attribute mapping

The collection of *Attribute*s associated with any given A&E *Event* is encapsulated within the ONEVENTSTRUCT. Therefore, the A&E COM UA Proxy populates the *Attribute* fields within the ONEVENTSTRUCT using corresponding values from UA *Event Notification*s either directly (e.g. Source, Time, Severity) or indirectly (e.g. OPC COM *Event* category determined by way of the UA *Event* type). Table D.5 lists the *Attribute*s currently defined in the ONEVENTSTRUCT in the leftmost column. The rightmost column of Table D.5 indicates how the A&E COM UA proxy defines that *Attribute*.

Table D.5 – Event category attribute mapping table

A&E ONEVENTSTRUCT "attribute"	A&E COM UA Proxy Mapping
The following items are present for all A&E event types	
szSource	UA *BaseEventType Property*: *SourceName*
ftTime	UA *BaseEventType Property*: *Time*
szMessage	UA *BaseEventType Property*: *Message*
dwEventType	See D.3.3
dwEventCategory	See D.3.4
dwSeverity	UA *BaseEventType Property*: *Severity*
dwNumEventAttrs	Calculated within A&E COM UA Proxy
pEventAttributes	Constructed within A&E COM UA Proxy
The following items are present only for A&E Condition-Related Events	
szConditionName	UA *ConditionType Property*: *ConditionName*
szSubConditionName	UA *ActiveState Property*: *EffectiveDisplayName*
wChangeMask	Calculated within Alarms and Events COM UA proxy
wNewState: OPC_CONDITION_ACTIVE	A&C *AlarmConditionType Property*: ActiveState Note that events mapped as non-*Condition Event*s and those that do not derive from *AlarmConditionType* are set to ACTIVE by default.
wNewState: OPC_CONDITION_ENABLED	A&C *ConditionType Property*: EnabledState Note, *Event*s mapped as non-*Condition Event*s are set to ENABLED (state bit mask = 0x1) by default.
wNewState: OPC_CONDITION_ACKED	A&C *AcknowledgeableConditionType Property*: AckedState Note that A&C *Event*s mapped as non-*Condition Event*s or which do not derive from *AcknowledgeableConditionType* are set to UNACKNOWLEDGED and AckRequired = False by default.
wQuality	A&C ConditionType *Property*: Quality Note that *Event*s mapped as non-*Condition Event*s are set to OPC_QUALITY_GOOD by default. In general, the Severity field of the StatusCode is used to map COM status codes OPC_QUALITY_BAD, OPC_QUALITY_GOOD and OPC_QUALITY_UNCERTAIN. When possible, specific status' are mapped directly. These include (UA => COM): Bad status codes Bad_ConfigurationError => OPC_QUALITY_CONFIG_ERROR Bad_NotConnected => OPC_QUALITY_NOT_CONNECTED Bad_DeviceFailure => OPC_QUALITY_DEVICE_FAILURE Bad_SensorFailure => OPC_QUALITY_SENSOR_FAILURE Bad_NoCommunication => OPC_QUALITY_COMM_FAILURE Bad_OutOfService => OPC_QUALITY_OUT_OF_SERVICE Uncertain status codes Uncertain_NoCommunicationLastUsableValue => OPC_QUALITY_LAST_USABLE Uncertain_LastUsableValue => OPC_QUALITY_LAST_USABLE Uncertain_SensorNotAccurate => OPC_QUALITY_SENSOR_CAL Uncertain_EngineeringUnitsExceeded => OPC_QUALITY_EGU_EXCEEDED Uncertain_SubNormal => OPC_QUALITY_SUB_NORMAL Good status codes Good_LocalOverride => OPC_QUALITY_LOCAL_OVERRIDE
bAckRequired	If the ACKNOWLEDGED bit (OPC_CONDITION_ACKED) is set then the Ack Required Boolean is set to False, otherwise the Ack Required Boolean is set to True. If the *Event* is not of type *AcknowledgeableConditionType* or subtype, then the AckRequired Boolean is set to False.

A&E ONEVENTSTRUCT "attribute"	A&E COM UA Proxy Mapping
ftActiveTime	If the *Event* is of type *AlarmConditionType* or subtype and a transition from *ActiveState* of False to *ActiveState* to True is being processed then the *TransitionTime Property* of *ActiveState* is used. If the *Event* is not of type *AlarmConditionType* or subtype then this field is set to current time. Note: Additional logic applies to exclusive limit alarms, This value should be mapped to the LimitState.TransitionTime.
dwCookie	Generated by the A&E COM UA Proxy. These unique *Condition Event* cookies are not associated with any related identifier from the address space of the UA A&C *Server*.
The following is used only for A&E tracking events and for A&E condition-relate events which are acknowledgement notifications	
szActorID	
Vendor specific *Attributes* – ALL	
ACK Comment	
AREAS	All A&E *Event*s are assumed to support the "Areas" *Attribute*. However, no *Attribute* or *Property* of an A&C *Event* is available which provides this value. Therefore, the A&E COM UA Proxy initializes the value of the Areas *Attribute* based on the *MonitoredItem* producing the *Event*. If the A&E COM *Client* has applied no area filtering to a *Subscription*, the corresponding A&C *Subscription* will contain just one *MonitoredItem* – that of the UA A&C *Server Object*. *Event*s forwarded to the A&E COM *Client* on behalf of this *Subscription* will carry an Areas *Attribute* value of empty string. If the A&E COM *Client* has applied an area filter to a *Subscription* then the related UA A&C *Subscription* will contain one or more *MonitoredItem*s for each notifier *Node* identified by the area string(s). *Event*s forwarded to the A&E COM *Client* on behalf of such a *Subscription* will carry an areas *Attribute* whose value is the relative path to the notifier which produced the *Event* (i.e. the fully qualified area name).
Vendor specific *Attribute*s – based on category	
SubtypeProperty1	All the UA A&C subtype *Properties* that are not part of the standard set exposed by *BaseEventType* or *ConditionType*
SubtypeProperty*n*	

Condition Event instance records are stored locally within the A&E COM UA Proxy. Each record holds ONEVENTSTRUCT data for each EventSource/*Condition* instance. When the *Condition* instance transitions to the state INACTIVE|ACKED, where AckRequired = True or simply INACTIVE, where AckRequired = False, the local *Condition* record is deleted. When a *Condition Event* is received from the UA A&C *Server* and a record for this *Event* (identified by source/*Condition* pair) already exists in the proxy *Condition Event* store, the existing record is simply updated to reflect the new state or other change to the *Condition*, setting the change mask accordingly and producing an OnEvent callback to any subscribing *Client*s. In the case where the *Client* application acknowledges an *Event* which is currently unacknowledged (AckRequired = True), the UA A&C *Server* Acknowledge *Method* associated with the *Condition* is called and the subsequent *Event* produced by the UA A&C *Server* indicating the transition to acknowledged will result in an update to the current state of the local *Condition* record, as well as an OnEvent *Notification* to any subscribing *Client*s.

The A&E COM UA Proxy maintains the mapping of *Attribute*s on an *Event* category basis. An *Event* category inherits its *Attribute*s from the *Properties* defined on all supertypes in the UA *Event* Type hierarchy. New *Attribute*s are added for any *Properties* defined on the direct UA *Event* type to A&E category mapping. The A&E COM UA Proxy adds two *Attribute*s to each category: AckComment and Areas. Figure D.4 shows an example of this mapping.

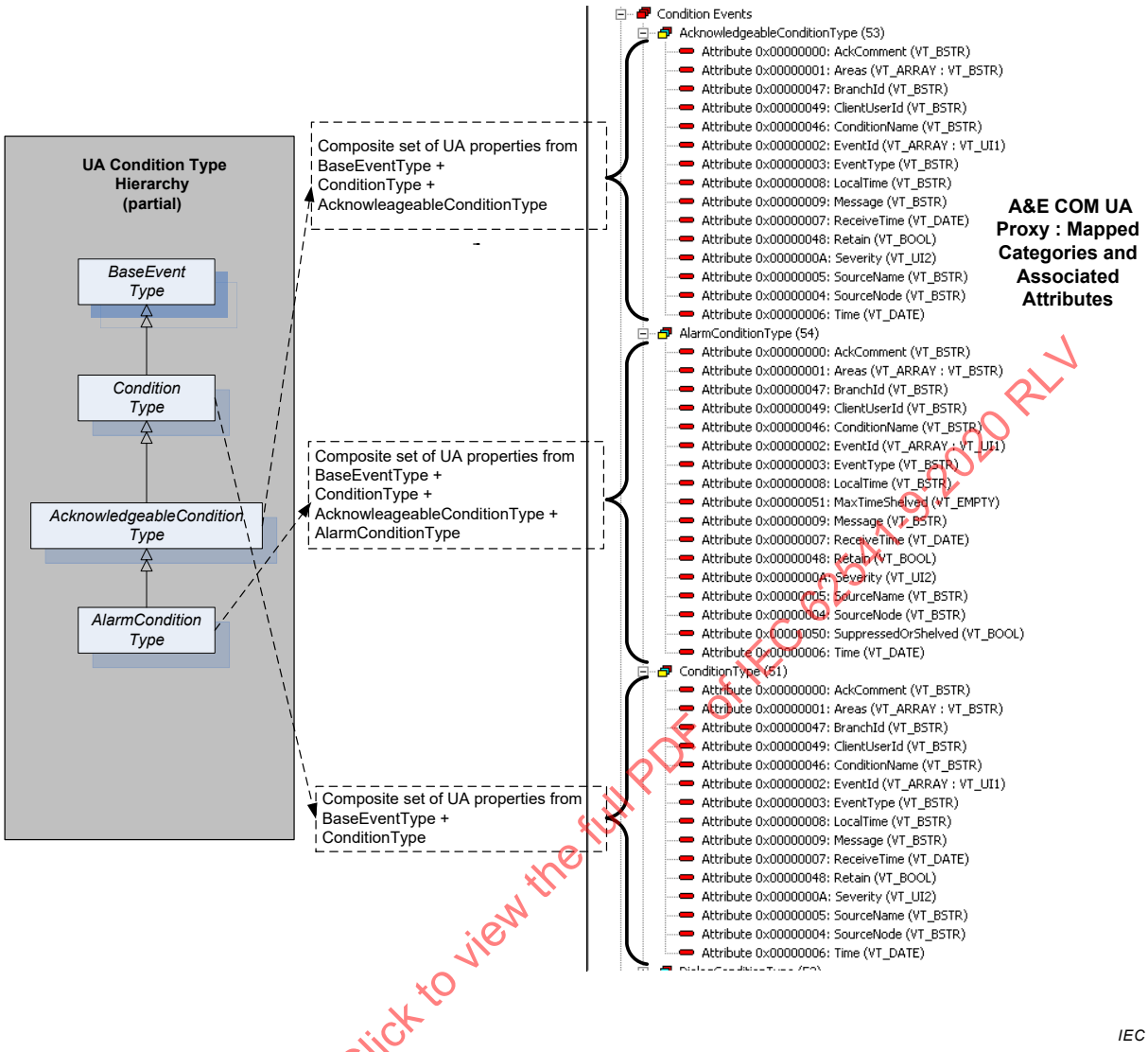


Figure D.4 – Example mapping of UA Event Types to A&E categories with attributes

D.3.6 Event Condition mapping

*Event*s of any subtype of *ConditionType* are designated COM *Condition Event*s and are subject to additional processing due to the stateful nature of *Condition Event*s. COM *Condition Event*s transition between states composed of the triplet ENABLED|ACTIVE|ACKNOWLEDGED. In UA A&C, *Event* subtypes of *ConditionType* only carry a value which can be mapped to ENABLED (DISABLED) and optionally, depending on further sub typing, may carry additional information which can be mapped to ACTIVE (INACTIVE) or ACKNOWLEDGED (UNACKNOWLEGED). *Condition Event* processing proceeds as described in Table D.5 (see A&E ONEVENTSTRUCT "*Attribute*" rows: OPC_CONDITION_ACTIVE, OPC_CONDITION_ENABLED and OPC_CONDITION_ACKED).

D.3.7 Browse mapping

A&E COM browsing yields a hierarchy of areas and sources. Areas can contain both sources and other areas in tree fashion where areas are the branches and sources are the leaves. The A&E COM UA Proxy relies on the "HasNotifier" *Reference* to assemble a hierarchy of branches/areas such that each *Object Node* which contains a *HasNotifier Reference* and whose *EventNotifier Attribute* is set to *SubscribeToEvents* is considered an area. The root for the *HasNotifier* hierarchy is the *Server Object*. Starting at the *Server Object*, *HasNotifier References* are followed and each *HasNotifier* target whose EventNotifier *Attribute* is set to SubscribeToEvents becomes a nested COM area within the hierarchy.

The HasNotifier target may also be a HasNotifier source. Further, any *Node* which is a HasEventSource source and whose EventNotifier *Attribute* is set to SubscribeToEvents is also considered a COM Area. The target *Node* of any HasEventSource *Reference* is considered an A&E COM "source" or leaf in the A&E COM browse tree.

In general, *Node*s which are the source *Node*s of the HasEventSource *Reference* and/or are the source *Node*s of the HasNotifier *Reference* are always A&ECOM Areas. *Node*s which are the target *Node*s of the HasEventSource *Reference* are always A&E COM Sources. Note however that targets of HasEventSource which cannot be found by following the HasNotifier *Reference*s from the *Server Object* are ignored.

Given the above logic, the A&E COM UA Proxy browsing will have the following limitations: Only those *Node*s in the UA A&C *Server*'s address space which are connected by the HasNotifier *Reference* (with exception of those contained within the top level *Object*s folder) are considered for area designation. Only those *Node*s in the UA A&C *Server*'s address space which are connected by the HasEventSource *Reference* (with exception of those contained within the top level *Object*s folder) are considered for area or source designation. To be an area, a *Node* shall contain a HasNotifier *Reference* and its EventNotifier *Attribute* shall be set to SubscribeToEvents. To be a source, a *Node* shall be the target *Node* of a HasEventSource *Reference* and shall have been found by following HasNotifier *Reference*s from the *Server Object*.

D.3.8 Qualified names

D.3.8.1 Qualified name syntax

From the root of any sub tree in the address space of the UA A&C *Server*, the A&E COM *Client* may request the list of areas and/or sources contained within that level. The resultant list of area names or source names will consist of the set of browse names belonging to those *Node*s which meet the criteria for area or source designation as described above. These names are "short" names meaning that they are not fully qualified. The A&E COM *Client* may request the fully qualified representation of any of the short area or source names. In the case of sources, the fully qualified source name returned to the A&E COM *Client* will be the string encoded value of the *NodeId* as defined in IEC 62541-6 (e.g., "ns=10;i=859"). In the case of areas, the fully qualified area name returned to the COM *Client* will be the relative path to the notifier *Node* as defined in IEC 62541-4 (e.g., "/6:Boiler1/6:Pipe100X/1:Input/2:Measurement"). Relative path indices refer to the namespace table described in D.3.8.2.

D.3.8.2 Namespace table

UA *Server* Namespace table indices may vary over time. This represents a problem for those A&E COM *Client*s which cache and reuse fully qualified area names. One solution to this problem would be to use a qualified name syntax which includes the complete URIs for all referenced table indices. This, however, would result in fully qualified area names which are unwieldy and impractical for use by A&E COM *Client*s. As an alternative, the A&E COM UA Proxy will maintain an internal copy of the UA A&C *Server*'s namespace table together with the locally cached endpoint description. The A&E COM UA Proxy will evaluate the UA A&C *Server*'s namespace table at connect time against the cached copy and automatically handle any re-mapping of indices if required. The A&E COM *Client* can continue to present cached fully qualified area names for filter purposes and the A&E COM UA Proxy will ensure these names continue to reference the same notifier *Node* even if the *Server*'s namespace table changes over time.

To implement the relative path, the A&E COM UA Proxy maintains a stack of *INode* interfaces of all the *Node*s browsed leading to the current level. When the A&E COM *Client* calls GetQualifiedAreaName, the A&E COM UA Proxy first validates that the area name provided is a valid area at the current level. Then looping through the stack, the A&E COM UA Proxy builds the relative path. Using the browse name of each *Node*, the A&E COM UA Proxy constructs the translated name as follows:

QualifiedName translatedName = new QualifiedName(Name,(ushort) ServerMappingTable[NamespaceIndex]) where

Name – the unqualified browse name of the *Node*

NamespaceIndex – the *Server* index

the *ServerMappingTable* provides the *Client* namespace index that corresponds to the *Server* index.

A '/' is appended to the translated name and the A&E COM UA Proxy continues to loop through the stack until the relative path is fully constructed.

D.3.9 Subscription filters

D.3.9.1 General

The A&E COM UA Proxy supports all of the defined A&E COM filter criteria.

D.3.9.2 Filter by Event, category or severity

These filter types are implemented using simple numeric comparisons. For *Event* filters, the received *Event* shall match the *Event* type(s) specified by the filter. For Category filters, the received *Event*'s category (as mapped from UA *Event* type) shall match the category or categories specified by the filter. For severity filters, the received *Event* severity shall be within the range specified by the *Subscription* filter.

D.3.9.3 Filter by source

In the case of source filters, the UA A&C *Server* is free to provide any appropriate, *Server*-specific value for SourceName. There is no expectation that source *Node*s discovered via browsing can be matched to the SourceName *Property* of the *Event* returned by the UA A&C Server using string comparisons. Further, the A&E COM *Client* may receive *Event*s from sources which are not discoverable by following only HasNotifier and/or HasEventSource *Reference*s. Thus, source filters will only apply if the source string can be matched to the SourceName *Property* of an *Event* as received from the target UA A & C *Server*. Source filter logic will use the pattern matching rules documented in the A&E COM specification, including the use of wildcard characters.

D.3.9.4 Filter by area

The A&E COM UA Proxy implements Area filtering by adjusting the set of *MonitoredItem*s associated with a *Subscription*. In the simple case where the *Client* selects no area filter, the A&E COM UA Proxy will create a UA *Subscription* which contains just one *MonitoredItem*, the *Server Object*. In doing so, the A&E COM UA Proxy will receive *Event*s from the entire *Server* address space – that is, all Areas. The A&E COM *Client* will discover the areas associated with the UA *Server* address space by browsing. The A&E COM *Client* will use GetQualifiedAreaName as usual in order to obtain area strings which may be used as filters. When the A&E COM *Client* applies one or more of these area strings to the COM *Subscription* filter, the A&E COM UA Proxy will create *MonitoredItem*s for each notifier *Node* identified by the area string(s). Recall that the fully qualified area name is in fact the namespace qualified relative path to the associated notifier *Node*.

The A&E COM UA Proxy calls the *TranslateBrowsePathsToNodeIds Service* to get the *Node* ids of the fully qualified area names in the filter. The *Node* ids are then added as *MonitoredItem*s to the UA *Subscription* maintained by the A&E COM UA Proxy. The A&E COM UA Proxy also maintains a reference count for each of the areas added, to handle the case of multiple A&E COM *Subscription* applying the same area filter. When the A&E COM *Subscription*s are removed or when the area name is removed from the filter, the ref count on the *MonitoredItem* corresponding to the area name is decremented. When the ref count goes to zero, the *MonitoredItem* is removed from the UA *Subscription.*

As with source filter strings, area filter strings may contain wildcard characters. Area filter strings which contain wildcard characters require more processing by the A&E COM UA Proxy. When the A&E COM *Client* specifies an area filter string containing wildcard characters, the A&E COM UA Proxy will scan the relative path for path elements that are completely specified. The partial path containing just those segments which are fully specified represents the root of the notifier sub tree of interest. From this sub tree root *Node*, the A&E COM UA Proxy will collect the list of notifier *Node*s below this point. The relative path associated with each of the collected notifier *Node*s in the sub tree will be matched against the *Client* supplied relative path containing the wildcard character. A *MonitoredItem* is created for each notifier *Node* in the sub-tree whose relative path matches that of the supplied relative path using established pattern matching rules. An area filter string which contains wildcard characters may result in multiple *MonitoredItem*s added to the UA *Subscription*. By contrast, an area filter string made up of fully specified path segments and no wildcard characters will result in one *MonitoredItem* added to the UA *Subscription*. So, the steps involved are:

1) check if the filter string contains any of these wild card characters, '*', '?', '#', '[', ']', '!', '-';
2) scan the string for path elements that are completely specified by retrieving the substring up to the last occurrence of the ‘/’ character;
3) obtain the *Node*Id for this path using *TranslateBrowsePathsToNodeIds*;
4) browse the *Node* for all notifiers below it;
5) using the ComUtils.Match() function match the browse names of these notifiers against the *Client* supplied string containing the wild card character;
6) add the *Node* ids of the notifiers that match as *MonitoredItem*s to the UA *Subscription*.

Annex E
(informative)

IEC 62682 Mapping

E.1 Overview

This annex provides a description of how the IEC 62682 information model may be mapped to OPC UA. It highlights term differences, concepts and other functionality. IEC 62682 provides additional information about managing and limiting alarms not covered by this specification.

NOTE ISA 18.2 is not discussed by this mapping, but IEC 62682 and ISA 18.2 are related and most definitions in ISA 18.2 correspond to the definitions in IEC 62682.

E.2 Terms

IEC 62682 defines a large number of terms that are covered by the OPC UA model but not used in the text. These IEC 62682 terms are listed in Table E.1 and include a description, mapping or relationship to OPC UA Alarms and Events:

Table E.1 – IEC 62682 Mapping

IEC 62682	OPC UA Mapping / Related Concept	IEC 62682 Definition OPC UA Application of
absolute alarm	ExclusiveDeviationAlarmType NonExclusiveDeviationAlarmType	An alarm generated when the alarm set point is exceeded.
		Both OPC UA models expose a set point and process the *Alarm* as an absolute *Alarm* requires, the only difference is the interaction between relative states (High, HighHigh...)
adaptive alarm		Alarm for which the setpoint is changed by an algorithm (e.g. rate based).
		In OPC UA, adaptive alarming may be part of a vendor specific alarm application, but it would or could make use of a number of standard *Alarm* functions described in this specification. OPC UA provides limit, rate of change and deviation alarming. Vendors may easily develop algorithms to adjust any of the limits that are exposed.
adjustable alarm / operator-set alarm	ExclusiveLimitAlarmType NonExclusiveLimitAlarmType	An alarm for which the set point can be changed manually by the *Operator*.
		Both OPC UA models allow *Alarm* limits to be writeable and allow for an *Operator* to change the limit. For all changes to limits, an audit event should be generated tracking the change.
advanced alarming		A collection of techniques that can help manage annunciations during specific situations.
		In OPC UA advanced alarming may be part of a vendor specific alarm application, but it would or could make use of a number of standard *Alarm* functions described in this specification, such as adaptive setting of a setpoint for deviation *Alarm*. It might also require the definition of new *Alarm* subtypes.

IEC 62682	OPC UA Mapping / Related Concept	IEC 62682 Definition
		OPC UA Application of
Annunciation / Alarm Annunciation	Retain	A function of the alarm system is to call the attention of the *Operator* to an alarm.
		OPC UA provides an *Alarm* model that includes concepts such as re-alarming, *Alarm* silence and *Alarm* delays, but it is up to the *Client* application to make use of these features to generate both audible and visual annunciation to the *Operator*. OPC UA does not provide visual indication but it does provide priority information on which the client can be configured to provide the appropriate visual display. A key concept for alarm display is the concept of *Alarm* states and a Retain bit (see Annex B for more details).
alarm attribute	Various Alarm Properties	The setting for an alarm within the process control system.
		OPC UA defines a number of Properties that reflect what would be termed alarm attributes in IEC 62682 such as *Alarm* setpoint which maps to the setpoint property in an *ExclusiveDeviationAlarmType*.
alarm class	ConditionClass, ConditionSubClass	A group of alarms with a common set of alarm management requirements (e.g. testing, training, monitoring, and audit requirements).
		OPC UA provides *ConditionClasses*, but also provides other groupings, like ConditionSubClass OPC UA also specifies a number of predefined classes, but it is expected that vendors, other standards group or even end users will define their own extensions to these classes. The OPC concepts allow *Alarm*s to be categorized as needed.
alarm Deadband	ExclusiveDeviationAlarmType NonExclusiveDeviationAlarmType	A change in signal from the alarm setpoint necessary for the alarm to return to normal.
		In OPC UA, the *ExclusiveDeviationAlarmType* and *NonExclusiveDeviationAlarmType* contain an *Alarm* deadband and can be used for the same functionality described in IEC 62682.
filtering(alarm)	Event Subscription	A function which selects alarm records to be displayed according to a given element of the alarm record.
		In OPC UA, *Alarms* are received by a *Client* according to the specific filter requested by the *Client*. The filtering can be very robust or very simple according to the needs of the client. It is up to the *Client* application to generate and provide the appropriate filter to the server. OPC UA's *Alarm* model is a subscription-based model, not a push model that is configured on a server. The choice of filter is a client's responsibility.
alarm flood	Alarm diagnostics	A condition during which the *Alarm* rate is greater than the *Operator* can effectively manage – (e.g. more than 10 *Alarm*s per 10 min).
		OPC UA does not define *Alarm* flooding but it does provide the capability to collect diagnostics that would allow an engineer to review overall *Alarm* performance.
alarm group	alarm group	A set of alarms with common association (e.g. process unit, process area, equipment set, or service). Alarm groups are primarily used for display purposes.
		OPC UA allows the definition of *Alarm* groups and the assignment of *Alarms* to these groups. In addition. OPC UA allows *Alarm*s to also be part of a category. OPC UA also allows *Alarm*s to be organized as a *HasNotifier* hierarchy (see Clause 6). Groups, categories and hierarchies can be used for filtering or restricting *Alarm*s that are being displayed.
alarm history	historical events	long-term repository for alarm records.
		IEC 62541-11 describes historical *Events*.

IEC 62682	OPC UA Mapping / Related Concept	IEC 62682 Definition OPC UA Application of
alarm log		short-term repository for alarm records.
		This part does not specify repositories for *Alarms*. *Alarm* logging is a *Client* function.
alarm management alarm system management		collection of processes and practices for determining, documenting, designing, operating, monitoring, and maintaining alarm systems.
		OPC UA provides an infrastructure to allow vendors and *Operator*s to provide *Alarm* management, as such it should be an integral part of an alarm management system.
alarm message	Events	text string displayed with the alarm indication that provides additional information to the *Operator* (e.g., *Operator* action).
		OPC UA provides an *Event* structure that includes many different pieces of information (see IEC 62541-5 for additional details). *Client*s can subscribe for as much of this information as desired and display this as an *Alarm* message. All typical fields that would be associated with an *Alarm* message are available. In addition, OPC UA provides significant additional information.
alarm priority	Priority	relative importance assigned to an alarm within the alarm system to indicate the urgency of response (e.g., seriousness of consequences and allowable response time)
		OPC UA provides a *Priority Variable* as part of the *Alarm Object* that provides the same functionality
alarm rate	Alarm diagnostics	the number of alarm annunciation, per *Operator*, in a specific time interval.
		OPC UA provides diagnostics allowing the collection of *Alarm* rate information at any level in the system.
Record (Alarm)	Events, Event filtering	a set of information which documents an alarm state change.
		In OPC UA all *Alarm*s are generated as an *Event* and the *Client* can select the fields that are to be included in the *Events*. This selection can be customized for each *AlarmConditionType*, which allows a customized *Alarm* record to be generated.
alarm setpoint, alarm limit, alarm trip point	Limit Alarms, Discrete Alarms	the threshold value of a process variable or discrete state that triggers the alarm indication.
		OPC UA supports *Alarm* limits and setpoints for multiple *Alarm* types, including limit *Alarms* and discrete *Alarms*.
Sorting (alarm)		a function which orders alarm records to be displayed according to a given element of alarm record.
		OPC UA does not provide *Alarm* sorting as part of an event subscription. Multiple filtering options are provided, but the *Client* is required to perform any ordering of *Alarms*.
alarm summary, alarm list		a display that lists alarm annunciations with selected information (e.g. date, time, priority, and alarm type).
		In OPC UA *Alarm* summaries and *Alarm* lists are *Client* functionality and are not specified. Extensive filtering capabilities are provided by the *Server* to allow easier implementation of *Alarm* summaries or lists by a *Client*.

IEC 62682	OPC UA Mapping / Related Concept	IEC 62682 Definition OPC UA Application of
Alert		An audible and/or visible means of indicating to the *Operator* an equipment or process condition that can require evaluation when time allows.
		Alerts are items that should be attended to, but are not as urgent as *Alarm*s. OPC UA does not differentiate between *Alarm*s and alerts, but it does provide a full range of priorities for *Alarm*s. It is up to the end users to determine what range of priorities are considered an alert vs an *Alarm* etc.
allowable response time		The maximum time between the annunciation of the alarm and when the *Operator* takes corrective action to avoid the consequence.
		OPC UA does not provide any specific fields for allowable response time, but it does track the times at which an *Alarm* occurs and when any actions are taken on the *Alarm*.
annunciator		device or group of devices that call attention to changes in process conditions
		OPC UA does not define annunciators, this is *Client* functionality that can be implemented using OPC UA
Audit		comprehensive assessment that includes the evaluation of alarm system performance and the effectiveness of the work practices used to administer the alarm system.
		OPC UA does provide a number of features that can facilitate an audit, including diagnostics and audit events. Do not confuse OPC *Audit Event* with the IEC audit concept.
bad-measurement alarm		an alarm generated when the signal for a process measurement is outside the expected range (e.g. 3.8 mA for a 4 mA to 20 mA signal).
		A bad measurement *Alarm* is not defined in OPC UA, but limit *Alarm*s are defined and they could be used directly to represent a bad-measurement *Alarm*. Alternatively, limit *Alarms* could be further subtyped to allow easier filtering on bad-measurement *Alarm*s if desired.
bit-pattern alarm	Discrete alarm	an alarm that is generated when a pattern of digital signals matches a predetermined pattern.
		In OPC UA a bit pattern *Alarm* can be mapped to a *DiscreteAlarmType*.
calculated alarm		An alarm generated from a calculated value instead of a direct process measurement.
		In OPC UA any of the defined *Alarm* types can be applied to calculated values or to process values.
call-out alarm		alarm that notifies and informs an *Operator* by means other than, or in addition to, a console display (e.g. pager or telephone)
		OPC UA does not specify call-out alarms, since this is client functionality. OPC UA does provide the ability to categorize or group an *Alarm* such that it could be easily identified as requiring a different type of annunciation.
chattering alarm	OnDelay, OffDelay	alarm that repeatedly transitions between the alarm state and the normal state in a short period of time.
		The OPC UA features of *OnDelay* and *OffDelay* can be used to help control chattering *Alarm*s.

IEC 62682	OPC UA Mapping / Related Concept	IEC 62682 Definition OPC UA Application of
classification	ConditionClasses	the process of separating alarms into alarm classes based on common requirements (e.g. testing, training, monitoring, and auditing requirements).
		OPC defines a number of extensible *ConditionClasses* that can be used for this purpose.
controller-output alarm		alarm generated from the output signal of a control algorithm (e.g. PID controller) instead of a direct process measurement.
		OPC UA does not provide an *Alarm* type for controller-output alarm, but a type could be created or an existing type could be used, depending on the requirements.
dynamic alarming		An automatic modification of alarm attributes based on process state or conditions.
		OPC UA does not define dynamic alarming behaviour, but it allows programmatic access to limits, set points or other parameters that would be required for a dynamic alarming solution.
enforcement		enhanced alarming technique that can verify and restore alarm attributes in the control system to the values in the master alarm database.
		OPC UA does not provide enforcement, but it enables enforcement by providing an information model that includes default setting for *Alarm* types as well as original settings for dynamic *Alarms*. These features may be used by a *Client* application to provide enforcement.
fleeting alarm	Suppression, Shelving	An alarm that transitions between an active alarm state and an inactive alarm state in a short period of time.
		OPC UA provides *Alarm Suppression* and *Shelving* which an *Operator* might use to control fleeting *Alarms*.
first-out alarm first-up alarm	FirstInGroup FirstInGroupFlag	An alarm determined (i.e. by first-out logic) to be the first, in a multiple-alarm scenario.
		OPC UA can support first-up/first-out *Alarms* as part of the *Alarm* information model, including definition of the group of *Alarms*.
instrument diagnostic alarm	InstrumentDiagnosticAlarmType	An alarm generated by a field device to indicate a fault (e.g. sensor failure).
		OPC UA provides support for InstrumentDiagnostic Alarms that can be used to represent a failed sensor or an instrument diagnostic.
monitoring	Alarm Diagnostics	measurement and reporting of quantitative (objective) aspects of alarm system performance.
		OPC UA provides diagnostic collection capabilities that can be used to measure and reports quantitative information related to alarm system performance.
nuisance alarm	Alarm Diagnostics	An alarm that annunciates excessively, unnecessarily, or does not return to normal after the *Operator* response is taken. EXAMPLE: Chattering alarm, fleeting alarm, or stale alarm.
		The OPC UA model provides Alarm Diagnostics for tracking the information needed to identify if an *Alarm* is a nuisance *Alarm* (i.e. has been in an *Alarm* state excessively or does not return to normal).
plant state plant mode	StateMachines	defined set of operational conditions for a process plant.
		OPC UA provides an example StateMachine (see Annex F) that can be customized or adapted to provide process information. This StateMachine could also be used to affect alarming.

IEC 62682	OPC UA Mapping / Related Concept	IEC 62682 Definition
		OPC UA Application of
process area	Event Hierarchies Object References (IEC 62541-5)	physical, geographical or logical grouping of resources determined by the site.
		OPC UA provides multiple manners in which an information model can be displayed, this includes grouping objects into process areas or any other desired grouping. This is an inherent part of the OPC UA information model.
re-alarming alarm, re-triggering alarm	ReAlarmTime ReAlarmRepeatCount	alarm that is automatically re-annunciated to the *Operator* under certain conditions.
		OPC UA supports re-alarming as part of its base AlarmConditionType.
recipe-driven alarm	StateMachines Alarm Limits	alarm with setpoints that depend on the recipe that is currently being executed.
		OPC UA provides support for adjustable *Alarm* limits. It also provides support for programs and other functionality that could be used to drive recipes. Annex F provides an example of a *StateMachine* and how it could be used to adjust *Alarm* settings.
Reset	LatchedState / Reset	*Operator* action that unlatches a latched alarm.
		OPC UA provides an optional *StateMachine* to indicate an Alarm is capable of being latched and is in a latched state. It also provides a *Reset Method* for clearing the latched state.
safety related alarm safety alarm	SafetyConditionClassType	an alarm that is classified as critical to process safety for the protection of human life or the environment.
		OPC UA defines a safety *ConditionClass* for grouping safety related alarms.
stale alarm	Alarm Diagnostics	alarm that remains annunciated for an extended period of time (e.g. 24 hours).
		OPC UA *Alarm Diagnostics* can track the length of time an *Alarm* is active.
state-based alarm – mode-based alarms	StateMachine	alarm that has attributes modified or is suppressed based on operating states or process conditions.
		OPC UA can provide a system state *StateMachine* to support process, device or system states (see Annex F). With this *StateMachine Servers* can adjust *Alarm* attributes or just *Suppress* or *Disable Alarms* based on the *StateMachine*. The *StateMachine* can be applied at multiple levels in the system.
statistical alarm	StatisticalConditionClassType	alarm generated based on statistical processing of a process variable or variables.
		OPC UA provides an *Alarm Condition* class that any of the existing *AlarmConditionTypes* can be assigned to. This allows any *Alarm* types, such as limit *Alarm*s, to be generated by statistical analysis.
Suppress	SuppressedOrShelved	Any mechanism to prevent the indication of the alarm to the *Operator* when the base alarm condition is present (i.e. shelving, suppressed by design, out-of-service).
		OPC UA provides a flag *SuppressedOrShelved* that matches this functionality.
suppressed by design	SuppressedState	alarm annunciation to the *Operator* prevented based on plant state or other conditions.
		OPC UA provides a SuppressedState that matches this functionality.

IEC 62682	OPC UA Mapping / Related Concept	IEC 62682 Definition
		OPC UA Application of
system diagnostic alarm	SystemDiagnosticAlarmType	alarm generated by the control system to indicate a fault within the system hardware, software or components.
		OPC UA defines a system diagnostic *Alarm* that can be used to represent faults with system hardware, software or components.,

The following terms in IEC 62682 match the terms/concepts defined in the OPC UA specification and do not need any additional mapping or discussion:

- Acknowledge
- Active
- Alarm
- Alarm OffDelay
- Alarm OnDelay
- Alarm Type
- Deviation Alarm
- Discrepancy Alarm
- Event
- Highly Managed Alarm
- LatchingAlarm
- OutofService
- Rateofchange alarms
- Return to normal
- Shelve
- Silence
- Unacknowledged

E.3 Alarm records and State indications

OPC UA provides all of the items listed as both required and recommended as part of its alarm definitions, but it is up to the client to subscribe for the information. In OPC UA the Client controls what alarm information is requested and obtained from the *Server*. The *Server* does not define visual aspects of the alarm system, but does provide priority information from which the visual aspect can be set on the client side.

OPC UA also supports all of the states described in IEC 62682. This includes tracking the process states, system states and individual alarm states. OPC UA also provides a StateMachine model that can be used in conjunction with an alarm system to alter alarm behaviour based on the state of a system or process. For example, during start-up or shutdown of a process or a system, some alarms might be suppressed.

The behaviour of an OPC UA alarm system also mimics that required by IEC 62682. All behaviour described in IEC 62682 can easily be mapped to functionality define in OPC UA Alarm & Conditions.

Annex F
(informative)

System State

F.1 Overview

The state of alarms is affected by the state of the process, equipment, system or plant. For example, when a tank is taken out of service, the level alarms associated with the tank would be no longer used, until the tank is returned to service. This annex describes a *StateMachine* that can be deployed as part of a system designed and used to reflect the current state of the system, process, equipment or item. A customized version of this model can be implemented for any system, this sample is just an illustration.

The current state from the *StateMachine* is applied to all items in the *HasNotifier* hierarchy below the object with which the *StateMachine* is associated. The *SystemState StateMachine* can be used to automatically disable, enable, suppress or un-suppress *Alarms* related to the Object (with in the hierarchy of alarms from the given object). The *StateMachine* can also be used by advanced alarming software to adjust the setpoint, limits or other items related to the *Alarms* in the hierarchy.

Optionally, multiple SystemState *StateMachines* can be deployed.

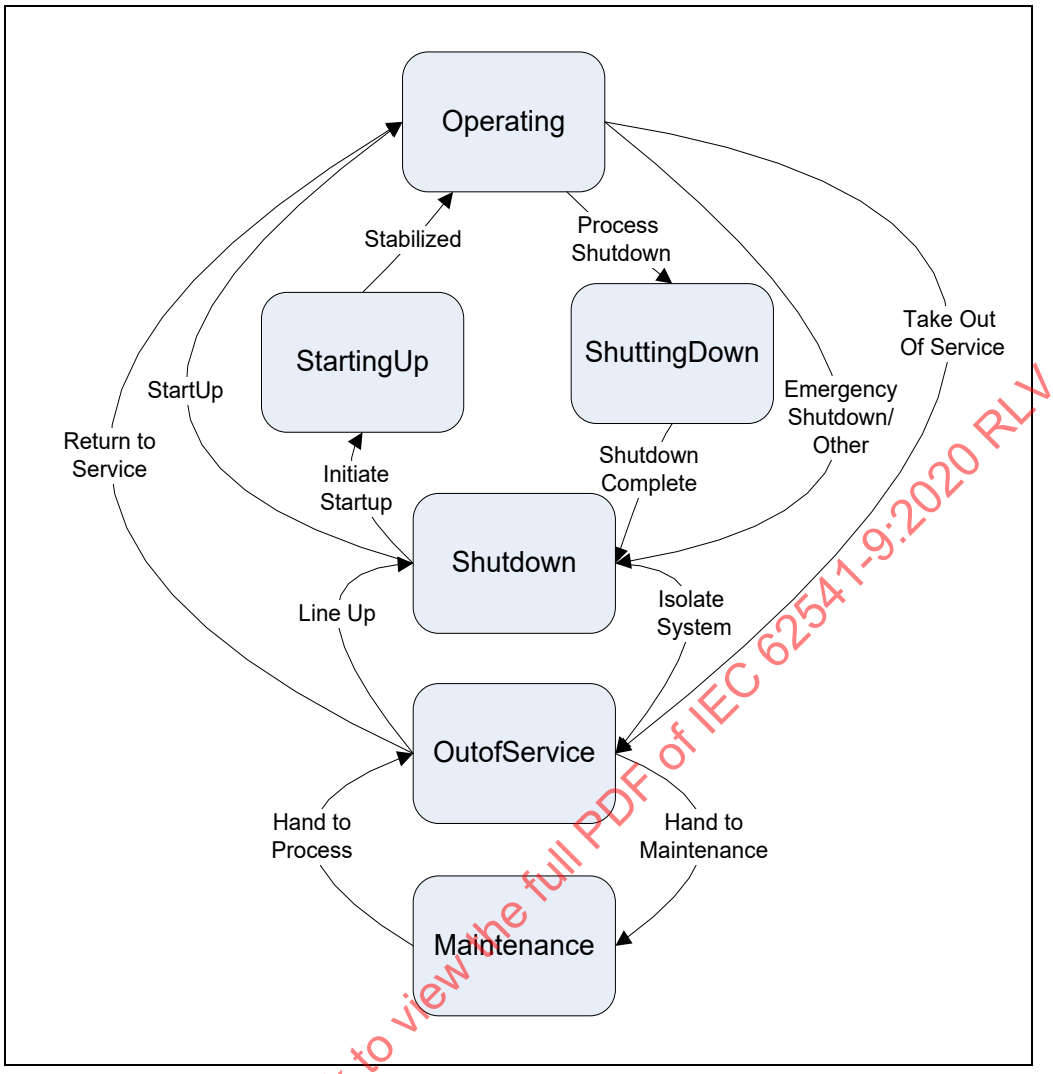


Figure F.1 – SystemState transitions

F.2 SystemStateStateMachineType

The *SystemStateStateMachineType* includes a hierarchy of substates. It supports multiple transitions between Operating, StartingUp, ShuttingDown, Shutdown, OutOfService and Maintenance.

The state machine is illustrated in Figure F.2 and formally defined in Table F.1.

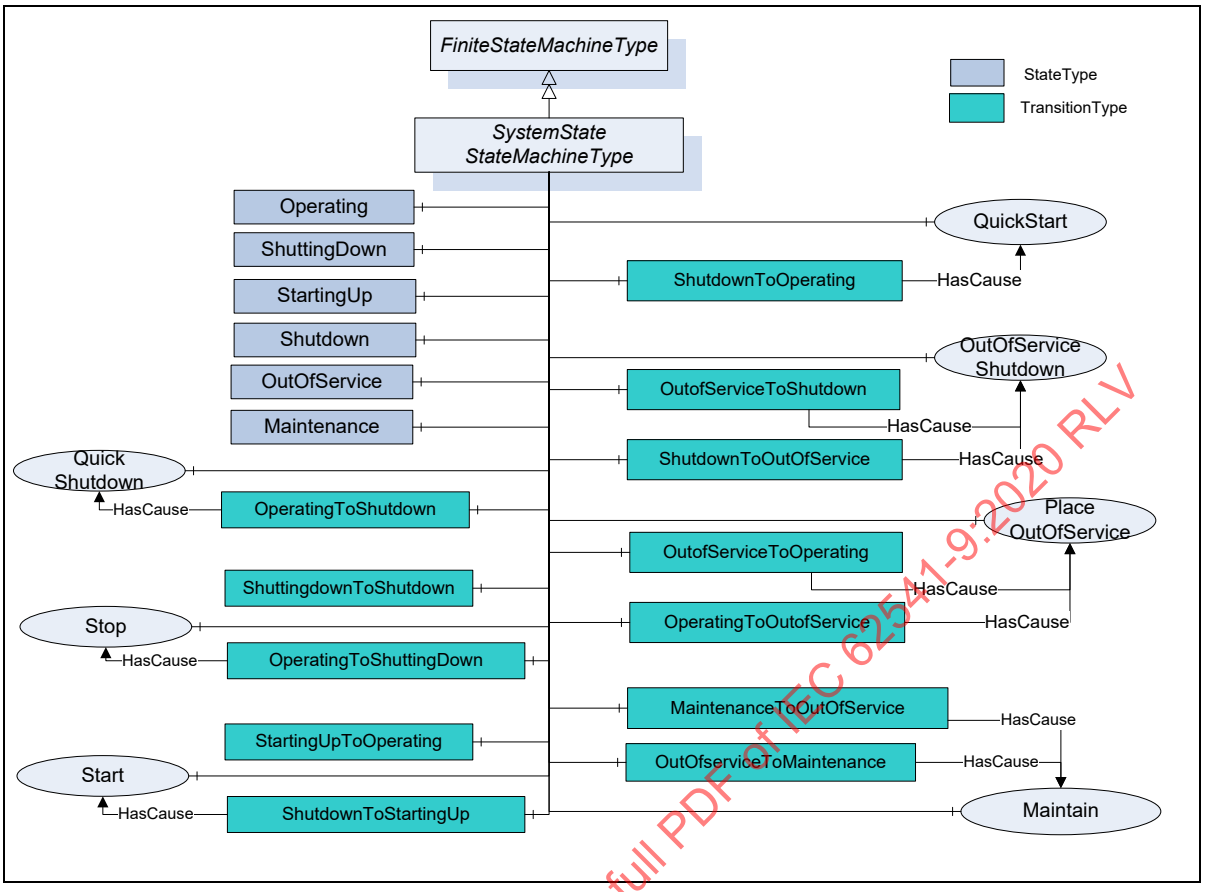


Figure F.2 – SystemStateStateMachineType Model

Table F.1 – SystemStateStateMachineType definition

Attribute	Value				
BrowseName	SystemStateStateMachineType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the Finite*StateMachineType* defined in IEC 62541-5					
HasComponent	Object	Operating		StateType	
HasComponent	Object	ShuttingDown		StateType	
HasComponent	Object	StartingUp		StateType	
HasComponent	Object	Shutdown		StateType	
HasComponent	Object	OutOfService		StateType	
HasComponent	Object	Maintenance		StateType	
HasComponent	Object	ShutdownToOperating		TransitionType	
HasComponent	Object	OperatingToShutdown		TransitionType	
HasComponent	Object	ShuttingdownToShutdown		TransitionType	
HasComponent	Object	OperatingToShuttingdown		TransitionType	
HasComponent	Object	StartingUpToOperating		TransitionType	
HasComponent	Object	ShutdownToStartingUp		TransitionType	
HasComponent	Object	OutofServiceToShutdown		TransitionType	
HasComponent	Object	ShutdownToOutOfService		TransitionType	
HasComponent	Object	OutofServiceToOperating		TransitionType	
HasComponent	Object	OperatingToOutofService		TransitionType	
HasComponent	Object	MaintenanceToOutOfService		TransitionType	
HasComponent	Object	OutOfServiceToMaintenance		TransitionType	
HasComponent	Method	Start	Defined in Clause XXX		Optional
HasComponent	Method	Maintain	Defined in Clause XXX		Optional
HasComponent	Method	Stop	Defined in Clause XXX		Optional
HasComponent	Method	PlaceOutOfservice	Defined in Clause XXX		Optional
HasComponent	Method	QuickShutdown	Defined in Clause XXX		Optional
HasComponent	Method	QuickStart	Defined in Clause XXX		Optional
HasComponent	Method	OutOfServiceShutdown	Defined in Clause XXX		Optional

The actual selection of *States* and *Transitions* would depend on the deployment of the *StateMachine*. If the *StateMachine* were being applied to a tank or other part of a process, it might have a different set of *States* than if it were applied to a meter or instrument. The meter may only have Operating, OutOfService and Maintenance, while the tank may have all of the described *States* and *Transitions*.

The *StateMachine* supports six possible states including: *Operating*, *ShuttingDown*, *StartingUp*, *Shutdown*, *OutOfService*, *Maintenance*. It supports 12 possible *Transitions* and 7 possible *Methods*.

The *SystemStateStateMachineType* transitions are formally defined in Table F.2.

Table F.2 – SystemStateStateMachineType transitions

BrowseName	References	BrowseName	TypeDefinition
Transitions			
ShutdownToOperating	FromState	Shutdown	StateType
	ToState	Operating	StateType
	HasCause	QuickStart	*Method*
OperatingToShutdown	FromState	Operating	StateType
	ToState	Shutdown	StateType
	HasCause	QuickShutdown	*Method*
ShuttingdownToShutdown	FromState	ShuttingDown	StateType
	ToState	Shutdown	StateType
OperatingToShuttingdown	FromState	Operating	StateType
	ToState	ShuttingDown	StateType
	HasCause	Stop	*Method*
StartingUpToOperating	FromState	StartingUp	StateType
	ToState	Operating	StateType
ShutdownToStartingUp	FromState	Shutdown	StateType
	ToState	StartingUp	StateType
	HasCause	Start	*Method*
OutofServiceToShutdown	FromState	OutOfService	StateType
	ToState	Shutdown	StateType
	HasCause	OutOfServiceShutdown	*Method*
ShutdownToOutOfService	FromState	Shutdown	StateType
	ToState	OutOfService	StateType
	HasCause	OutOfServiceShutdown	*Method*
OutOfServiceToOperating	FromState	OutOfService	StateType
	ToState	Operating	StateType
	HasCause	PlaceOutOfService	*Method*
OperatingToOutofService	FromState	Operating	StateType
	ToState	OutOfService	StateType
	HasCause	PlaceOutOfService	*Method*
MaintenanceToOutofService	FromState	Maintenance	StateType
	ToState	OutOfService	StateType
	HasCause	Maintain	*Method*
OutOfServiceToMaintenance	FromState	OutOfService	StateType
	ToState	Maintenance	StateType
	HasCause	Maintain	*Method*

The system can always generate additional *HasCause References*, such as internal code. No *HasEffect References* are defined, but an implementation might define *HasEffect References* (such as *HasEffectDisable*) for disabling or enabling *Alarms*, suppressing *Alarms* or adjusting setpoints or limits of *Alarms*. The targets of the reference might be an individual *Alarm* or portion of a plant or piece of equipment. See Clause 7 for a list of *HasEffect References* that could be used.

Bibliography

ISA 18.2, *Management of Alarm Systems for the Process Industries*
https://www.isa.org/store/ansi/isa-182-2016,-management-of-alarm-systems-for-the-process-industries/46962105

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 RFC 2047, *Multipurpose Internet Mail Extensions (MIME) Part Three*
https://www.ietf.org/rfc/rfc2047.txt

SOMMAIRE

COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

_____

ARCHITECTURE UNIFIÉE OPC –

Partie 9: Alarmes et Conditions

AVANT-PROPOS

1) La Commission Electrotechnique Internationale (IEC) est une organisation mondiale de normalisation composée de l'ensemble des comités électrotechniques nationaux (Comités nationaux de l'IEC). L'IEC a pour objet de favoriser la coopération internationale pour toutes les questions de normalisation dans les domaines de l'électricité et de l'électronique. A cet effet, l'IEC – entre autres activités – publie des Normes internationales, des Spécifications techniques, des Rapports techniques, des Spécifications accessibles au public (PAS) et des Guides (ci-après dénommés "Publication(s) de l'IEC"). Leur élaboration est confiée à des comités d'études, aux travaux desquels tout Comité national intéressé par le sujet traité peut participer. Les organisations internationales, gouvernementales et non gouvernementales, en liaison avec l'IEC, participent également aux travaux. L'IEC collabore étroitement avec l'Organisation Internationale de Normalisation (ISO), selon des conditions fixées par accord entre les deux organisations.
2) Les décisions ou accords officiels de l'IEC concernant les questions techniques représentent, dans la mesure du possible, un accord international sur les sujets étudiés, étant donné que les Comités nationaux de l'IEC intéressés sont représentés dans chaque comité d'études.
3) Les Publications de l'IEC se présentent sous la forme de recommandations internationales et sont agréées comme telles par les Comités nationaux de l'IEC. Tous les efforts raisonnables sont entrepris afin que l'IEC s'assure de l'exactitude du contenu technique de ses publications; l'IEC ne peut pas être tenue responsable de l'éventuelle mauvaise utilisation ou interprétation qui en est faite par un quelconque utilisateur final.
4) Dans le but d'encourager l'uniformité internationale, les Comités nationaux de l'IEC s'engagent, dans toute la mesure possible, à appliquer de façon transparente les Publications de l'IEC dans leurs publications nationales et régionales. Toutes divergences entre toutes Publications de l'IEC et toutes publications nationales ou régionales correspondantes doivent être indiquées en termes clairs dans ces dernières.
5) L'IEC elle-même ne fournit aucune attestation de conformité. Des organismes de certification indépendants fournissent des services d'évaluation de conformité et, dans certains secteurs, accèdent aux marques de conformité de l'IEC. L'IEC n'est responsable d'aucun des services effectués par les organismes de certification indépendants.
6) Tous les utilisateurs doivent s'assurer qu'ils sont en possession de la dernière édition de cette publication.
7) Aucune responsabilité ne doit être imputée à l'IEC, à ses administrateurs, employés, auxiliaires ou mandataires, y compris ses experts particuliers et les membres de ses comités d'études et des Comités nationaux de l'IEC, pour tout préjudice causé en cas de dommages corporels et matériels, ou de tout autre dommage de quelque nature que ce soit, directe ou indirecte, ou pour supporter les coûts (y compris les frais de justice) et les dépenses découlant de la publication ou de l'utilisation de cette Publication de l'IEC ou de toute autre Publication de l'IEC, ou au crédit qui lui est accordé.
8) L'attention est attirée sur les références normatives citées dans cette publication. L'utilisation de publications référencées est obligatoire pour une application correcte de la présente publication.
9) L'attention est attirée sur le fait que certains des éléments de la présente Publication de l'IEC peuvent faire l'objet de droits de brevet. L'IEC ne saurait être tenue pour responsable de ne pas avoir identifié de tels droits de brevets et de ne pas avoir signalé leur existence.

La Norme internationale IEC 62541-9 a été établie par le sous-comité 65E: Les dispositifs et leur intégration dans les systèmes de l'entreprise, du comité d'études 65 de l'IEC: Mesure, commande et automation dans les processus industriels.

Cette troisième édition annule et remplace la deuxième édition parue en 2015. Cette édition constitue une révision technique.

Cette édition inclut les modifications techniques majeures suivantes par rapport à l'édition précédente:

a) des unités techniques facultatives ont été ajoutées à la définition des alarmes RateOfChange;
b) afin de respecter le modèle IEC 62682, les éléments suivants ont été ajoutés:
   - états d'AlarmConditionType: Suppression, Silence, OutOfService, Latched;

- Propriétés d'AlarmConditionType: OnDelay, OffDelay, FirstInGroup, ReAlarmTime;
- nouveaux types d'alarmes: DiscrepancyAlarm, DeviationAlarm, InstrumentDiagnosticAlarm, SystemDiagnosticAlarm;

c) ajout d'une annexe qui spécifie la manière dont les concepts de cette partie d'OPC UA assurent la correspondance avec l'IEC 62682 et l'ISA 18.2;

d) nouvelles ConditionClasses ajoutées: Safety, HighlyManaged, Statistical, Testing, Training;

e) ajout de l'AlarmType CertificateExpiration;

f) ajout d'un modèle de Mesures d'Alarme.

Le texte de cette Norme internationale est issu des documents suivants:

FDIS	Rapport de vote
65E/709/FDIS	65E/727/RVD

Le rapport de vote indiqué dans le tableau ci-dessus donne toute information sur le vote ayant abouti à l'approbation de cette Norme internationale.

Ce document a été rédigé selon les Directives ISO/IEC, Partie 2.

Dans l'ensemble du présent document et dans les autres parties de la série IEC 62541, certaines conventions de document sont utilisées:

Le format *italique* est utilisé pour mettre en évidence un terme défini ou une définition qui apparaît à l'article "Termes et définitions" dans l'une des parties de la série IEC 62541.

Le format *italique* est également utilisé pour mettre en évidence le nom d'un paramètre d'entrée ou de sortie de service, ou le nom d'une structure ou d'un élément de structure habituellement défini dans les tableaux.

Par ailleurs, les *termes* et les *noms en italique* sont, à quelques exceptions près, écrits en camel-case (pratique qui consiste à joindre, sans espace, les éléments des mots ou expressions composés, la première lettre de chaque élément étant en majuscule). Par exemple, le terme défini est *AddressSpace* et non Espace d'adressage. Cela permet de mieux comprendre qu'il existe une définition unique pour *AddressSpace*, et non deux définitions distinctes pour Espace et pour Adressage.

Une liste de toutes les parties de la série IEC 62541, publiées sous le titre général *Architecture unifiée OPC Unified Architecture*, peut être consultée sur le site web de l'IEC.

Le comité a décidé que le contenu de ce document ne sera pas modifié avant la date de stabilité indiquée sur le site web de l'IEC sous "http://webstore.iec.ch" dans les données relatives au document recherché. A cette date, le document sera

- reconduit,
- supprimé,
- remplacé par une édition révisée, ou
- amendé.

IMPORTANT – Le logo *"colour inside"* qui se trouve sur la page de couverture de cette publication indique qu'elle contient des couleurs qui sont considérées comme utiles à une bonne compréhension de son contenu. Les utilisateurs devraient, par conséquent, imprimer cette publication en utilisant une imprimante couleur.

ARCHITECTURE UNIFIÉE OPC –

Partie 9: Alarmes et Conditions

1 Domaine d'application

La présente partie de l'IEC 62541 spécifie la représentation des *Alarmes* et des *Conditions* dans l'Architecture unifiée OPC. Il comprend la représentation par le *Modèle d'information* des *Alarmes* et des *Conditions* dans l'espace d'adressage OPC UA. Les autres aspects des systèmes d'alarme tels que la philosophie d'alarme, le cycle de vie, le temps de réponse de l'alarme, les types d'alarmes et de nombreux autres détails figurent dans des documents tels que l'IEC 62682 et l'ISA 18.2. Le *Modèle d'information* sur les *Alarmes* et les *Conditions* de la présente spécification est conçu conformément à l'IEC 62682 et à l'ISA 18.2.

2 Références normatives

Les documents suivants cités dans le texte constituent, pour tout ou partie de leur contenu, des exigences du présent document. Pour les références datées, seule l'édition citée s'applique. Pour les références non datées, la dernière édition du document de référence s'applique (y compris les éventuels amendements).

IEC TR 62541-1, *OPC unified architecture – Part 1: Overview and concepts* (disponible en anglais seulement)

IEC 62541-3, Architecture unifiée OPC – Partie 3: Modèle d'espace d'adressage

IEC 62541-4, Architecture unifie OPC – Partie 4: Services

IEC 62541-5, *Architecture unifiée OPC – Partie 5: Modèle d'information*

IEC 62541-6, *Architecture unifiée OPC – Partie 6: Mappings*

IEC 62541-7, *Architecture unifiée OPC – Partie 7: Profils*

IEC 62541-8, *Architecture unifiée OPC – Partie 8: Accès aux données*

IEC 62541-11, *Architecture unifiée OPC – Partie 11: Accès à l'historique*

IEC 62682: Gestion de systèmes d'alarme dans les industries de transformation

EEMUA: 2nd Edition EEMUA 191 – *Alarm System – A guide to design, management and procurement (Appendixes 6, 7, 8, 9)* (disponible en anglais seulement), disponible à l'adresse https://www.eemua.org/Products/Publications/Print/EEMUA-Publication-191.aspx

3 Termes, définitions, termes abrégés et types de données utilisés

3.1 Termes et définitions

Pour les besoins du présent document, les termes et définitions donnés dans l'IEC TR 62541-1, l'IEC 62541-3, l'IEC 62541-4, l'IEC 62541-5 ainsi que les suivants s'appliquent.

L'ISO et l'IEC tiennent à jour des bases de données terminologiques destinées à être utilisées en normalisation, consultables aux adresses suivantes:

- IEC Electropedia: disponible à l'adresse http://www.electropedia.org/
- ISO Online browsing platform: disponible à l'adresse http://www.iso.org/obp

3.1.1
Acquittement
action de l'*Opérateur* qui indique la reconnaissance d'une *Alarme*

Note 1 à l'article: Cette définition est tirée de l'EEMUA. Le terme "Acceptation" est un autre terme courant utilisé pour décrire l'*Acquittement*. Les deux termes peuvent être utilisés de manière interchangeable. Le présent document utilise *Acquittement*.

3.1.2
Active
état d'une *Alarme* qui indique que la situation que l'*Alarme* représente existe actuellement

Note 1 à l'article: D'autres termes courants définis par l'EEMUA sont "en cours" pour une *Alarme active* et "effacée" lorsque la *Condition* est revenue à la normale et n'est plus *Active*.

3.1.3
AdaptiveAlarm
Alarme dont le point de consigne ou les limites sont modifiés par un algorithme

Note 1 à l'article: Les *AdaptiveAlarms* sont des alarmes qui sont automatiquement ajustées par des algorithmes. Ces algorithmes peuvent détecter des conditions au sein d'une installation et modifier les points de consigne ou les limites afin d'empêcher les alarmes. Très souvent, ces modifications se produisent sans interaction de l'*Opérateur*.

3.1.4
AlarmFlood
condition pendant laquelle le taux d'alarme est supérieur à ce que l'*Opérateur* peut gérer efficacement

Note 1 à l'article: L'OPC UA ne définit pas les conditions qui seraient jugées comme des afflux d'alarmes; ces conditions sont définies dans d'autres spécifications telles que l'IEC 62682 ou l'ISA 18.2.

3.1.5
AlarmSuppressionGroup
groupe d'*Alarmes* utilisé pour supprimer d'autres *Alarmes*

Note 1 à l'article: Un *AlarmSuppressionGroup* est une instance d'un *AlarmGroupType* utilisé pour supprimer d'autres *Alarmes*. Si une *Alarme* du groupe est active, alors l'*AlarmSuppressionGroup* est actif. Si toutes les *Alarmes* de l'*AlarmSuppressionGroup* sont inactives, alors l'*AlarmSuppressionGroup* est inactif.

Note 2 à l'article: L'*Alarme* à affecter référence les *AlarmSuppressionGroups* avec un *ReferenceType HasAlarmSuppressionGroup*.

3.1.6
ConditionClass
ensemble de *Conditions* qui indique le domaine ou le but pour lequel une certaine *Condition* est utilisée

Note 1 à l'article: Un certain nombre de *ConditionClasses* de haut niveau sont définies dans la présente spécification. Les fournisseurs ou les organisations peuvent obtenir des classes plus concrètes ou définir des classes de haut niveau différentes.

3.1.7
ConditionBranch
état spécifique d'une *Condition*

Note 1 à l'article: Le *Serveur* peut maintenir des *ConditionBranches* pour l'état courant ainsi que pour des états antérieurs.

3.1.8
ConditionSource
élément sur lequel repose une *Condition* spécifique ou auquel celle-ci se rapporte

Note 1 à l'article: Il s'agit habituellement d'une *Variable* représentant un marqueur de processus (par exemple FIC101) ou d'un *Objet* représentant un appareil ou un sous-système.

Note 2 à l'article: Dans des *Evénements* générés pour des *Conditions*, la *Propriété SourceNode* (héritée du *BaseEventType*) contient le *NodeId* de la *ConditionSource*.

3.1.9
confirmer
action de l'*Opérateur* informant le *Serveur* qu'une action corrective a été entreprise pour traiter la cause de l'*Alarme*

3.1.10
désactiver
action de configuration d'un système de telle manière que l'*Alarme* n'est pas générée même si la *Condition* d'*Alarme* de base est présente

Note 1 à l'article: Cette définition est tirée de l'EEMUA et est décrite plus en détail dans l'EEMUA.

Note 2 à l'article: Dans l'IEC 62682, "désactiver" est référencé sous la dénomination "Hors service".

3.1.11
LatchingAlarm
alarme qui reste à l'état d'alarme après que la condition de processus est revenue à la normale et qui nécessite une réinitialisation de l'*Opérateur* avant que l'alarme revienne à la normale

Note 1 à l'article: Les alarmes à enclenchement sont en général des alarmes d'anomalie, lorsqu'une action ne se produit pas dans un délai spécifique. Lorsque l'action se produit, l'alarme reste active jusqu'à sa réinitialisation.

3.1.12
Opérateur
utilisateur spécial qui est affecté à la surveillance et à la commande d'une partie d'un processus

Note 1 à l'article: "Un membre d'une équipe opérations affecté à la surveillance et à la commande d'une partie du processus et travaillant au niveau de la Console du système de commande" selon la définition de l'EEMUA. Dans le présent document, un Opérateur est un utilisateur spécial. Toutes les descriptions qui s'appliquent aux utilisateurs généraux s'appliquent aussi aux Opérateurs.

3.1.13
Rafraîchissement
action de mise à jour d'un *Abonnement* à des *Evénements* qui fournit toutes les *Alarmes* jugées comme étant *Retenues*

Note 1 à l'article: Cette notion est définie plus en détail dans l'EEMUA.

3.1.14
Retain
Alarme dans un état intéressant pour un *Client* qui souhaite synchroniser son état de *Conditions* à l'état du *Serveur*

3.1.15
Suspension
moyen par lequel l'*Opérateur* est capable d'empêcher temporairement qu'une *Alarme* ne s'affiche à l'attention de l'*Opérateur* lorsqu'elle cause une gêne pour l'*Opérateur*

Note 1 à l'article: "Une *Alarme* suspendue est retirée de la liste et n'est pas annoncée de nouveau tant qu'elle sa suspension n'est pas annulée" selon la définition de l'EEMUA.

3.1.16
Supprimer
action de déterminer qu'il convient qu'une *Alarme* ne se produise pas

Note 1 à l'article: "Une Alarme est supprimée lorsque des critères logiques sont appliqués pour déterminer qu'il convient que l'Alarme ne se produise pas, même si la Condition de base de l'Alarme (valeur de consigne d'Alarme dépassée par exemple) est présente" selon la définition de l'EEMUA. Dans l'IEC 62682, les Alarmes Supprimées sont également décrites comme étant "Supprimées par Conception", le système étant conçu pour Supprimer une Alarme en présence de certains critères. Par exemple, si une unité de traitement est déconnectée, les alarmes de niveau bas sont Supprimées pour tous les équipements de l'unité déconnectée.

3.2 Termes abrégés

A&E Alarme et Evénement (comme utilisé pour le COM OPC)

COM (Microsoft Windows) Component Object Model (Modèle d'objet composant)

DA data access (accès aux données)

UA Unified Architecture (Architecture unifiée)

3.3 Types de données utilisés

Les Tableaux 1 et 2 décrivent les types de données qui sont utilisés tout au long du présent document. Ces types sont répartis en deux tableaux. Les types de données de base définis dans l'IEC 62541-3 sont consignés dans le Tableau 1. Les types de base et les types de données de base définis dans l'IEC 62541-4 sont consignés dans le Tableau 2.

Tableau 1 – Types de paramètres définis dans l'IEC 62541-3

Type de paramètre
Argument
BaseDataType
NodeId
LocalizedText
Booléen
ByteString
Double
Durée
Chaîne
UInt16
Int32
UtcTime

Tableau 2 – Types de paramètres définis dans l'IEC 62541-4

Type de paramètre
IntegerId
StatusCode

4 Concepts

4.1 Généralités

Le présent document définit un *Modèle d'information* pour les *Conditions*, les *Conditions* de dialogue et les *Alarmes*, y compris les fonctionnalités d'acquittement. Ce modèle s'appuie sur la gestion d'Evénements de base définie dans l'IEC 62541-3, IEC 62541-4 et l'IEC 62541-5 et les complète. Ce *Modèle d'information* peut aussi être étendu pour prendre en charge les autres besoins de domaines spécifiques. Les détails des aspects du Modèle d'information pris en charge sont définis par le biais des Profils (voir l'IEC 62541-7 pour la définition des Profils). Certains systèmes peuvent présenter les Evénements et Conditions historiques par le biais du Cadre d'Accès à l'historique (voir l'IEC 62541-11 pour la définition des Evénements historiques).

4.2 Conditions

Les *Conditions* sont utilisées pour représenter l'état d'un système ou de l'un de ses composants. Certains exemples communs sont:

- une température dépassant une limite configurée;
- un appareil présentant la nécessité d'une maintenance;
- un processus par lots qui exige que l'utilisateur confirme une certaine étape du processus avant de se poursuivre.

Chaque instance de *Condition* est d'un *ConditionType* spécifique. Le *ConditionType* et les types résultants sont des sous-types du *BaseEventType* (voir l'IEC 62541-3 et l'IEC 62541-5). La présente partie définit les types qui sont communs à de nombreux secteurs. Il est prévu que les fournisseurs ou autres groupes de normalisation définissent des *ConditionTypes* supplémentaires obtenus à partir des types de base communs définis dans la présente partie. Les *ConditionTypes* pris en charge par un *Serveur* sont présentés dans l'*AddressSpace* du *Serveur*.

Les instances de *Condition* sont des mises en œuvre spécifiques d'un *ConditionType*. Il incombe au *Serveur* de décider si, oui ou non, ces instances sont également présentées dans l'*AddressSpace* du *Serveur*. Le 4.10 fournit un contexte supplémentaire concernant les instances de *Condition*. Les instances de *Condition* doivent avoir un identificateur unique pour les différencier des autres instances, qu'elles soient présentées ou non dans l'*AddressSpace*.

Comme mentionné ci-dessus, les *Conditions* représentent l'état d'un système ou de l'un de ses composants. Cependant, dans certains cas, les états antérieurs qui nécessitent encore de l'attention doivent également être maintenus. Les *ConditionBranches* sont introduites afin de traiter de cette exigence et établir la distinction entre l'état courant et les états antérieurs. Chaque *ConditionBranch* a un BranchId qui la différencie des autres branches de la même instance de *Condition*. La *ConditionBranch* qui représente l'état courant de la *Condition* (le tronc) a un BranchId NULL. Les *Serveurs* peuvent générer des *Notifications d'Evénements* pour chaque branche. Lorsqu'il n'est pas nécessaire que l'état représenté par une *ConditionBranch* fasse l'objet d'attention supplémentaire, une *Notification d'Evénement* finale pour cette branche a sa *Propriété Retain* définie sur False. Le 4.4 fournit plus d'informations et des cas d'utilisation. La conservation d'états antérieurs et donc la prise en charge de plusieurs branches sont facultatives pour les *Serveurs*.

D'un point de vue conceptuel, la durée de vie de l'instance de *Condition* est indépendante de son état. Cependant, les *Serveurs* peuvent fournir l'accès à des instances de *Condition* uniquement tant qu'il existe des *ConditionBranches*.

Le modèle d'état de base d'une *Condition* est représenté à la Figure 1. Il est étendu par les divers sous-types de *Conditions* définis dans le présent document et peut être étendu davantage par les fournisseurs ou autres groupes de normalisation. Les états principaux d'une *Condition* sont "Disabled" et "Enabled". L'état *Disabled* est destiné à permettre la désactivation des *Conditions* au niveau du *Serveur* ou en dessous du *Serveur* (dans un appareil ou un certain système sous-jacent). L'état *Enabled* est normalement étendu par l'ajout de sous-états.

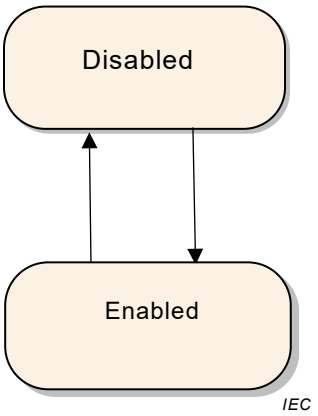


Figure 1 – Modèle d'état de base d'une Condition

Un passage à l'état *Disabled* se traduit par un *Evénement* de *Condition*, mais il n'est pas généré de *Notifications* d'*Evénements* consécutives tant que la *Condition* n'est pas revenue à l'état Enabled.

Lorsqu'une *Condition* entre dans l'état Enabled, ce passage et tous les passages consécutifs se traduisent par la création d'*Evénements* de *Condition* par le *Serveur*.

Lorsque l'*Audit* est pris en charge par un *Serveur*, l'action suivante liée à un *Audit* doit être réalisée. Le *Serveur* crée des *AuditEvents* pour les opérations *Enable* et *Disable* (soit par invocation d'une *Méthode* ou par certains moyens spécifiques au *Serveur*/fournisseur), plutôt que de créer une *Notification* d'*AuditEvent* pour chaque instance de *Condition* activée ou désactivée. Pour plus d'informations, voir la définition d'*AuditConditionEnableEventType* en 5.10.2. Les *AuditEvents* sont également générés pour toute autre action de l'*Opérateur* qui entraîne des changements des *Conditions*.

4.3 Conditions acquittables

Les *AcknowledgeableConditions* sont des sous-types du *ConditionType* de base. Les *AcknowledgeableConditions* présentent les états pour indiquer qu'une *Condition* doit être acquittée ou confirmée.

Un *AckedState* et un *ConfirmedState* étendent l'état *Enabled* défini par la *Condition*. Le modèle d'état est représenté à la Figure 2. L'état activé est étendu en ajoutant l'*AckedState* et (facultativement) le *ConfirmedState*.

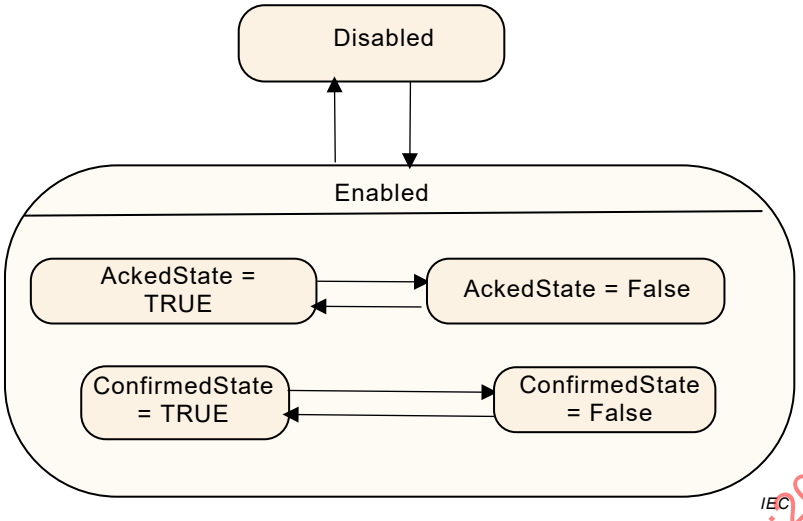


Figure 2 – Modèle d'état des AcknowledgeableConditions

L'acquittement de la transition peut venir du *Client* ou peut être dû à une certaine logique interne au *Serveur*. Par exemple, l'acquittement d'une *Condition* connexe peut faire que cette *Condition* devienne acquittée, ou la *Condition* peut être réglée pour s'*Acquitter* elle-même automatiquement lorsque la situation acquittable disparaît.

Deux modèles d'états d'*Acquittement* sont pris en charge dans le présent document. Chacun de ces modèles d'états peut être étendu afin de prendre en charge des situations d'acquittement plus complexes.

Le modèle d'état de base d'*Acquittement* est représenté à la Figure 3. Ce modèle définit un AckedState. Les changements d'état spécifiques qui se traduisent par un changement vers l'état dépendent de la mise en œuvre du Serveur. Par exemple, dans les modèles d'*Alarme* types, le changement se limite à une transition vers l'état *Active* ou à des transitions au sein de l'état *Active*. Cependant, des modèles plus complexes peuvent aussi admettre des changements vers l'AckedState lorsque la *Condition* passe à un état inactif.

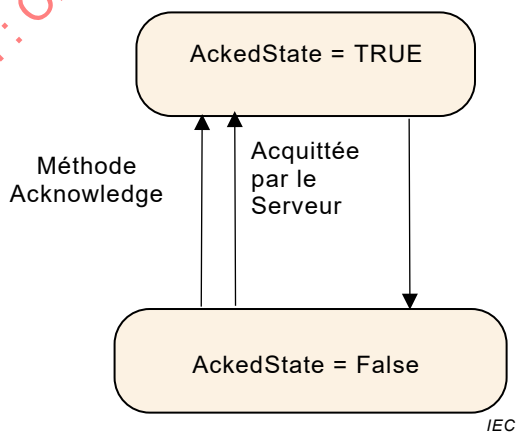


Figure 3 – Modèle d'état d'Acquittement

Un modèle d'état plus complexe qui ajoute une confirmation à l'*Acquittement* de base est représenté à la Figure 4. Le modèle d'*Acquittement confirmé* est habituellement utilisé pour établir une différence entre le fait d'acquitter la présence d'une *Condition* et le fait d'avoir entrepris une action pour traiter la *Condition*. Par exemple, un *Opérateur* recevant une *Notification* de température élevée du moteur appelle la *Méthode Acknowledge* pour signaler au *Serveur* qu'une température élevée a été observée. L'*Opérateur* entreprend ensuite une certaine action, comme diminuer la charge sur le moteur afin de faire baisser la température.

L'*Opérateur* appelle ensuite la *Méthode Confirm* pour signaler au *Serveur* qu'une action corrective a été entreprise.

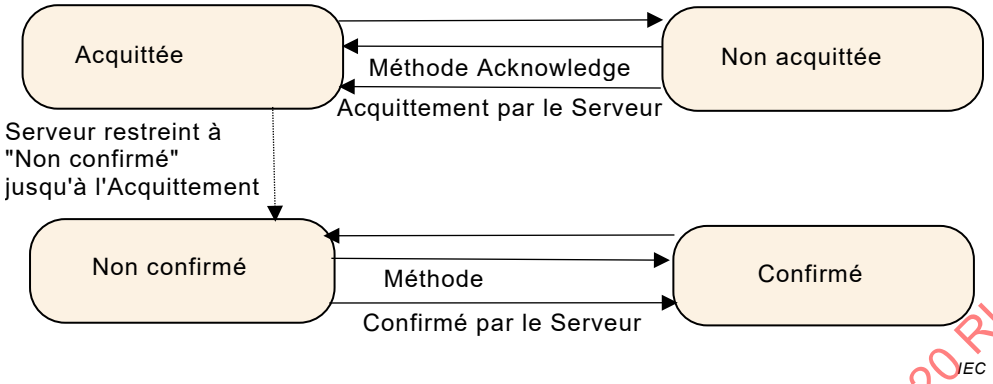


Figure 4 – Modèle d'état d'un Acquittement confirmé

4.4 Etats antérieurs des Conditions

Certains systèmes exigent que les états antérieurs d'une *Condition* soient conservés pendant un certain temps. Un cas d'utilisation commun est le processus d'acquittement. Dans certains environnements, tant l'acquittement du passage à l'état *Active* que l'acquittement du passage à l'état inactif sont exigés. Les systèmes avec des règles de sécurité strictes exigent parfois que chaque transition vers l'état *Active* soit acquittée. Dans les situations où les changements d'état se succèdent rapidement, il peut exister plusieurs états non acquittés, et le *Serveur* doit maintenir les *ConditionBranches* pour tous les états antérieurs non acquittés. Ces branches sont supprimées dès qu'elles ont été acquittées ou si elles ont atteint leur état final.

Les *ConditionBranches multiples* peuvent également être utilisées pour d'autres cas d'utilisation où des états antérieurs instantanés d'une *Condition* exigent des actions supplémentaires.

4.5 Synchronisation des états d'une condition

Lorsqu'un *Client* s'abonne à des *Evénements*, la *Notification* des transitions commence au moment de l'*Abonnement*. L'état courant existant n'est pas signalé. Cela signifie par exemple que les *Clients* ne sont pas informés des *Alarmes* actuellement *Actives* jusqu'à ce qu'un nouveau changement d'état se produise.

Les *Clients* peuvent obtenir l'état courant de toutes les instances de *Condition* qui sont dans un état intéressant en demandant le *Rafraîchissement* d'un *Abonnement*. Il convient de noter que le *Rafraîchissement* n'est pas un moyen de réitération systématique, car le *Serveur* n'est pas tenu de conserver un historique des *Evénements*.

Les *Clients* demandent un *Rafraîchissement* en appelant la *Méthode* ConditionRefresh. Le *Serveur* répond par un *Evénement de RefreshStartEventType*. Cet *Evénement* est suivi des *Conditions Retenues*. Le *Serveur* peut aussi envoyer de nouvelles *Notifications* d'*Evénements* parsemées de *Notifications* d'*Evénements* de *Rafraîchissement*. Lorsque le *Serveur* en a terminé avec le *Rafraîchissement*, un *RefreshEndEvent* est produit et marque la fin du Rafraîchissement. Les *Clients* doivent vérifier toutes les *Notifications* d'*Evénements* pour détecter une *ConditionBranch* afin d'éviter d'écraser un nouvel état délivré en même temps qu'un état plus ancien par le processus de *Rafraîchissement*. Si une *ConditionBranch* existe, la *Condition* courante doit alors être consignée. Ceci est vrai même si le seul élément intéressant concernant la *Condition* est que des *ConditionBranches* existent. Cela permettre à un *Client* de représenter avec précision l'état courant de la *Condition*.

Un *Client* souhaitant afficher le statut courant des *Alarmes* et des *Conditions* (appelé "affichage d'*Alarmes* courantes") utiliserait la logique suivante pour traiter les *Notifications* d'*Evénements* de *Rafraîchissement*. A la réception de l'*Evénement* du *RefreshStartEventType*, le *Client* marque toutes les *Conditions Retenues* comme suspectes. Le *Client* ajoute tous les éventuels nouveaux *Evénements* qui ont été reçus pendant le Rafraîchissement sans les marquer comme suspects. Le *Client* enlève également le fanion "suspect" de toutes les éventuelles *Conditions Retenues* qui sont renvoyées comme partie intégrante du Rafraîchissement. Lorsque le *Client* reçoit un *RefreshEndEvent*, il retire tous les éventuels *Evénements* suspects restants, car ils ne s'appliquent plus.

Il convient de noter les éléments suivants en ce qui concerne ConditionRefresh:

- comme décrit en 4.4, certains systèmes exigent que les états antérieurs d'une *Condition* soient conservés pendant un certain temps. Certains *Serveurs*, en particulier ceux qui exigent l'acquittement des états antérieurs, maintiennent des *ConditionBranches* distinctes pour les états antérieurs qui nécessitent encore de l'attention.

  ConditionRefresh doit produire des *Notifications* d'*Evénements* pour tous les états intéressants (courants et antérieurs) d'une instance de *Condition*, et les *Clients* peuvent par conséquent recevoir plusieurs *Evénements* pour une même instance de *Condition* avec des BranchIds (identificateurs de branche) différents;

- dans certaines circonstances, un *Serveur* peut ne pas être capable de garantir que le *Client* est totalement synchronisé avec l'état courant des instances de *Condition*. Par exemple, si le système sous-jacent représenté par le *Serveur* est réinitialisé ou que les communications sont perdues pendant un certain temps, il peut être nécessaire pour le *Serveur* de se resynchroniser au système sous-jacent. Dans de tels cas, le *Serveur* doit envoyer un *Evénement* de type *RefreshRequiredEventType* pour annoncer au *Client* qu'un *Rafraîchissement* peut être nécessaire. Il convient qu'un *Client* recevant cet *Evénement* spécial déclenche un *ConditionRefresh* comme indiqué dans le présent paragraphe;

- afin de s'assurer qu'un *Client* est toujours informé, les trois *EventTypes* spéciaux (*RefreshEndEventType, RefreshStartEventType* et *RefreshRequiredEventType*) ignorent le filtrage de contenu d'*Evénement* associé à un *Abonnement* et sont toujours délivrés au *Client*;

- *ConditionRefresh* s'applique à un *Abonnement*. Si plusieurs Notifications d'Evénement sont incluses dans le même *Abonnement*, toutes les *Notifications d'Evénement* sont rafraîchies.

4.6 Sévérité, qualité et commentaire

Commentaire, sévérité et qualité sont des éléments importants de *Conditions*, et tout changement qui leur est apporté engendre des *Notifications* d'*Evénement*.

La Sévérité d'une *Condition* est héritée du modèle d'*Evénement* de base défini dans l'IEC 62541-5. Elle indique l'urgence de la *Condition* et elle est également communément appelée "priorité", notamment en rapport avec les *Alarmes* appartenant au *ProcessConditionClassType*.

Un Commentaire est une chaîne créée par l'utilisateur qui doit être associée à un certain état d'une *Condition*.

La Qualité se réfère à la qualité de la ou des valeurs de données sur lesquelles cette *Condition* est fondée. Etant donné qu'une *Condition* est habituellement fondée sur une ou plusieurs *Variables*, la *Condition* hérite de la qualité de ces *Variables*. Par exemple, si la valeur de processus est "Uncertain", la *Condition* "Level Alarm" est également douteuse. Lorsque deux variables ou plus sont représentées par une condition donnée ou si la condition provient d'un système sous-jacent, et lorsqu'aucun mapping direct avec une variable n'est disponible, il revient à l'application de déterminer le niveau de qualité affiché comme partie intégrante de la condition.

4.7 Dialogues

Les Dialogues sont des *ConditionTypes* utilisés par un *Serveur* pour demander des données d'utilisateur. Ils sont habituellement utilisés lorsqu'un *Serveur* est entré dans un état qui exige l'intervention d'un *Client*. Par exemple, un *Serveur* surveillant une machine à papier indique qu'un rouleau de papier a été enroulé et est prêt à l'inspection. Le *Serveur* activerait une *Condition* de Dialogue indiquant à l'utilisateur qu'une inspection est exigée. Lorsque l'inspection a eu lieu, l'utilisateur répond en informant le *Serveur* que l'inspection est acceptée ou n'est pas acceptée, permettant au processus de se poursuivre.

4.8 Alarmes

Les *Alarmes* sont des spécialisations des *AcknowledgeableConditions* qui ajoutent à une *Condition* les concepts d'état *Active* et d'autres états tels que *Shelving* et *Suppressed*. Le modèle d'état est représenté à la Figure 5. Le modèle complet avec tous les états est défini en 5.8.

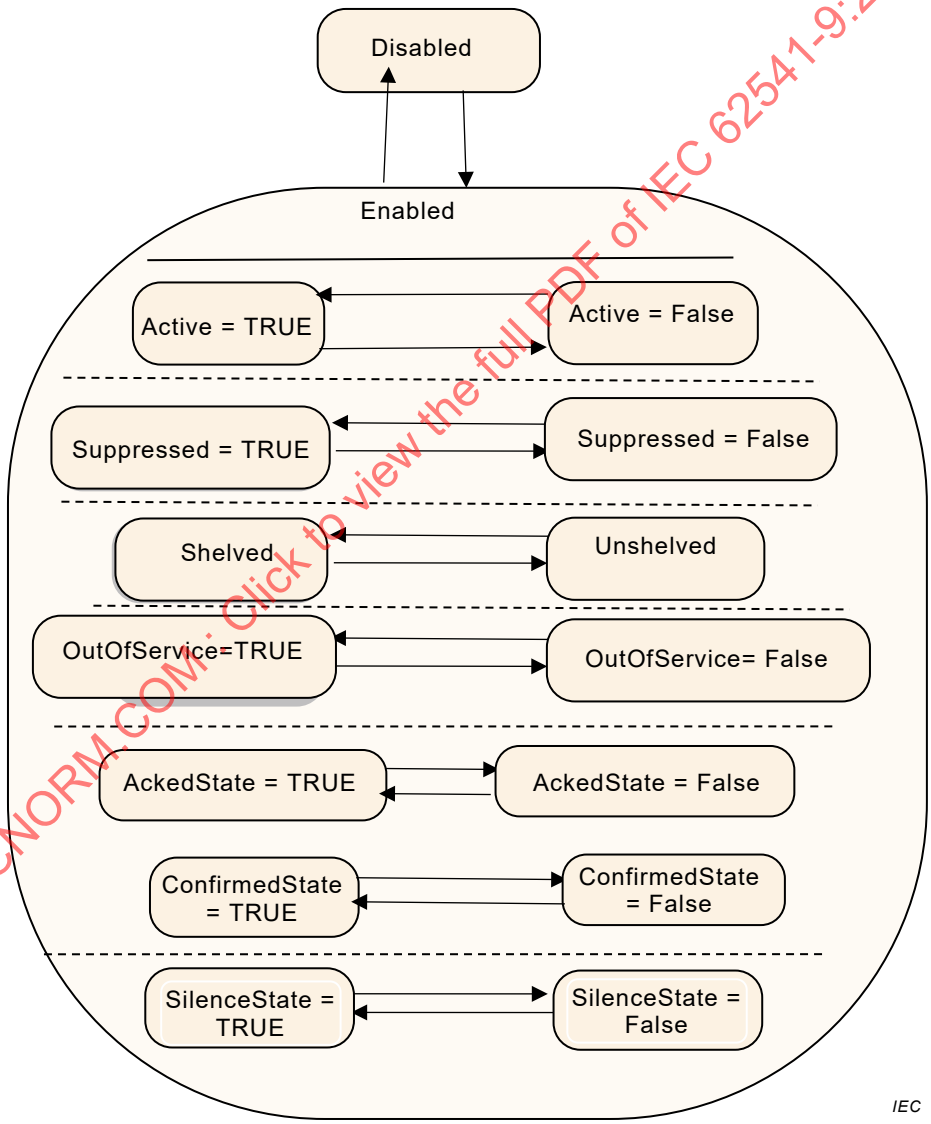


Figure 5 – Modèle de diagramme d'états des alarmes

Une *Alarme* à l'état *Active* indique que la situation décrite par la *Condition* est actuellement présente. Lorsqu'une *Alarme* est dans l'état inactif, elle indique une situation qui est revenue à un état normal.

Certains sous-types d'*Alarmes* introduisent des sous-états de l'état *Active*. Par exemple, une *Alarme* représentant une température peut fournir aussi bien un état de niveau élevé qu'un état de niveau élevé critique (voir le paragraphe suivant).

L'état *Shelving* peut être établi par un *Opérateur* par le biais des *Méthodes* OPC UA. L'état *Suppressed* est établi en interne par le *Serveur* pour des raisons spécifiques au système. Les systèmes d'*Alarme* mettent habituellement en œuvre les fonctionnalités de suppression, de mise hors service et de suspension pour éviter que les *Opérateurs* ne soient submergés par des flots d'*Alarmes* en limitant le nombre d'*Alarmes* qu'un *Opérateur* voit sur l'affichage des *Alarmes* en cours. Cela est accompli par le réglage du fanion *SuppressedOrShelved* sur des *Alarmes* dépendantes de second ordre et/ou des *Alarmes* de moindre sévérité, amenant l'*Opérateur* à se concentrer sur les questions les plus critiques.

Les états "Shelved", "Out of Service" et "Suppressed" diffèrent de l'état *Disabled*, car les *Alarmes* sont encore totalement fonctionnelles et peuvent être incluses dans des *Notifications* d'*Abonnement* adressées à un *Client*.

Les alarmes suivent une chronologie type représentée à la Figure 6. Plusieurs temps de retard et états qu'elles peuvent occuper leur sont associés. Le but d'un système d'alarme est d'informer les *Opérateurs* des conditions dans le temps prévu et de permettre à l'*Opérateur* de prendre des mesures avant que des conséquences se produisent. Les conséquences peuvent être économiques (le produit n'est pas utilisable et doit être éliminé), physiques (débordement de réservoir), liées à la sécurité (un incendie ou une explosion peut se produire) ou toute autre possibilité. Habituellement, si aucune mesure n'est prise par rapport à une alarme après un certain temps, le processus passe un point au-delà duquel des conséquences commencent à se produire. Le modèle d'*Alarme* OPC UA décrit ces états, retards et mesures.

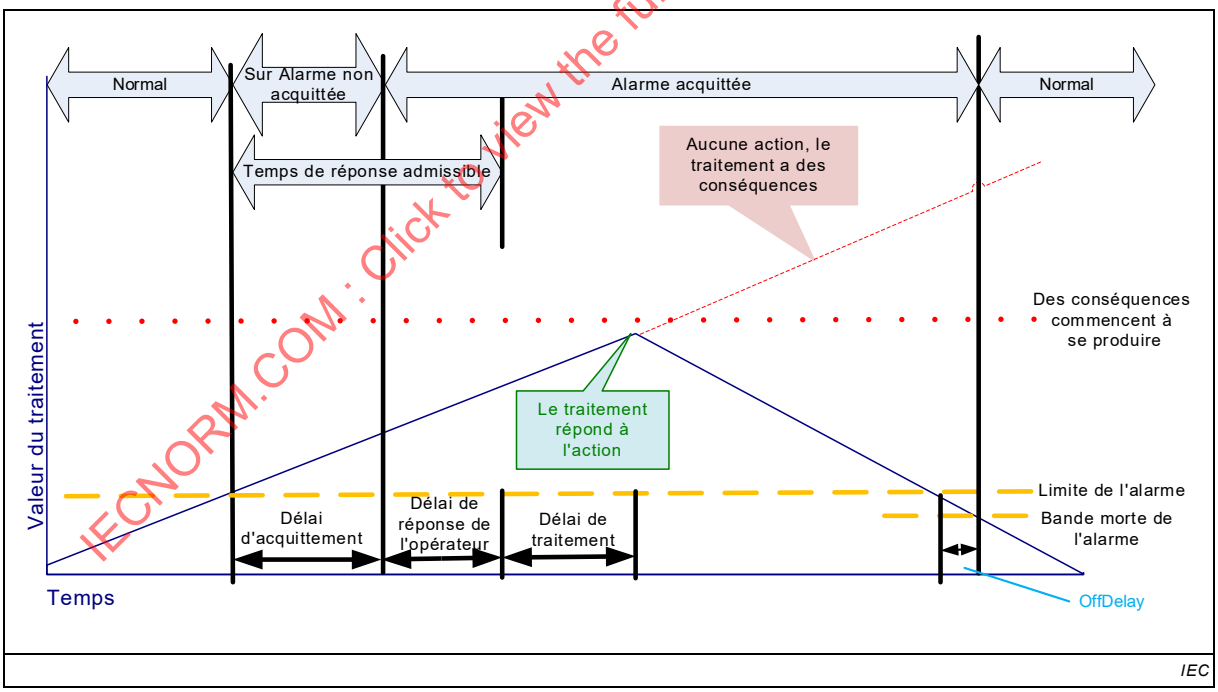


Figure 6 – Exemple de Chronologie d'Alarme type

4.9 Etats actifs multiples

Dans certains cas, il est souhaitable de définir plus en détail l'état *Active* d'une *Alarme* en fournissant un diagramme de sous-états pour l'état *Active*. Par exemple, une *Alarme* de niveau à états multiples lorsque son état est *Active* peut s'inscrire dans l'un des sous-états suivants: LowLow, Low, High ou HighHigh. Le modèle d'état est représenté à la Figure 7.

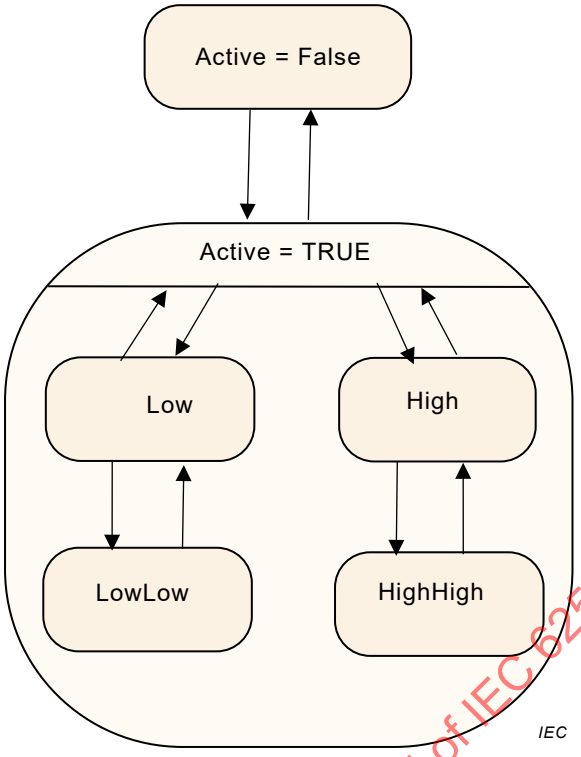


Figure 7 – Exemple d'états actifs multiples

Avec le modèle d'*Alarme* à états multiples, les transitions d'états parmi les sous-états d'*Active* sont admises sans entraîner de transition hors de l'état *Active*.

Pour prendre en charge différents cas d'utilisation, un modèle (mutuellement) exclusif et un modèle non exclusif sont pris en charge.

Exclusif signifie que l'*Alarme* ne peut être que dans un seul sous-état à la fois. Si, par exemple, la température dépasse la limite HighHigh, l'*Alarme* de niveau exclusive associée est dans le sous-état "HighHigh" et non dans le sous-état "High".

Certains systèmes d'*Alarme* autorisent, toutefois, la coexistence de plusieurs sous-états en parallèle. Cela est qualifié de "non exclusif". Dans l'exemple précédent où la température dépasse la limite HighHigh, une *Alarme* de niveau non exclusive est à la fois dans le sous-état "High" et dans le sous-état "HighHigh".

4.10 Instances de Condition dans l'AddressSpace

Sachant que les *Conditions* ont toujours un état (*Enabled* ou *Disabled*) et éventuellement plusieurs sous-états, il est logique que des instances de *Condition* soient présentes dans l'*AddressSpace*. Si le *Serveur* présente des instances de *Condition*, celles-ci apparaissent habituellement dans l'*AddressSpace* comme composants des *Objets* qui les "possèdent". Par exemple, un transmetteur de température qui comporte une *Alarme* intégrée de température élevée apparaîtrait dans l'*AddressSpace* comme une instance d'un certain *Objet* transmetteur de température avec une *Référence HasComponent* à une instance d'un *LimitAlarmType*.

La disponibilité d'instances donne la possibilité à des *Clients* d'accès aux données de surveiller l'état courant de la *Condition* en s'abonnant aux valeurs d'*Attributs* des *Nœuds* de *Variables*. Les valeurs des nœuds peuvent ne pas toujours correspondre à la valeur qui apparaît dans les *Evénements*, elles peuvent être plus récentes que celles de l'*Evénement*.

Alors que le fait de présenter des instances de *Condition* dans l'*AddressSpace* n'est pas toujours possible, le faire permettre l'interaction directe (lecture, écriture et invocation de *Méthode*) avec une instance spécifique de *Condition*. Par exemple, si une instance de *Condition* n'est pas présentée, il n'y a aucun moyen d'invoquer la *Méthode Enable* ou *Disable* pour l'instance de *Condition* spécifique.

4.11 Conduite d'audits pour les Alarmes et les Conditions

La série IEC 62541 inclut des dispositions pour la conduite d'audits. La conduite d'audits est un concept important de sécurité et de suivi. Les enregistrements d'audit fournissent les informations de type "Qui", "Quand" et "Quoi" concernant les interactions utilisateur avec un système. Ces enregistrements d'audit sont particulièrement importants lorsque la gestion des *Alarmes* est envisagée. Les *Alarmes* constituent l'instrument type pour informer un utilisateur que quelque chose exige son attention. Un enregistrement de la manière dont l'utilisateur réagit à ces informations est indispensable dans de nombreux cas. Les enregistrements d'audit sont générés par tous les appels de *Méthode* qui ont une incidence sur l'état du système, par exemple un appel de la *Méthode Acknowledge* génère un *Evénement AuditConditionAcknowledgeEventType*.

Les *AuditEventTypes* normalisés définis dans l'IEC 62541-5 incluent déjà les champs exigés pour les enregistrements d'audit relatifs à la *Condition*. Pour permettre le filtrage et le regroupement, le présent document définit un certain nombre de sous-types des *AuditEventTypes*, mais sans leur ajouter de nouveaux champs.

Le présent document décrit l'*AuditEventType* que chaque *Méthode* est tenue de générer. Par exemple, la *Méthode Disable* comporte une *Référence AlwaysGeneratesEvent* à un *AuditConditionEnableEventType*. Un *Evénement* de ce type doit être généré pour chaque invocation de la *Méthode*. L'*Evénement* d'audit décrit l'interaction de l'utilisateur avec le système; dans certains cas, cette interaction peut affecter plusieurs *Conditions* ou être liée à plusieurs états.

5 Modèle

5.1 Généralités

Le modèle d'*Alarme* et de *Condition* étend le modèle d'*Evénement* de base OPC UA en définissant divers *Types* d'*Evénements* fondés sur le *BaseEventType*. Tous les *Types* d'*Evénements* définis dans le présent document peuvent être étendus davantage pour former des *Types* d'*Alarmes* et de *Conditions* spécifiques au domaine ou au *Serveur*.

Des instances des *Types* d'*Alarmes* et de *Conditions* peuvent être facultativement présentées dans l'*AddressSpace* afin de permettre un accès direct à l'état d'une *Alarme* ou d'une *Condition*.

Les *Types* d'*Alarmes* et de *Conditions* selon OPC UA sont définis du 5.5 au 5.8. La Figure 8 décrit de manière informelle la hiérarchie de ces *Types*. Les sous-types du *LimitAlarmType* et du *DiscreteAlarmType* ne sont pas représentés. La hiérarchie complète d'*AlarmConditionType* est disponible à la Figure 8.

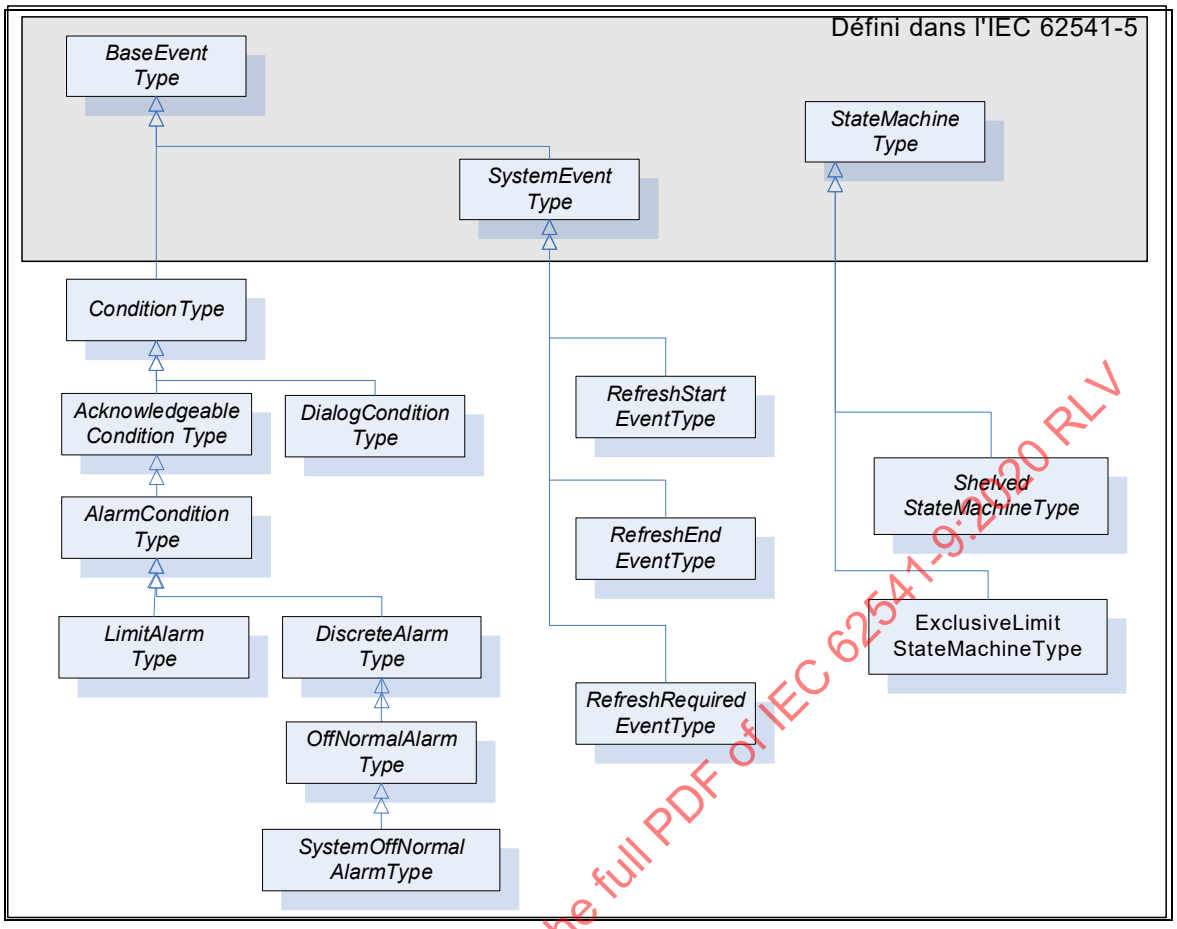


Figure 8 – Hiérarchie du ConditionType

L'Annexe C spécifie comment mapper le modèle décrit dans le présent document avec l'EEMUA.

L'Annexe D spécifie un mapping recommandé entre les serveurs OPC A&E et le modèle décrit dans le présent document.

5.2 Diagrammes d'états à deux états

La plupart des états définis dans le présent document sont simples: soit True, soit False. Le *TwoStateVariableType* est introduit spécialement pour ce cas d'utilisation. Des états plus complexes sont modélisés en utilisant un *StateMachineType* défini dans l'IEC 62541-5.

Le *TwoStateVariableType* est obtenu à partir du *StateVariableType* défini dans l'IEC 62541-5 et défini de façon formelle dans le Tableau 3.

Tableau 3 – Définition de TwoStateVariableType

Attribut	Valeur				
BrowseName	TwoStateVariableType				
DataType	LocalizedText				
ValueRank	-1 (-1 = Scalar)				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Sous-type du *StateVariableType* défini dans l'IEC 62541-5. Noter qu'une *Reference* à ce sous-type n'apparaît pas dans la définition du *StateVariableType*.					
HasProperty	Variable	Id	Booléen	PropertyType	Obligatoire
HasProperty	Variable	TransitionTime	UtcTime	PropertyType	Facultative
HasProperty	Variable	EffectiveTransitionTime	UtcTime	PropertyType	Facultative
HasProperty	Variable	TrueState	LocalizedText	PropertyType	Facultative
HasProperty	Variable	FalseState	LocalizedText	PropertyType	Facultative
HasTrueSubState	StateMachine ou TwoStateVariableType	<StateIdentifier>	Défini en 5.4.2		Facultative
HasFalseSubState	StateMachine ou TwoStateVariableType	<StateIdentifier>	Défini en 5.4.3		Facultative

L'*Attribut Valeur* d'une instance de *TwoStateVariableType* contient l'état courant sous la forme d'un nom lisible par l'homme. L'*EnabledState*, par exemple, pourrait contenir le nom "Enabled" lorsqu'il est True et "Disabled" lorsqu'il est False.

Id est hérité du *StateVariableType* et remplacé pour refléter le *DataType* (Booléen) exigé. La valeur doit être l'état courant, à savoir soit True, soit False.

TransitionTime spécifie l'heure de l'entrée dans l'état courant.

EffectiveTransitionTime spécifie l'heure de l'entrée dans l'état courant ou dans l'un de ses sous-états. Si, par exemple, une LevelAlarm est active et, alors qu'elle est active, commute plusieurs fois entre High et HighHigh, la *TransitionTime* reste à l'instant auquel l'*Alarme* est devenue active alors que la *EffectiveTransitionTime* change à chaque changement d'un sous-état.

La *Propriété* facultative *EffectiveDisplayName* issue du *StateVariableType* est utilisée si un état comporte des sous-états. Elle contient un nom lisible par l'homme pour l'état courant après prise en compte de l'état de tous les éventuels *SubStateMachines*. A titre d'exemple, l'*EffectiveDisplayName* de l'*EnabledState* pourrait contenir "Active/HighHigh" pour spécifier que la *Condition* est active et a dépassé la limite HighHigh.

D'autres *Propriétés* facultatives du *StateVariableType* n'ont pas de signification définie pour *TwoStateVariableType*.

TrueState et *FalseState* contiennent la chaîne localisée pour la valeur de *TwoStateVariableType* lorsque sa *Propriété Id* a respectivement la valeur True ou False. Sachant que les deux *Propriétés* fournissent des métadonnées pour le *Type*, les *Serveurs* peuvent ne pas offrir la possibilité de choisir ces *Propriétés* dans le filtre d'*Evénements* pour un MonitoredItem. Les *Clients* peuvent utiliser le *Service* Read pour récupérer des informations à partir du *ConditionType* spécifique.

La *Référence HasTrueSubState* est utilisée pour indiquer que l'état True comporte des sous-états.

La *Référence HasFalseSubState* est utilisée pour indiquer que l'état False comporte des sous-états.

5.3 ConditionVariable

Les divers éléments d'information d'une *Condition* ne sont pas jugés comme étant des états. Cependant, une modification de leur valeur est jugée importante et censée déclencher une *Notification d'Evénement*. Ces éléments d'information sont appelés *ConditionVariable*.

Les *ConditionVariables* sont représentées par un *ConditionVariableType* défini de façon formelle dans le Tableau 4.

Tableau 4 – Définition de ConditionVariableType

Attribut	**Valeur**				
BrowseName	ConditionVariableType				
DataType	BaseDataType				
ValueRank	-2 (-2 = Any)				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Sous-type du *BaseDataVariableType* défini dans l'IEC 62541-5.					
HasProperty	Variable	SourceTimestamp	UtcTime	PropertyType	Obligatoire

Le *SourceTimestamp* indique l'heure du dernier changement de la *Valeur* de cette *ConditionVariable*. Il doit s'agir de la même heure qui serait renvoyée par le *Service Read* au sein de la structure de *DataValue* pour l'*Attribut Valeur* de la *ConditionVariable*.

5.4 ReferenceTypes

5.4.1 Généralités

Le présent paragraphe définit les *ReferenceTypes* qu'il est nécessaire de connaître en plus de ceux déjà spécifiés dans le cadre de l'IEC 62541-3 et de l'IEC 62541-5. Cela comprend l'extension des diagrammes d'états de *TwoStateVariableType* avec des sous-états et l'ajout d'un groupement d'*Alarmes*.

Les *Références TwoStateVariableType* n'existent que lorsque des sous-états sont disponibles. Par exemple, si un diagramme *TwoStateVariableType* a un état False, alors tout sous-état dont la référence est l'état True n'est pas disponible. Lorsqu'un Evénement est généré alors que son état est False et lorsque les informations issues du sous-état de l'état True font partie des données à consigner, ces données sont alors consignées comme NULL. Avec cette approche, les *TwoStateVariableTypes* peuvent être étendus avec des diagrammes d'états subordonnés, d'une manière similaire au *StateMachineType* défini dans l'IEC 62541-5.

5.4.2 ReferenceType HasTrueSubState

Le *ReferenceType HasTrueSubState* est un *ReferenceType* concret qui peut être directement utilisé. Il s'agit d'un sous-type du *ReferenceType NonHierarchicalReferences*.

La sémantique indique que le sous-état (le *Nœud* cible) est un état subordonné au super état True. Si plusieurs états au sein d'une *Condition* sont des sous-états du même super état (à savoir, plusieurs *Références HasTrueSubState* existent pour le même super état), ils sont tous traités comme des sous-états indépendants. La représentation dans l'*AddressSpace* est spécifiée dans le Tableau 5.

Le *SourceNode* de la *Référence* doit être une instance d'un *TwoStateVariableType* et le *TargetNode* doit être soit une instance d'un *TwoStateVariableType*, soit une instance d'un sous-type d'un *StateMachineType*.

Il n'est pas nécessaire que la *Référence* HasTrueSubState d'un super état à un sous-état soit fournie, mais il est exigé que le sous-état fournisse la *Référence* inverse (*IsTrueSubStateOf*) à son super état.

Tableau 5 – ReferenceType HasTrueSubState

Attributs	Valeur		
BrowseName	HasTrueSubState		
InverseName	IsTrueSubStateOf		
Symmetric	False		
IsAbstract	False		
Références	**NodeClass**	**BrowseName**	**Commentaire**

5.4.3 ReferenceType HasFalseSubState

Le *ReferenceType HasFalseSubState* est un *ReferenceType* concret qui peut être directement utilisé. Il s'agit d'un sous-type du *ReferenceType NonHierarchicalReferences*.

La sémantique indique que le sous-état (le *Nœud* cible) est un état subordonné au super état False. Si plusieurs états au sein d'une *Condition* sont des sous-états du même super état (à savoir, plusieurs *Références HasFalseSubState* existent pour le même super état), ils sont tous traités comme des sous-états indépendants. La représentation dans l'*AddressSpace* est spécifiée dans le Tableau 6.

Le *SourceNode* de la *Référence* doit être une instance d'un *TwoStateVariableType* et le *TargetNode* doit être soit une instance d'un *TwoStateVariableType*, soit une instance d'un sous-type d'un *StateMachineType*.

Il n'est pas nécessaire que la *Référence* HasFalseSubState d'un super état à un sous-état soit fournie, mais il est exigé que le sous-état fournisse la *Référence* inverse (IsFalseSubStateOf) à son super état.

Tableau 6 – ReferenceType HasFalseSubState

Attributs	Valeur		
BrowseName	HasFalseSubState		
InverseName	IsFalseSubStateOf		
Symmetric	False		
IsAbstract	False		
Références	**NodeClass**	**BrowseName**	**Commentaire**

5.4.4 ReferenceType HasAlarmSuppressionGroup

Le *ReferenceType HasAlarmSuppressionGroup* est un *ReferenceType* concret qui peut être directement utilisé. Il s'agit d'un sous-type du *ReferenceType HasComponent*.

Ce *ReferenceType* lie un AlarmSuppressionGroup à une *Alarme*.

Le *SourceNode* de la *Référence* doit être une instance d'un *AlarmConditionType* ou d'un sous-type, et le *TargetNode* doit être une instance d'un *AlarmGroupType*.

Tableau 7 – ReferenceType HasAlarmSuppressionGroup

Attributs	Valeur		
BrowseName	HasAlarmSuppressionGroup		
InverseName	IsAlarmSuppressionGroupOf		
Symmetric	False		
IsAbstract	False		
Références	**NodeClass**	**BrowseName**	**Commentaire**

5.4.5 ReferenceType AlarmGroupMember

Le *ReferenceType AlarmGroupMember* est un *ReferenceType* concret qui peut être directement utilisé. Il s'agit d'un sous-type du *ReferenceType Organizes*.

Ce *ReferenceType* permet d'indiquer les instances d'*Alarmes* qui font partie d'un *Groupe d'Alarmes*.

Le *SourceNode* de la *Référence* doit être une instance d'un *AlarmGroupType* ou d'un sous-type de celui-ci et le *TargetNode* doit être une instance d'un *AlarmConditionType* ou d'un sous-type de celui-ci.

Tableau 8 – ReferenceType AlarmGroupMember

Attributs	Valeur		
BrowseName	AlarmGroupMember		
InverseName	MemberOfAlarmGroup		
Symmetric	False		
IsAbstract	False		
Références	**NodeClass**	**BrowseName**	**Commentaire**

5.5 Modèle de Condition

5.5.1 Généralités

Le modèle de *Condition* étend le modèle d'*Evénement* en définissant le *ConditionType*. Le *ConditionType* introduit le concept d'états qui le différencie du modèle d'*Evénement* de base. Contrairement aux *BaseEventType*, les *Conditions* ne sont pas transitoires. Le *ConditionType* est davantage étendu en Dialogue et *AcknowledgeableConditionType*, chacun de ceux-ci ayant leurs propres sous-types.

Le modèle de *Condition* est représenté à la Figure 9 et défini de façon formelle dans les tableaux suivants. Il est utile de préciser que cette figure, comme toutes les figures du présent document, est volontairement incomplète. Les figures représentent uniquement des informations fournies par les définitions formelles.

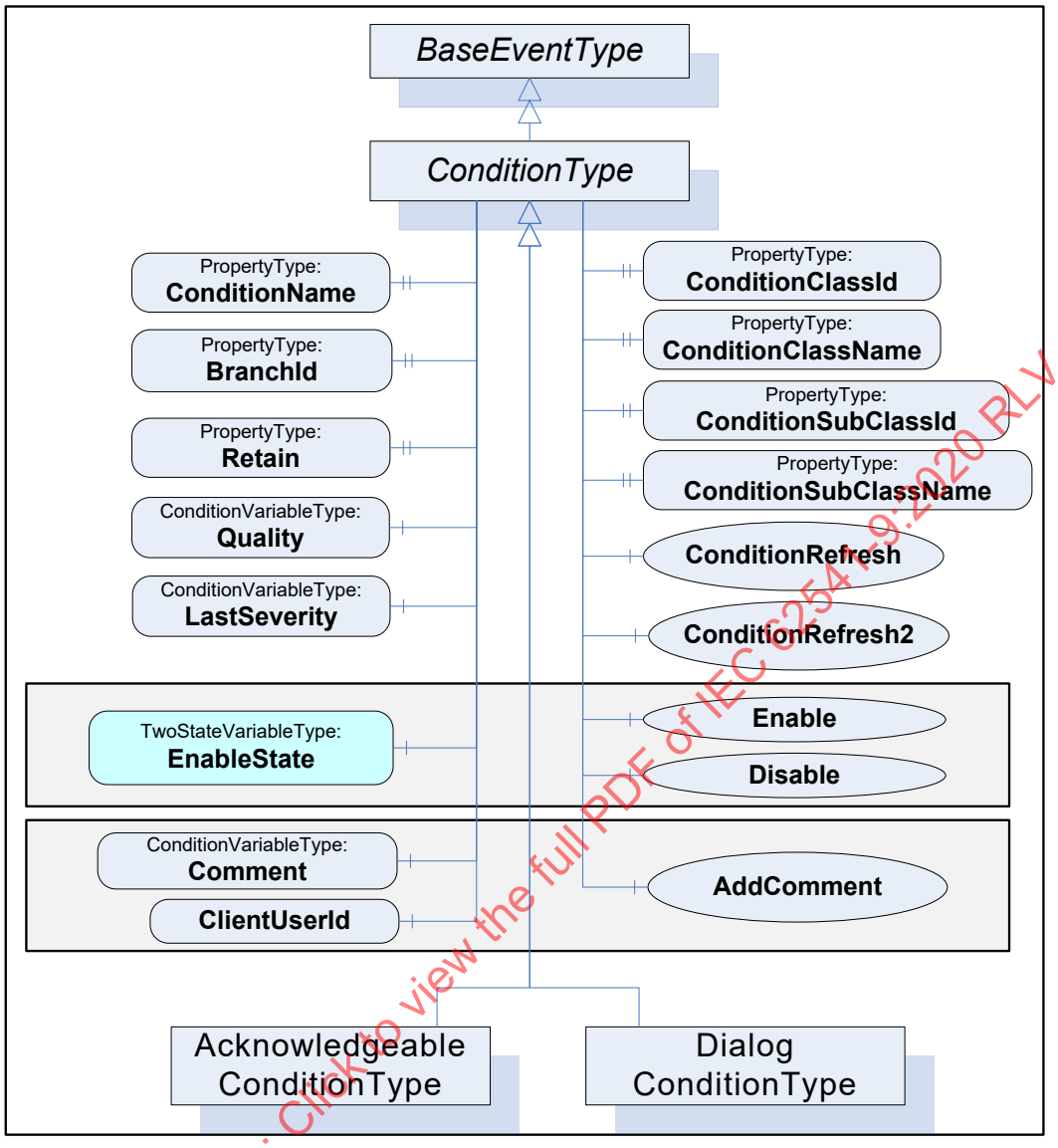


Figure 9 – Modèle de Condition

5.5.2 ConditionType

Le *ConditionType* définit toutes les caractéristiques générales d'une *Condition*. Tous les autres *ConditionTypes* en sont issus. Il est défini de façon formelle dans le Tableau 9. L'état False de l'*EnabledState* ne doit pas être étendu avec un diagramme de sous-états.

Tableau 9 – Définition de ConditionType

Attribut	Valeur				
BrowseName	ConditionType				
IsAbstract	True				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Sous-type du *BaseEventType* défini dans l'IEC 62541-5					
HasSubtype	ObjectType	DialogConditionType	Défini en 5.6.2		
HasSubtype	ObjectType	AcknowledgeableConditionType	Défini en 5.7.2		
HasProperty	Variable	ConditionClassId	NodeId	PropertyType	Obligatoire
HasProperty	Variable	ConditionClassName	LocalizedText	PropertyType	Obligatoire
HasProperty	Variable	ConditionSubClassId	NodeId	PropertyType	Facultative
HasProperty	Variable	ConditionSubClassName	LocalizedText	PropertyType	Facultative
HasProperty	Variable	ConditionName	Chaîne	PropertyType	Obligatoire
HasProperty	Variable	BranchId	NodeId	PropertyType	Obligatoire
HasProperty	Variable	Retain	Booléen	PropertyType	Obligatoire
HasComponent	Variable	EnabledState	LocalizedText	TwoStateVariableType	Obligatoire
HasComponent	Variable	Quality	StatusCode	ConditionVariableType	Obligatoire
HasComponent	Variable	LastSeverity	UInt16	ConditionVariableType	Obligatoire
HasComponent	Variable	Commentaire	LocalizedText	ConditionVariableType	Obligatoire
HasProperty	Variable	ClientUserId	Chaîne	PropertyType	Obligatoire
HasComponent	Méthode	Disable	Défini en 5.5.4		Obligatoire
HasComponent	Méthode	Enable	Défini en 5.5.5		Obligatoire
HasComponent	Méthode	AddComment	Défini en 5.5.6		Obligatoire
HasComponent	Méthode	ConditionRefresh	Défini en 5.5.7		Aucune
HasComponent	Méthode	ConditionRefresh2	Défini en 5.5.8		Aucune

Le *ConditionType* hérite de toutes les *Propriétés* du *BaseEventType*. Leur sémantique est définie dans l'IEC 62541-5. La *Propriété SourceNode* identifie la *ConditionSource*. Voir 5.12 pour plus de détails. Si la *ConditionSource* n'est pas un *Nœud* dans l'*AddressSpace*, le *NodeId* est défini sur NULL. Le *SourceNode* est le *Nœud* auquel la *Condition* est associée; il peut être identique à l'*InputNode* pour une *Alarme*, mais peut être un nœud distinct. Par exemple un moteur, qui est une *Variable* dont la *Valeur* est r/min (RPM), peut être la *ConditionSource* pour les *Conditions* associées au moteur, tout comme un capteur de température également associé au moteur. Dans le premier cas, l'*InputNode* de l'*Alarme* RPM High est la valeur de r/min du moteur, tandis que dans le dernier cas, l'*InputNode* de l'Alarme High serait la valeur du capteur de température associé au moteur.

Le *ConditionClassId* spécifie le domaine dans lequel cette *Condition* est utilisée. Il s'agit du *NodeId* du *BaseConditionClassType* correspondant. Voir 5.9 pour la définition de la *ConditionClass* et un jeu de *ConditionClasses* définies dans le présent document. Lorsqu'ils utilisent cette *Propriété* à des fins de filtrage, les *Clients* doivent spécifier tous les *NodeIds* des sous-types individuels de *BaseConditionClassType*. L'opérateur *OfType* ne peut pas être appliqué. Le *BaseConditionClassType* est utilisé comme une classe chaque fois qu'une *Condition* ne peut pas être affectée à une classe plus concrète.

Le *ConditionClassName* fournit le nom d'affichage du sous-type de *BaseConditionClassType*.

Le *ConditionSubClassId* spécifie la ou les classes supplémentaires qui s'appliquent à la *Condition*. Il s'agit du *NodeId* du *BaseConditionClassType* correspondant. Voir 5.9.6 pour la définition de la *ConditionClass* et un jeu de *ConditionClasses* définies dans le présent document. Lorsqu'ils utilisent cette *Propriété* à des fins de filtrage, les *Clients* doivent spécifier tous les *NodeIds* des sous-types individuels de *BaseConditionClassType*. L'opérateur *OfType* ne peut pas être appliqué. Le *Client* spécifie NULL dans le filtre, pour renvoyer les *Conditions* pour lesquelles aucune sous-classe ne s'applique. Lors du renvoi des *Conditions*, si ce champ facultatif n'est pas disponible dans une *Condition*, une valeur NULL doit être renvoyée pour le champ.

Le *ConditionSubClassName* fournit le ou les noms d'affichage des *ConditionClassTypes* répertoriés dans le *ConditionSubClassId*.

Le *ConditionName* identifie l'instance de *Condition* qui est à l'origine de l'*Evénement*. Il peut être utilisé avec le *SourceName* dans un affichage utilisateur pour distinguer les différentes instances de *Condition*. Si une *ConditionSource* n'a qu'une seule instance d'un *ConditionType* et que le *Serveur* n'a pas de nom d'instance, le *Serveur* doit fournir le nom de navigation du *ConditionType*.

Le *BranchId* est NULL pour toutes les *Notifications d'Evénements* qui se rapportent à l'état courant de l'instance de *Condition*. Si le *BranchId* n'est pas NULL, il identifie un état antérieur de cette instance de *Condition* qui nécessite encore l'attention d'un *Opérateur*. Si la *ConditionBranch* courante est transformée en une *ConditionBranch* antérieure, il est nécessaire que le Serveur affecte un *BranchId* non NUL. Un *Evénement* initial pour la branche est généré avec les valeurs de la *ConditionBranch* et du nouveau *BranchId*. La *ConditionBranch* peut être mise à jour plusieurs fois avant que cela ne soit plus nécessaire. Lorsque la *ConditionBranch* n'exige plus d'entrée de l'*Opérateur*, *Retain* est défini sur False pour l'*Evénement* final. Le bit "Retain" sur l'Evénement courant est True tant que les ConditionBranches exigent une entrée de l'Opérateur. Voir 4.4 pour plus d'informations sur la nécessité de créer et de maintenir des *ConditionBranches* antérieures, et l'Article B.1 pour un exemple utilisant des branches. Le *DataType BranchId* est le *NodeId*, bien que le *Serveur* ne soit pas tenu d'avoir des *ConditionBranches* dans l'*AddressSpace*. L'utilisation d'un *NodeId* permet au *Serveur* d'utiliser de simples identificateurs numériques, chaînes ou matrices d'octets.

La définition de *Retain* sur True permet à une *Condition* (ou une *ConditionBranch*) d'être dans un état intéressant pour un *Client* souhaitant synchroniser son état avec l'état du *Serveur*. La logique permettant de déterminer la manière dont le fanion est réglé est spécifique au *Serveur*. Habituellement, le fanion *Retain* de toutes les *Alarmes Actives* est défini; cependant, il est également possible que le fanion *Retain* des *Alarmes* inactives soit défini sur TRUE.

En traitement normal, lorsqu'un *Client* reçoit un *Evénement* dont le fanion *Retain* est défini sur False, il convient que le *Client* voie cela comme une *ConditionBranch* qui n'a plus d'intérêt; dans le cas d'un "affichage d'*Alarmes* courantes", la *ConditionBranch* serait retirée de l'affichage.

EnabledState indique si, oui ou non, la *Condition* est activée. *EnabledState/Id* est True si elle est activée, False autrement. *EnabledState/TransitionTime* définit le moment où l'*EnabledState* a changé pour la dernière fois. Les noms d'états recommandés sont décrits dans l'Annexe A.

L'*EnabledState* d'une *Condition* effectue la création de *Notifications d'Evénements* et, à ce titre, se traduit par le comportement spécifique suivant:

- lorsque l'instance de *Condition* entre dans l'état *Disabled*, la *Propriété Retain* de cette *Condition* doit être définie sur False par le *Serveur* afin d'indiquer au *Client* que l'instance de *Condition* ne présente pas actuellement d'intérêt pour les *Clients*. Cela comprend toutes les *ConditionBranches* si des branches existent;

- lorsque l'instance de *Condition* entre dans l'état activé, la *Condition* doit être évaluée et toutes ses *Propriétés* mises à jour pour refléter les valeurs actuelles. Si cette évaluation fait passer la *Propriété Retain* à True pour une *ConditionBranch*, une *Notification d'Evénement* doit être générée pour la *ConditionBranch* en question;
- le *Serveur* peut choisir de continuer à effectuer des essais pour une instance de *Condition* pendant qu'elle est *Désactivée*. Cependant, aucune *Notification d'Evénement* n'est générée pendant que l'instance de *Condition* est désactivée;
- pour n'importe quelle *Condition* qui existe dans l'*AddressSpace*, les *Attributs* et les *Variables* suivantes continuent à être valides, même dans l'état *Disabled*: EventId, *Event* Type, Source *Node*, Source Name, Time et EnabledState. D'autres *Propriétés* peuvent ne plus fournir de valeurs valides courantes. Toutes les *Variables* qui ne reçoivent plus de valeurs doivent renvoyer le statut Bad_ConditionDisabled. Il convient que l'Evénement qui signale l'état Disabled signale les *Propriétés* comme étant NULL ou ayant le statut Bad_ConditionDisabled.

Dans l'état Enabled, les changements apportés aux composants suivants doivent entraîner une *Notification d'Evénement* de *ConditionType*:

- Quality (qualité);
- Severity (sévérité, héritée de *BaseEventType*);
- Comment (commentaire).

Cette liste peut ne pas être exhaustive. Les sous-types peuvent définir des *Variables* supplémentaires qui déclenchent des *Notifications d'Evénements*. En général, les changements apportés aux *Variables* des types *TwoStateVariableType* ou *ConditionVariableType* déclenchent des *Notifications d'Evénements*.

Quality révèle le statut des valeurs de processus ou autres ressources sur lesquelles cette instance de *Condition* est fondée. Si, par exemple, une valeur de processus est "Uncertain", la *Condition* "LevelAlarm" associée est également douteuse. Les valeurs de *Quality* peuvent être tous les *StatusCodes* OPC définis dans l'IEC 62541-8 ainsi que *Good*, *Uncertain* et *Bad*, définis dans l'IEC 62541-4. Ces *StatusCodes* sont semblables à la description de la qualité de données, mais légèrement plus génériques que celle des diverses spécifications relatives aux bus de terrain. Il est de la responsabilité du *Serveur* de de réaliser le mapping entre les informations de statut interne avec ces codes. Un *Serveur* qui ne prend en charge aucune information relative à la qualité doit renvoyer *Good*. Cette qualité peut également refléter le statut de communication associé au système sur lequel est fondée cette valeur ou ressource, et duquel cette *Alarme* a été reçue. Pour les erreurs de communication du système sous-jacent, notamment celles donnant lieu à l'indisponibilité de certains champs d'*Evénement*, la qualité doit être l'erreur Bad_NoCommunication.

Les *Evénements* sont uniquement générés pour les *Conditions* dont le champ *Retain* est défini sur True et pour la transition initiale du champ *Retain* de True à False.

LastSeverity fournit la sévérité antérieure de la *ConditionBranch*. Initialement, cette *Variable* contient une valeur nulle; elle renvoie une valeur uniquement après un changement de sévérité. La nouvelle sévérité est fournie dans la *Propriété Severity* qui est héritée du *BaseEventType*.

Comment contient le dernier commentaire fourni pour un état donné (*ConditionBranch*). Il peut avoir été fourni par une *Méthode AddComment*, une autre *Méthode* ou d'une tout autre manière. La valeur initiale de cette *Variable* est NULL, à moins qu'elle ne soit fournie d'une tout autre manière. Si une Méthode fournit comme option la possibilité de définir *Comment*, la valeur de cette *Variable* est alors réinitialisée à NULL si un commentaire facultatif n'est pas fourni.

Le *ClientUserId* est lié au champ *Comment* et contient l'identité de l'utilisateur qui a inséré le *Commentaire* le plus récent. La logique pour obtenir le *ClientUserId* est définie dans l'IEC 62541-5.

Le *NodeId* de l'instance de *Condition* est utilisé comme *ConditionId*. Il n'est pas explicitement modélisé comme un composant du *ConditionType*. Il peut toutefois faire l'objet d'une demande avec le *SimpleAttributeOperand* suivant (voir Tableau 10) dans la *SelectClause* de l'*EventFilter*.

Tableau 10 – SimpleAttributeOperand

Nom	Type	Description
SimpleAttributeOperand		
typeId	NodeId	*NodeId* du *Nœud de ConditionType*
browsePath[]	QualifiedName	vide
attributeId	IntegerId	Identificateur de l'*Attribut NodeId*

5.5.3 Instances de Condition et de branche

Les *Conditions* sont des *Objets* qui ont un état qui change au fil du temps. Chaque instance de *Condition* a un *ConditionId* qui l'identifie de manière unique au sein du *Serveur*.

Une instance de *Condition* peut être un *Objet* qui apparaît dans l'*Espace* d'*Adressage* du *Serveur*. Si tel est le cas, le *ConditionId* est le *NodeId* de l'*Objet*.

L'état d'une instance de *Condition* à un instant donné correspond aux valeurs établies pour les *Variables* qui appartiennent à l'instance de *Condition*. En cas de changement d'une ou plusieurs valeurs des *Variables*, le *Serveur* génère un *Evénement* avec un *EventId* unique.

Si un *Client* appelle le *Rafraîchissement*, le *Serveur* signale l'état courant d'une instance de *Condition* en envoyant de nouveau le dernier *Evénement* (à savoir, les mêmes valeurs d'*EventId* et de *Time*).

Une *ConditionBranch* est une copie de l'état de l'instance de *Condition* qui peut changer indépendamment de l'état courant de l'instance de *Condition*. Chaque *Branche* a un identificateur appelé *BranchId*, qui est unique parmi toutes les *Branches* actives pour une instance de *Condition*. De manière générale, les *Branches* ne sont pas visibles dans l'*Espace* d'*Adressage* et le présent document ne définit pas de moyen normalisé de les rendre visibles.

5.5.4 Méthode Disable

La *Méthode Disable* est utilisée pour faire passer une instance de *Condition* à l'état *Disabled*. Généralement, le *NodeId* de l'instance d'objet est transmis en tant qu'*ObjectId* au *Service* d'Appel. Cependant, certains *Serveurs* ne présentent pas d'instances de *Condition* dans l'*AddressSpace*. Par conséquent, tous les *Serveurs* doivent autoriser les *Clients* à appeler la *Méthode Disable* en spécifiant le *ConditionId* en tant qu'*ObjectId*. La *Méthode* ne peut pas être appelée avec un *ObjectId* du *Nœud* de *ConditionType*.

Signature

```
Disable();
```

Le Tableau 11 présente les codes de résultats de la Méthode (définis dans le *Service* d'Appel).

Tableau 11 – Codes de résultats de la Méthode Disable

Code de résultat	Description
Bad_ConditionAlreadyDisabled	Voir Tableau 101 pour la description de ce code de résultat.

Le Tableau 12 spécifie la représentation de l'*AddressSpace* pour la *Méthode Disable*.

Tableau 12 – Définition de l'AddressSpace pour la Méthode Disable

Attribut	Valeur				
BrowseName	Disable				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionEnableEventType	Défini en 5.10.2		

5.5.5 Méthode Enable

La *Méthode Enable* est utilisée pour faire passer une instance de *Condition* à l'état Enabled. Généralement, le *NodeId* de l'instance d'objet est transmis en tant qu'*ObjectId* au *Service* d'Appel. Cependant, certains *Serveurs* ne présentent pas d'instances de *Condition* dans l'*AddressSpace*. Par conséquent, tous les *Serveurs* doivent autoriser les *Clients* à appeler la *Méthode Enable* en spécifiant le *ConditionId* en tant qu'*ObjectId*. La *Méthode* ne peut pas être appelée avec un *ObjectId* du *Nœud* de *ConditionType*. Si l'instance de *Condition* n'est pas présentée, il peut alors être difficile pour un *Client* de déterminer le *ConditionId* d'une *Condition* désactivée.

Signature

```
Enable();
```

Le Tableau 13 présente les codes de résultats de la *Méthode* (définis dans le *Service* d'Appel).

Tableau 13 – Codes de résultats de la Méthode Enable

Code de résultat	Description
Bad_ConditionAlreadyEnabled	Voir Tableau 101 pour la description de ce code de résultat.

Le Tableau 14 spécifie la représentation de l'*AddressSpace* pour la *Méthode Enable*.

Tableau 14 – Définition de l'AddressSpace pour la Méthode Enable

Attribut	Valeur				
BrowseName	Enable				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionEnableEventType	Défini en 5.10.2		

5.5.6 Méthode AddComment

La *Méthode AddComment* permet d'associer un commentaire à un état spécifique d'une instance de *Condition*. Généralement, le *NodeId* de l'instance d'*Objet* est transmis en tant qu'*ObjectId* au *Service* d'*Appel*. Cependant, certains *Serveurs* ne présentent pas d'instances de *Condition* dans l'*AddressSpace*. Par conséquent, tous les *Serveurs* doivent également autoriser les *Clients* à appeler la *Méthode AddComment* en spécifiant le *ConditionId* en tant qu'*ObjectId*. La *Méthode* ne peut pas être appelée avec un *ObjectId* du *Nœud* de *ConditionType*.

Signature

```
AddComment(
    [in] ByteString EventId
    [in] LocalizedText Commentaire
    );
```

Les paramètres sont définis dans le Tableau 15.

Tableau 15 – Arguments de la Méthode AddComment

Argument	Description
EventId	EventId identifiant une *Notification d'Evénement* particulière où un état a été consigné pour une *Condition*.
Comment	Texte localisé à appliquer à la *Condition*.

Le Tableau 16 présente les codes de résultats de la *Méthode* (définis dans le Service d'Appel).

Tableau 16 – Codes de résultats de la Méthode AddComment

Code de résultat	Description
Bad_MethodInvalid	Le *MethodId* fourni ne correspond pas à l'*ObjectId* fourni. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.
Bad_EventIdUnknown	Voir Tableau 101 pour la description de ce code de résultat.
Bad_NodeIdInvalid	Utilisé pour indiquer que l'*ObjectId* spécifié n'est pas valide ou que la *Méthode* a été appelée sur le *Nœud* de ConditionType. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.

Commentaires

Des *Commentaires* sont ajoutés aux occurrences d'*Evénements* identifiées par un *EventId*. Les *EventIds* où l'*EventType* concerné n'est pas un ConditionType (ou un sous-type de celui-ci) et qui ne prennent donc pas en charge les *Commentaires* sont rejetés.

Un *ConditionEvent* où la *Variable Comment* contient ce texte, est envoyé pour l'état identifié. Si un commentaire est ajouté à un état antérieur (à savoir un état pour lequel le Serveur a créé une branche), le *BranchId* et toutes les valeurs de *Condition* pour cette branche sont consignés.

Le Tableau 17 spécifie la représentation de l'*AddressSpace* pour la *Méthode AddComment*.

Tableau 17 – Définition de l'AddressSpace pour la Méthode AddComment

Attribut	Valeur				
BrowseName	AddComment				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	*Variable*	InputArguments	Argument	PropertyType	Obligatoire
AlwaysGeneratesEvent	ObjectType	AuditConditionCommentEventType	Défini en 5.10.4		

5.5.7 Méthode ConditionRefresh

ConditionRefresh permet à un *Client* de demander un *Rafraîchissement* de toutes les instances de *Condition* qui sont actuellement dans un état intéressant (leur fanion *Retain* est défini). Cela inclut les états antérieurs d'une instance de *Condition* pour lesquels le *Serveur* maintient des *Branches*. Un *Client* invoque généralement cette *Méthode* lorsqu'il se connecte initialement à un *Serveur* et à la suite de toutes les situations, telles que les interruptions de communication, dans lesquelles il exige la resynchronisation au *Serveur*. Cette *Méthode* n'est disponible qu'avec le *ConditionType* ou ses sous-types. Pour invoquer cette *Méthode*, l'appel doit transmettre le *MethodId* notoire de la *Méthode* au *ConditionType* et l'*ObjectId* doit être l'*ObjectId* notoire de l'*Objet* de *ConditionType*.

Signature

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

Les paramètres sont définis dans le Tableau 18.

Tableau 18 – Paramètres de la Méthode ConditionRefresh

Argument	Description
SubscriptionId	Identificateur d'*Abonnement* valide pour l'*Abonnement* à rafraîchir. Le *Serveur* doit vérifier que le *SubscriptionId* fourni fait partie de la *Session* qui invoque la *Méthode*.

Le Tableau 19 présente les codes de résultats de la *Méthode* (définis dans le *Service* d'Appel).

Tableau 19 – Codes de résultats de la Méthode ConditionRefresh

Code de résultat	Description
Bad_SubscriptionIdInvalid	Voir l'IEC 62541-4 pour la description de ce code de résultat
Bad_RefreshInProgress	Voir Tableau 101 pour la description de ce code de résultat
Bad_UserAccessDenied	La *Méthode* n'a pas été appelée dans le contexte de la *Session* qui possède l'*Abonnement*. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.

Commentaires

Le 4.5 décrit plus précisément le concept, les cas d'utilisation et le flux d'informations.

L'argument d'entrée fournit un identificateur d'*Abonnement* indiquant l'*Abonnement Client* qui doit être rafraîchi. Si l'*Abonnement* est accepté, le *Serveur* réagit comme suit:

1) le *Serveur* émet un événement de *RefreshStartEventType* (défini en 5.11.2) marquant le début du *Rafraîchissement*. Une copie de l'instance du *RefreshStartEventType* est mise en file d'attente dans le flux d'*Evénements* pour chaque *MonitoredItem Notificateur* dans l'*Abonnement*. Chacune des copies de l'*Evénement* doit contenir le même *EventId*;
2) le *Serveur* émet des *Notifications d'Evénements* de n'importe quelles *Conditions Retenues* et *Branches Retenues* des *Conditions* qui satisfont aux critères du filtre de contenu des *Abonnements*. Noter que l'EventId de cette *Notification* rafraîchie doit être identique à celui de la *Notification* originale: les valeurs des autres *Propriétés* sont spécifiques au *Serveur*, car certains *Serveurs* peuvent être capables de réitérer les *Evénements* exacts avec toutes les *Propriétés*/*Variables* conservant les mêmes valeurs que celles envoyées à l'origine, mais d'autres *Serveurs* peuvent être seulement capables de régénérer

l'*Evénement*. L'*Evénement* régénéré peut contenir des valeurs de *Propriété/Variable* mises à jour. Par exemple, si les limites de l'*Alarme* associées à une *Variable* ont été modifiées après la génération de l'*Evénement* sans générer de modification de l'état de l'*Alarme*, la nouvelle limite peut être consignée. Dans un autre exemple, si HighLimit est 100 et la *Variable* 120: si la limite est modifiée pour 90, aucun nouvel *Evénement* n'est généré puisqu'aucune modification du *StateMachine* ne s'est produite, mais la limite sur un *Rafraîchissement* indiquerait 90 alors que l'*Evénement* original indiquait 100;

3) le *Serveur* peut intercaler de nouvelles *Notifications* d'*Evénements* qui n'ont pas été émises précédemment à l'attention du *Notificateur* avec celles envoyées dans le cadre de la demande de *Rafraîchissement*. Les *Clients* doivent vérifier toutes les *Notifications* d'*Evénements* pour détecter une *ConditionBranch* afin d'éviter d'écraser un nouvel état délivré en même temps qu'un état plus ancien par le processus de *Rafraîchissement*;

4) le *Serveur* émet une instance du *RefreshEndEventType* (défini en 5.11.3) pour signaler la fin du *Rafraîchissement*. Une copie de l'instance du *RefreshEndEventType* est mise en file d'attente dans le flux d'*Evénements* pour chaque *MonitoredItem Notificateur* dans l'*Abonnement*. Chaque copie des *Evénements* doit contenir le même *EventId*.

Il est important de noter que si plusieurs *Notifications* d'*Evénements* sont dans un *Abonnement*, toutes les *Notifications d'Evénements* sont traitées. Si un *Client* ne souhaite pas que tous les *MonitoredItems* soient rafraîchis, il convient alors que le *Client* place chaque *MonitoredItem* dans un *Abonnement* distinct ou appelle *ConditionRefresh2* si le *Serveur* le prend en charge.

Si plusieurs *Abonnements* doivent être rafraîchis, le traitement de matrice de *Service* d'appel normalisé peut alors être utilisé.

Comme mentionné ci-dessus, *ConditionRefresh* doit aussi émettre des *Notifications d'Evénements* pour les états antérieurs qui nécessitent encore de l'attention. Cela est particulièrement vrai pour les instances de *Condition* dans lesquelles il existe des états antérieurs pour lesquels il est encore nécessaire de donner un acquittement ou une confirmation.

Le Tableau 20 spécifie la représentation de l'*AddressSpace* pour la *Méthode ConditionRefresh*.

Tableau 20 – Définition de l'AddressSpace pour la Méthode ConditionRefresh

Attribut	Valeur				
BrowseName	ConditionRefresh				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	*Variable*	InputArguments	Argument	PropertyType	Obligatoire
AlwaysGeneratesEvent	ObjectType	RefreshStartEvent	Défini en 5.11.2		
AlwaysGeneratesEvent	ObjectType	RefreshEndEvent	Défini en 5.11.3		

5.5.8 Méthode ConditionRefresh2

ConditionRefresh2 permet à un *Client* de demander un *Rafraîchissement* de toutes les instances de *Condition* qui sont actuellement dans un état intéressant (leur fanion *Retain* est défini) et associées à l'élément surveillé donné. Cette méthode fonctionne à tous autres égards comme ConditionRefresh. Un *Client* invoque généralement cette *Méthode* lorsqu'il se connecte initialement à un *Serveur* et à la suite de toutes les situations, telles que les interruptions de communication, dans lesquelles un seul *MonitoredItem* doit être resynchronisé avec le *Serveur*. Cette *Méthode* n'est disponible qu'avec le *ConditionType* ou ses sous-types. Pour invoquer cette *Méthode*, l'appel doit transmettre le *MethodId* notoire de la *Méthode* au *ConditionType* et l'*ObjectId* doit être l'*ObjectId* notoire de l'*Objet* de *ConditionType*.

Cette *Méthode* est facultative et, en tant que telle, les *Clients* doivent être prêts à traiter des *Serveurs* qui ne fournissent pas la *Méthode*. Si la *Méthode* renvoie *Bad_MethodInvalid*, le *Client* doit revenir à *ConditionRefresh*.

Signature

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

Les paramètres sont définis dans le Tableau 21.

Tableau 21 – Paramètres de la Méthode ConditionRefresh2

Argument	Description
SubscriptionId	Identificateur de l'*Abonnement* qui contient le *MonitoredItem* à rafraîchir. Le *Serveur* doit vérifier que le *SubscriptionId* fourni fait partie de la *Session* qui invoque la *Méthode*.
MonitoredItemId	Identificateur du *MonitoredItem* à rafraîchir. Le *MonitoredItemId* doit être dans l'*Abonnement* fourni.

Le Tableau 22 présente les codes de résultats de la *Méthode* (définis dans le *Service* d'Appel).

Tableau 22 – Codes de résultats de la Méthode ConditionRefresh2

Code de résultat	Description
Bad_SubscriptionIdInvalid	Voir l'IEC 62541-4 pour la description de ce code de résultat
Bad_MonitoredItemIdInvalid	Voir l'IEC 62541-4 pour la description de ce code de résultat
Bad_RefreshInProgress	Voir Tableau 101 pour la description de ce code de résultat
Bad_UserAccessDenied	La *Méthode* n'a pas été appelée dans le contexte de la Session qui possède l'*Abonnement*. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.
Bad_MethodInvalid	Voir l'IEC 62541-4 pour la description de ce code de résultat

Commentaires

Le 4.5 décrit plus précisément le concept, les cas d'utilisation et le flux d'informations.

L'argument d'entrée fournit un identificateur d'*Abonnement* et un identificateur de *MonitoredItem* qui indique quel *MonitoredItem* de l'*Abonnement Client* choisi doit être rafraîchi. Si l'*Abonnement* et le *MonitoredItem* sont acceptés, le *Serveur* réagit comme suit:

1) le *Serveur* émet un *RefreshStartEvent* (défini en 5.11.2) marquant le début du *Rafraîchissement*. Le *RefreshStartEvent* est mis en file d'attente dans le flux d'*Evénements* pour le *MonitoredItem Notificateur* dans l'*Abonnement*.

2) le *Serveur* émet des *Notifications d'Evénements* de n'importe quelles *Conditions Retenues* et *Branches Retenues* des *Conditions* qui satisfont aux critères du filtre de contenu des *Abonnements*. Noter que l'EventId de cette *Notification* rafraîchie doit être identique à celui de la *Notification* originale: les valeurs des autres *Propriétés* sont spécifiques au *Serveur*, car certains *Serveurs* peuvent être capables de réitérer les *Evénements* exacts avec toutes les *Propriétés*/*Variables* conservant les mêmes valeurs que celles envoyées à l'origine, mais d'autres *Serveurs* peuvent être seulement capables de régénérer l'*Evénement*. L'*Evénement* régénéré peut contenir des valeurs de *Propriété/Variable* mises à jour. Par exemple, si les limites de l'*Alarme* associées à une *Variable* ont été modifiées après la génération de l'*Evénement* sans générer de modification de l'état de

l'*Alarme*, la nouvelle limite peut être consignée. Dans un autre exemple, si HighLimit est 100 et la *Variable* 120: si la limite est modifiée pour 90, aucun nouvel *Evénement* n'est généré puisqu'aucune modification du *StateMachine* ne s'est produite, mais la limite sur un *Rafraîchissement* indiquerait 90 alors que l'*Evénement* original indiquait 100.

3) le *Serveur* peut intercaler de nouvelles *Notifications* d'*Evénements* qui n'ont pas été émises précédemment à l'attention du notificateur avec celles envoyées dans le cadre de la demande de *Rafraîchissement*. Les *Clients* doivent vérifier toutes les *Notifications* d'*Evénements* pour détecter une *ConditionBranch* afin d'éviter d'écraser un nouvel état délivré en même temps qu'un état plus ancien par le processus de *Rafraîchissement*.

4) le *Serveur* émet un *RefreshEndEvent* (défini en 5.11.3) pour signaler la fin du *Rafraîchissement*; Le *RefreshEndEvent* est mis en file d'attente dans le flux d'*Evénements* pour le *MonitoredItem Notificateur* dans l'*Abonnement*.

Si plusieurs *MonitoredItems* ou *Abonnements* doivent être rafraîchis, le traitement de matrice de *Service* d'appel normalisé peut alors être utilisé.

Comme mentionné ci-dessus, *ConditionRefresh2* doit aussi émettre des *Notifications* d'*Evénements* pour les états antérieurs qui nécessitent encore de l'attention. Cela est particulièrement vrai pour les instances de *Condition* dans lesquelles il existe des états antérieurs pour lesquels il est encore nécessaire de donner un acquittement ou une confirmation.

Le Tableau 23 spécifie la représentation de l'*AddressSpace* pour la *Méthode ConditionRefresh2*.

Tableau 23 – Définition de l'AddressSpace pour la Méthode ConditionRefresh2

Attribut	**Valeur**				
BrowseName	ConditionRefresh2				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	*Variable*	InputArguments	Argument	PropertyType	Obligatoire
AlwaysGeneratesEvent	ObjectType	RefreshStartEvent	Défini en 5.11.2		
AlwaysGeneratesEvent	ObjectType	RefreshEndEvent	Défini en 5.11.3		

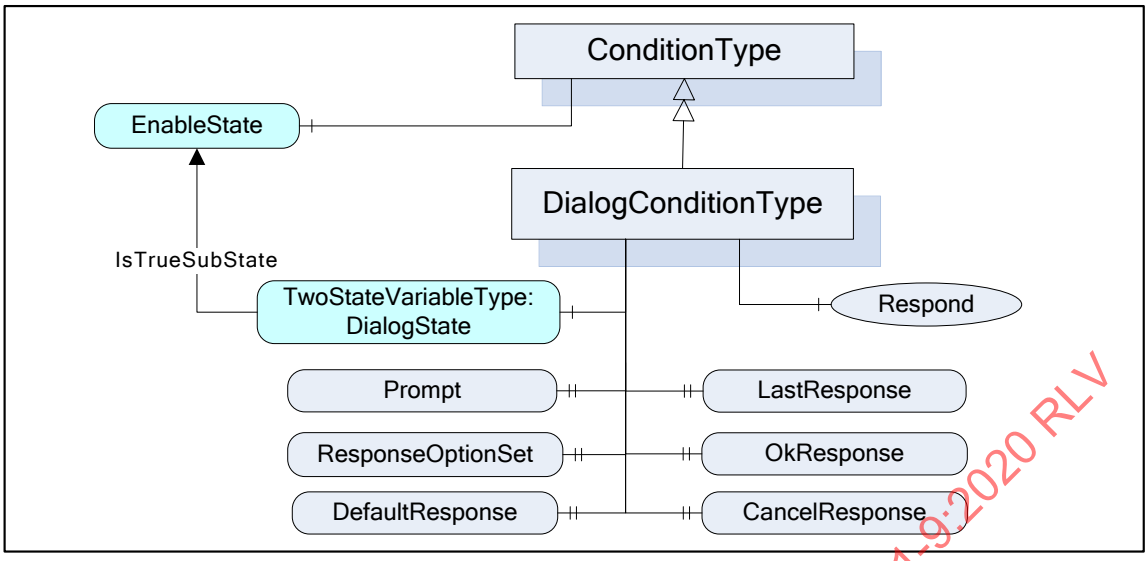
5.6 Modèle de Dialogue

5.6.1 Généralités

Le modèle de Dialogue est une extension du modèle de *Condition* utilisé par un *Serveur* pour demander des données d'utilisateur. Il fournit une fonctionnalité semblable aux dialogues de *Messages* normalisés rencontrés dans la plupart des systèmes d'exploitation. Le modèle peut être facilement personnalisé en fournissant des options de réponse spécifiques au *Serveur* dans le *ResponseOptionSet* et en ajoutant une fonctionnalité supplémentaire aux *Types* de *Conditions* résultants.

5.6.2 DialogConditionType

Le *DialogConditionType* permet de représenter des *Conditions* sous la forme de dialogues. Il est représenté à la Figure 10 et défini de façon formelle dans le Tableau 24.



IEC

Figure 10 – Vue d'ensemble du DialogConditionType

Tableau 24 – Définition de DialogConditionType

Attribut	**Valeur**				
BrowseName	DialogConditionType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Sous-type du ConditionType défini en 5.5.2.					
HasComponent	Variable	DialogState	LocalizedText	TwoStateVariableType	Obligatoire
HasProperty	Variable	Prompt	LocalizedText	PropertyType	Obligatoire
HasProperty	Variable	ResponseOptionSet	LocalizedText [ ]	PropertyType	Obligatoire
HasProperty	Variable	DefaultResponse	Int32	PropertyType	Obligatoire
HasProperty	Variable	LastResponse	Int32	PropertyType	Obligatoire
HasProperty	Variable	OkResponse	Int32	PropertyType	Obligatoire
HasProperty	Variable	CancelResponse	Int32	PropertyType	Obligatoire
HasComponent	Méthode	Respond	Défini en 5.6.3		Obligatoire

Le *DialogConditionType* hérite de toutes les *Propriétés* du *ConditionType*.

Lorsqu'il est défini sur True, le *DialogState/Id* indique que le *Dialogue* est actif et attend une réponse. Les noms d'états recommandés sont décrits dans l'Annexe A.

Prompt est une invite de dialogue à montrer à l'utilisateur.

ResponseOptionSet spécifie le jeu souhaité de réponses sous la forme d'une matrice de *LocalizedText*. L'indice dans cette matrice est utilisé pour les champs correspondants tels que *DefaultResponse*, *LastResponse* et *SelectedOption* dans la *Méthode Respond*. Les noms localisés recommandés pour les options communes sont décrits dans l'Annexe A.

Les combinaisons types d'options de réponse sont:

- OK;
- OK, Annuler;
- Oui, Non, Annuler;
- Abandonner, Réessayer, Ignorer;
- Réessayer, Annuler;
- Oui, Non.

DefaultResponse identifie l'option de réponse qu'il convient de montrer comme valeur par défaut à l'utilisateur. Il s'agit de l'indice dans la matrice *ResponseOptionSet*. Si aucune option de réponse n'est la valeur par défaut, la valeur de la *Propriété* est "−1".

LastResponse contient la dernière réponse fournie par un *Client* dans la *Méthode Respond*. Si aucune réponse antérieure n'existe, la valeur de la *Propriété* est "−1".

OkResponse fournit l'indice de l'option OK dans la matrice *ResponseOptionSet*. Ce choix est la réponse qui permet au système de poursuivre avec l'opération décrite par l'invite de commande. Cela permet au *Client* d'identifier l'option OK si une gestion spéciale est disponible pour cette option. Si aucune option OK n'est disponible, la valeur de cette *Propriété* est "−1".

CancelResponse fournit l'indice de réponse dans la matrice *ResponseOptionSet* qui fait passer le Dialogue à l'état inactif sans procéder à l'opération décrite par l'invite de commande. Cela permet au *Client* d'identifier l'option Annuler si une gestion spéciale est disponible pour cette option. Si aucune option Annuler n'est disponible, la valeur de cette *Propriété* est "−1".

5.6.3 Méthode Respond

La *Méthode Respond* permet de transmettre l'option de réponse choisie et de terminer le dialogue. Le *DialogState/Id* revient à False.

Signature

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

Les paramètres sont définis dans le Tableau 25.

Tableau 25 – Paramètres de la Méthode Respond

Argument	Description
SelectedResponse	Indice choisi dans la matrice *ResponseOptionSet*.

Le Tableau 26 présente les codes de résultats de la *Méthode* (définis dans le *Service* d'Appel).

Tableau 26 – Codes de résultats de la Méthode Respond

Code de résultat	Description
Bad_DialogNotActive	Voir Tableau 101 pour la description de ce code de résultat.
Bad_DialogResponseInvalid	Voir Tableau 101 pour la description de ce code de résultat.

Le Tableau 27 spécifie la représentation de l'*AddressSpace* pour la *Méthode Respond*.

Tableau 27 – Définition de l'AddressSpace pour la Méthode Respond

Attribut	Valeur				
BrowseName	Respond				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	*Variable*	InputArguments	Argument	PropertyType	Obligatoire
AlwaysGeneratesEvent	ObjectType	AuditConditionRespondEventType	Défini en 5.10.5		

5.7 Modèle de Condition acquittable

5.7.1 Généralités

Le modèle de *Condition* acquittable étend le modèle de *Condition*. Des états pour l'acquittement et la confirmation sont ajoutés au modèle de *Condition*.

Les *AcknowledgeableConditions* sont représentées par l'*AcknowledgeableConditionType* qui est un sous-type du *ConditionType*. Le modèle est défini de façon formelle du 5.7.2 au 5.7.4.

5.7.2 AcknowledgeableConditionType

L'*AcknowledgeableConditionType* étend le *ConditionType* en définissant des caractéristiques d'acquittement. Il s'agit d'un type abstrait. L'*AcknowledgeableConditionType* est représenté à la Figure 11 et défini de façon formelle dans le Tableau 28.

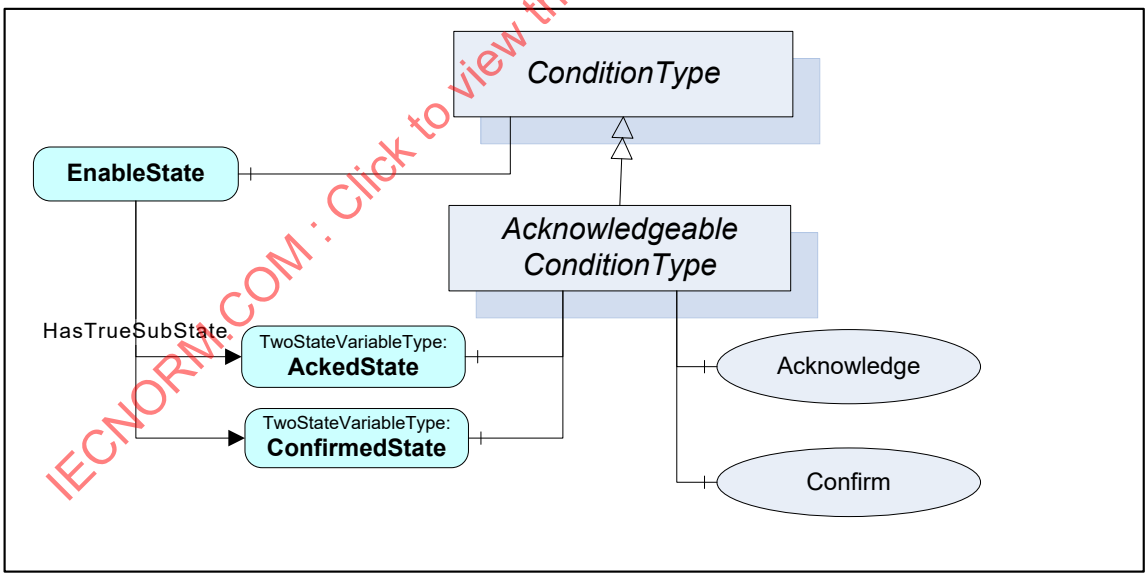


Figure 11 – Vue d'ensemble de l'AcknowledgeableConditionType

Tableau 28 – Définition d'AcknowledgeableConditionType

Attribut	Valeur				
BrowseName	AcknowledgeableConditionType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Sous-type du *ConditionType* défini en 5.5.2.					
HasSubtype	ObjectType	AlarmConditionType	Défini en 5.8.2		
HasComponent	Variable	AckedState	LocalizedText	TwoStateVariableType	Obligatoire
HasComponent	Variable	ConfirmedState	LocalizedText	TwoStateVariableType	Facultative
HasComponent	Méthode	Acquittement	Défini en 5.7.3		Obligatoire
HasComponent	Méthode	Confirm	Défini en 5.7.4		Facultative

L'*AcknowledgeableConditionType* hérite de toutes les *Propriétés* du *ConditionType*.

Lorsqu'il est False, *AckedState* indique que l'instance de *Condition* exige l'acquittement pour l'état de *Condition* consigné. Lorsque l'instance de *Condition* est acquittée, l'*AckedState* est défini sur True. *ConfirmedState* indique si, oui ou non, cela nécessite une confirmation. Les noms d'états recommandés sont décrits dans l'Annexe A. Les deux états sont des sous-états d'*EnabledState* True. Voir 4.3 pour plus d'informations sur les modèles d'acquittement et de confirmation. L'*EventId* utilisé dans la *Notification d'Evénement* est vu comme l'identificateur de cet état et doit être utilisé pour appeler les *Méthodes* pour l'acquittement ou la confirmation.

Un *Serveur* peut exiger que des états antérieurs soient acquittés. Si l'acquittement d'un état antérieur est encore ouvert et un nouvel état exige également un acquittement, le *Serveur* doit créer une branche de l'instance de *Condition* comme spécifié en 4.4. Il est prévu que les *Clients* gardent une trace de toutes les *ConditionBranches* où *AckedState*/Id est False pour permettre leur acquittement. Voir également 5.5.2 pour plus d'informations sur les *ConditionBranches* et les exemples de l'Article B.1. La gestion de l'*AckedState* et des branches s'applique aussi au *ConfirmedState*.

5.7.3 Méthode Acknowledge

La *Méthode Acknowledge* est utilisée pour acquitter une *Notification d'Evénement* pour un état de l'instance de *Condition* où *AckedState* est False. Généralement, le *NodeId* de l'instance d'objet est transmis en tant qu'*ObjectId* au *Service* d'*Appel*. Cependant, certains *Serveurs* ne présentent pas d'instances de *Condition* dans l'*AddressSpace*. Par conséquent, les *Serveurs* doivent autoriser les *Clients* à appeler la *Méthode Acknowledge* en spécifiant le *ConditionId* en tant qu'*ObjectId*. La *Méthode* ne peut pas être appelée avec un *ObjectId* du *Nœud* d'*AcknowledgeableConditionType*.

Signature

```
Acknowledge(
    [in] ByteString EventId
    [in] LocalizedText Commentaire
    );
```

Les paramètres sont définis dans le Tableau 29.

Tableau 29 – Paramètres de la Méthode Acknowledge

Argument	Description
EventId	EventId identifiant une *Notification d'Evénement* particulière. Seules les *Notifications d'Evénements* où AckedState/Id était False peuvent être acquittées.
Comment	Texte localisé à appliquer à la *Condition*.

Le Tableau 30 présente les codes de résultats de la *Méthode* (définis dans le Service d'Appel).

Tableau 30 – Codes de résultats de la Méthode Acknowledge

Code de résultat	Description
Bad_ConditionBranchAlreadyAcked	Voir Tableau 101 pour la description de ce code de résultat.
Bad_MethodInvalid	L'identificateur de la méthode ne se réfère pas à une méthode pour l'objet ou le ConditionId spécifié.
Bad_EventIdUnknown	Voir Tableau 101 pour la description de ce code de résultat.
Bad_NodeIdInvalid	Utilisé pour indiquer que l'*ObjectId* spécifié n'est pas valide ou que la *Méthode* a été appelée sur le *Nœud* de ConditionType. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.

Commentaires

Un *Serveur* est chargé de s'assurer que chaque *Evénement* a un *EventId* unique. Cela permet aux *Clients* d'identifier et d'acquitter une *Notification d'Evénement* particulière.

L'*EventId* identifie une *Notification d'Evénement* spécifique où un état à acquitter avait été consigné. L'acquittement et le commentaire facultatif sont appliqués à l'état identifié par l'*EventId*. Si le champ de commentaire est NULL (le paramètre de lieu et le texte sont vides), il est ignoré et les commentaires existants éventuels restent inchangés. Si le commentaire est à réinitialiser, un texte vide avec un paramètre de lieu doit être fourni.

Un *EventId* valide se traduit par une *Notification d'Evénement* où l'*AckedState*/Id est défini sur True et la *Propriété Comment* contient le texte de l'argument de commentaire facultatif. Si un état antérieur est acquitté, le *BranchId* et toutes les valeurs de *Condition* de cette branche sont consignés. Le Tableau 31 spécifie la représentation de l'*AddressSpace* pour la *Méthode Acknowledge*.

Tableau 31 – Définition de l'AddressSpace pour la Méthode Acknowledge

Attribut	Valeur				
BrowseName	Acquittement				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	*Variable*	InputArguments	Argument	PropertyType	Obligatoire
AlwaysGenerates Event	ObjectType	AuditConditionAcknowledge EventType	Défini en 5.10.5		

5.7.4 Méthode Confirm

La *Méthode Confirm* est utilisée pour confirmer une *Notification d'Evénement* pour un état de l'instance de *Condition* où le *ConfirmedState* est False. Généralement, le *NodeId* de l'instance d'objet est transmis en tant qu'*ObjectId* au *Service* d'*Appel*. Cependant, certains *Serveurs* ne présentent pas d'instances de *Condition* dans l'*AddressSpace*. Par conséquent, les *Serveurs* doivent autoriser les *Clients* à appeler la *Méthode Confirm* en spécifiant le *ConditionId* en tant qu'*ObjectId*. La *Méthode* ne peut pas être appelée avec un *ObjectId* du *Nœud d'AcknowledgeableConditionType*.

Signature

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

Les paramètres sont définis dans le Tableau 32.

Tableau 32 – Paramètres de la Méthode Confirm

Argument	Description
EventId	*EventId* identifiant une *Notification d'Evénement* particulière. Seules les *Notifications d'Evénement* dont la propriété Id du *ConfirmedState* est False peuvent être confirmées.
Comment	Texte localisé à appliquer aux *Conditions*.

Le Tableau 33 présente les codes de résultats de la *Méthode* (définis dans le Service d'Appel).

Tableau 33 – Codes de résultats de la Méthode Confirm

Code de résultat	Description
Bad_ConditionBranchAlreadyConfirmed	Voir Tableau 101 pour la description de ce code de résultat.
Bad_MethodInvalid	L'identificateur de la méthode ne se réfère pas à une méthode pour l'objet ou le *ConditionId* spécifié. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.
Bad_EventIdUnknown	Voir Tableau 101 pour la description de ce code de résultat.
Bad_NodeIdUnknown	Utilisé pour indiquer que l'*ObjectId* spécifié n'est pas valide ou que la *Méthode* a été appelée sur le *Nœud* de ConditionType. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.

Commentaires

Un *Serveur* est chargé de s'assurer que chaque *Evénement* a un *EventId* unique. Cela permet aux *Clients* d'identifier et de confirmer une *Notification d'Evénement* particulière.

L'*EventId* identifie une *Notification d'Evénement* spécifique où un état à confirmer avait été consigné. Un *Commentaire* qui est appliqué à l'état identifié par l'*EventId* peut être fourni.

Un *EventId* valide se traduit par une *Notification d'Evénement* où le *ConfirmedState*/Id est défini sur True, et la *Propriété Comment* contient le texte de l'argument de commentaire facultatif. Si un état antérieur est confirmé, le *BranchId* et toutes les valeurs de *Condition* pour cette branche sont consignés. Un *Client* peut confirmer uniquement les événements qui ont un *ConfirmedState*/Id défini sur False. La logique de définition du *ConfirmedState*/Id sur False est spécifique au *Serveur* et peut même être spécifique à l'événement ou à la condition.

Le Tableau 34 spécifie la représentation de l'*AddressSpace* pour la *Méthode Confirm*.

Tableau 34 – Définition de l'AddressSpace pour la Méthode Confirm

Attribut	Valeur				
BrowseName	Confirm				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	*Variable*	InputArguments	Argument	PropertyType	Obligatoire
AlwaysGeneratesEvent	ObjectType	AuditConditionConfirmEventType	Défini en 5.10.7		

5.8 Modèle d'Alarme

5.8.1 Généralités

La Figure 12 décrit de manière informelle l'AlarmConditionType, ses sous-types et sa position dans la hiérarchie des *Types d'Evénements*.

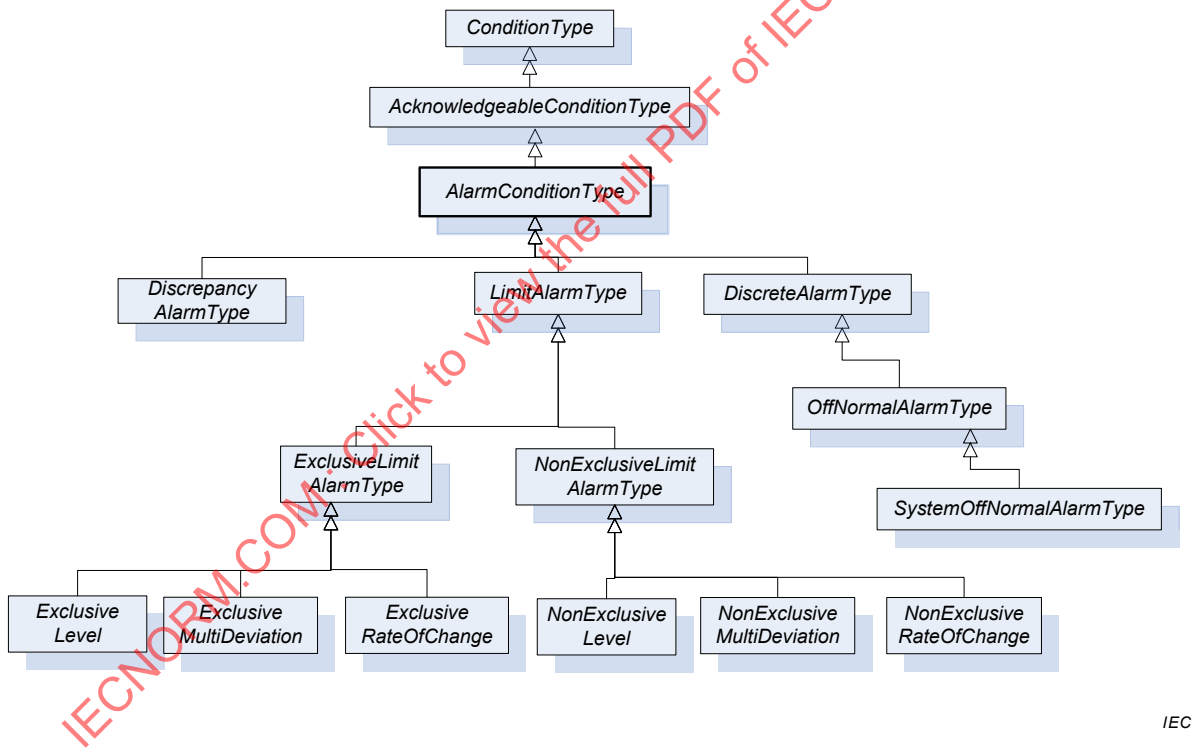


Figure 12 – Modèle de la hiérarchie d'AlarmConditionType

5.8.2 AlarmConditionType

L'*AlarmConditionType* est un type abstrait qui étend l'*AcknowledgeableConditionType* en introduisant un *ActiveState*, un *SuppressedState* et un *ShelvingState*. Il ajoute également la possibilité de définir un temps de retard, un temps de nouvelle alarme, des groupes d'*Alarmes* et des paramètres d'*Alarme* sonore. Le modèle d'*Alarme* est représenté à la Figure 13. Cette représentation est une définition volontairement incomplète. Il est défini de façon formelle dans le Tableau 35.

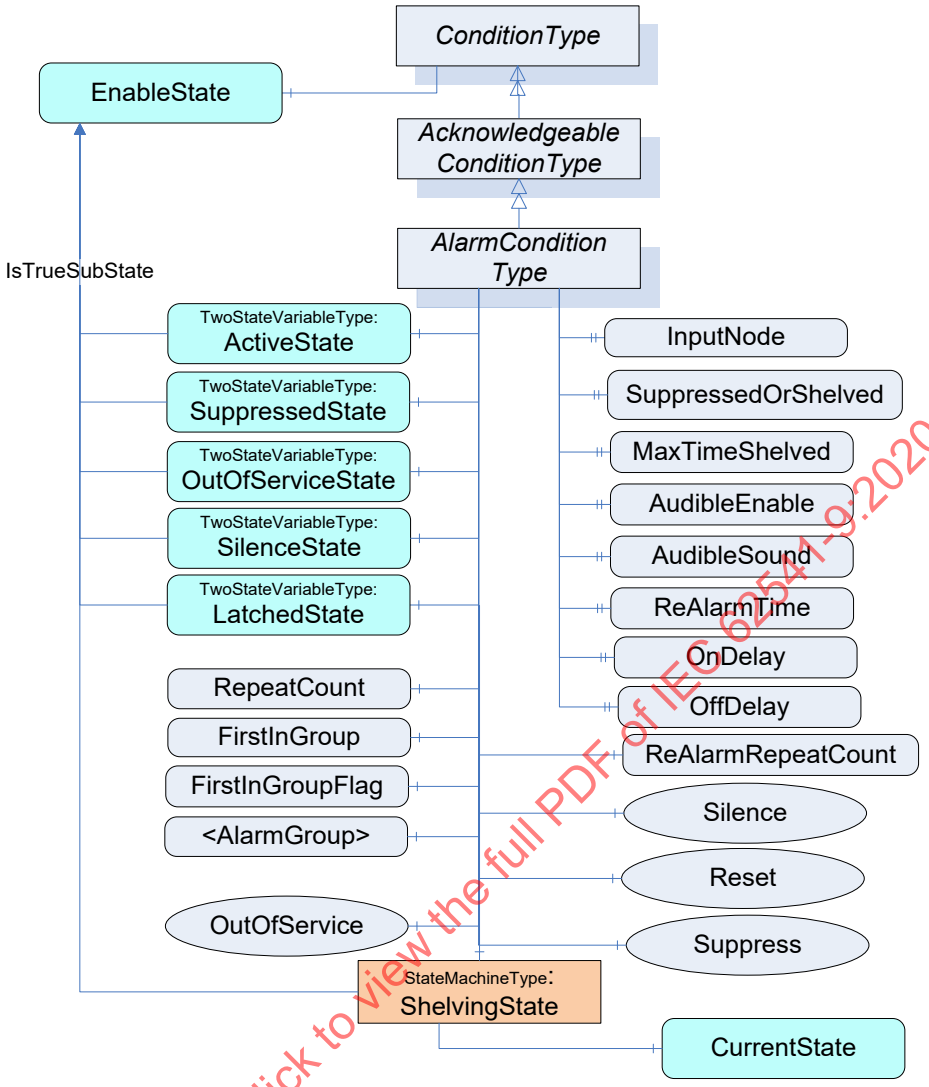


Figure 13 – Modèle d'Alarme

Tableau 35 – Définition d'AlarmConditionType

Attribut	Valeur				
BrowseName	AlarmConditionType				
IsAbstract	False				
Références	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Sous-type de l'AcknowledgeableConditionType défini en 5.7.2					
HasComponent	Variable	ActiveState	LocalizedText	TwoStateVariableType	Obligatoire
HasProperty	Variable	InputNode	NodeId	PropertyType	Obligatoire
HasComponent	Variable	SuppressedState	LocalizedText	TwoStateVariableType	Facultative
HasComponent	Variable	OutOfServiceState	LocalizedText	TwoStateVariableType	Facultative
HasComponent	Objet	ShelvingState		ShelvedStateMachineType	Facultative
HasProperty	Variable	SuppressedOrShelved	Booléen	PropertyType	Obligatoire
HasProperty	Variable	MaxTimeShelved	Durée	PropertyType	Facultative
HasProperty	Variable	AudibleEnabled	Booléen	PropertyType	Facultative
HasComponent	Variable	AudibleSound	AudioData Type	AudioVariableType	Facultative
HasComponent	Variable	SilenceState	LocalizedText	TwoStateVariableType	Facultative
HasProperty	Variable	OnDelay	Durée	PropertyType	Facultative
HasProperty	Variable	OffDelay	Durée	PropertyType	Facultative
HasComponent	Variable	FirstInGroupFlag	Booléen	BaseDataVariableType	Facultative
HasComponent	Objet	FirstInGroup		AlarmGroupType	Facultative
HasComponent	Objet	LatchedState	LocalizedText	TwoStateVariableType	Facultative
HasAlarmSuppressionGroup	Objet	<AlarmGroup>		AlarmGroupType	OptionalPlaceholder
HasProperty	Variable	ReAlarmTime	Durée	PropertyType	Facultative
HasComponent	Variable	ReAlarmRepeatCount	Int16	BaseDataVariableType	Facultative
HasComponent	Méthode	Silence	Défini en 5.8.5		Facultative
HasComponent	Méthode	Supprimer	Défini en 5.8.6		Facultative
HasComponent	Méthode	Unsuppress	Défini en 5.8.7		Facultative
HasComponent	Méthode	RemoveFromService	Défini en 5.8.8		Facultative
HasComponent	Méthode	PlaceInService	Défini en 5.8.9		Facultative
HasComponent	Méthode	Reset	Défini en 5.8.4		Facultative
HasSubtype	Objet	DiscreteAlarmType			
HasSubtype	Objet	LimitAlarmType			
HasSubtype	Objet	DiscrepancyAlarmType			

L'*AlarmConditionType* hérite de toutes les *Propriétés* de l'*AcknowledgeableConditionType*. Les états suivants sont des sous-états d'*EnabledState* True.

Lorsqu'il est défini sur True, l'*ActiveState/Id* indique que la situation représentée par la *Condition* est actuellement présente. Lorsqu'une instance de *Condition* est dans l'état inactif (l'*ActiveState/Id* défini sur False), cela représente une situation qui est revenue à un état normal. Les transitions de *Conditions* vers les états inactif et *Active* sont déclenchées par des actions spécifiques au *Serveur*. Les sous-types de l'*AlarmConditionType* spécifiés plus loin dans le présent document ont des modèles de sous-états qui définissent l'état *Active* de façon plus approfondie. Les noms d'états recommandés sont décrits dans l'Annexe A.

La *Propriété InputNode* fournit le *NodeId* de la *Variable* dont la *Valeur* est utilisée comme donnée d'entrée primaire dans le calcul de l'état d'*Alarme*. Si cette *Variable* n'est pas dans l'*AddressSpace*, un *NodeId* NULL doit être fourni. Dans certains systèmes, une *Alarme* peut être calculée sur la base de plusieurs *Valeurs* de *Variables*; il incombe au système de déterminer quel *NodeId* de *Variable* est utilisé.

L'association de *SuppressedState*, *OutOfServiceState* et *ShelvingState* permet de supprimer les *Alarmes* sur les systèmes d'affichage. Ces trois suppressions sont généralement utilisées par différents personnels ou systèmes d'une installation, c'est-à-dire des systèmes automatiques, du personnel de maintenance et des *Opérateurs*.

SuppressedState est utilisé en interne par un *Serveur* afin de supprimer automatiquement des *Alarmes* pour des raisons spécifiques au système. Par exemple, un système peut être configuré pour supprimer des *Alarmes* associées à des machines qui sont à l'arrêt. Par exemple, une *Alarme* de niveau bas pour un réservoir qui n'est pas en cours d'utilisation peut être supprimée. Les noms d'états recommandés sont décrits dans l'Annexe A.

OutOfServiceState permet au personnel de maintenance de supprimer des *Alarmes* dues à un problème de maintenance. Par exemple, si un instrument est mis hors service à des fins de maintenance ou est temporairement retiré pour être remplacé ou entretenu, l'état de l'élément est défini sur *OutOfServiceState*. Les noms d'états recommandés sont décrits dans l'Annexe A.

ShelvingState suggère si, oui ou non, une *Alarme* doit (temporairement) ne pas être affichée à l'attention de l'utilisateur. Cela est très souvent utilisé par les *Opérateurs* pour bloquer les *Alarmes* injustifiées. Le *ShelvingState* est défini en 5.8.10.

Lorsqu'une *Alarme* a l'un ou tous les états *SuppressedState, OutOfServiceState* ou *ShelvingState* définis sur True, la propriété *SuppressedOrShelved* doit être définie sur True et cette *Alarme* n'est alors généralement pas affichée par le *Client*. Les transitions d'états associées à l'*Alarme* ont effectivement lieu, mais elles ne sont généralement pas affichées par les *Clients* tant que l'*Alarme* reste dans l'un des états *SuppressedState, OutOfServiceState* ou *Shelved*.

La *Propriété* facultative *MaxTimeShelved* est utilisée pour attribuer la durée maximale pendant laquelle une *Condition* d'*Alarme* peut être suspendue. La valeur est exprimée sous la forme d'une durée. Les systèmes peuvent utiliser cette *Propriété* pour empêcher la *Suspension* permanente d'une *Alarme*. Si cette *Propriété* est présente, elle représente une limite supérieure pour la durée passée dans un appel de *Méthode TimedShelve*. Si une valeur supérieure à la valeur de cette *Propriété* est transmise à la *Méthode TimedShelve*, un code d'erreur *Bad_ShelvingTimeOutOfRange* est alors renvoyé lors de l'appel. Si cette *Propriété* est présente, elle est également en vigueur pour l'état *OneShotShelved*, ainsi une *Condition d'Alarme* passe à l'état *Unshelved* à partir de l'état *OneShotShelved* si la durée spécifiée dans cette *Propriété* expire à la suite d'une opération *OneShotShelve* sans changement des autres éléments éventuels associés à la *Condition*.

La *Propriété AudibleEnabled* facultative est un Booléen qui indique si l'état courant de cette *Alarme* inclut une *Alarme* sonore.

La *Propriété AudibleSound* facultative contient le fichier son qui doit être joué si une *Alarme* sonore doit être générée. Ce fichier est joué/généré tant que l'*Alarme* est active et non acquittée, sauf si le *StateMachine* silence est inclus, auquel cas il peut également être réduit au silence par ce *StateMachine*.

Le *SilenceState* permet de supprimer la génération d'*Alarmes* sonores. Généralement, il est utilisé lorsqu'un *Opérateur* réduit au silence toutes les *Alarmes* sur un écran, mais qu'il est nécessaire qu'il acquitte les *Alarmes* de manière individuelle. Réduire une *Alarme* au silence doit réduire l'*Alarme* au silence sur tous les systèmes (écrans) sur lesquels elle est signalée. Tous les *Clients* ne font pas appel à ce *StateMachine*, mais il permet à plusieurs *Clients* de synchroniser des états d'*Alarme* sonore. Acquitter une *Alarme* doit automatiquement la réduire au silence.

Les *Propriétés OnDelay* et *OffDelay* peuvent être utilisées pour éliminer les *Alarmes* injustifiées. *OnDelay* permet d'éviter les Alarmes inutiles lorsqu'un signal dépasse temporairement son point de consigne, empêchant ainsi l'*Alarme* d'être déclenchée avant que le signal reste à l'état d'*Alarme* en continu pour une période spécifiée (durée *OnDelay*). *OffDelay* permet de réduire les *Alarmes* intermittentes en verrouillant l'indication de l'*Alarme* pour une certaine durée après que la situation est revenue à la normale. C'est-à-dire que l'*Alarme* doit rester active pendant la durée *OffDelay* et ne doit pas se régénérer si elle redevient active pendant cette période. Si l'*Alarme* reste dans la zone inactive pendant le temps *OffDelay*, elle devient alors inactive.

La variable *FirstInGroupFlag* facultative est utilisée avec l'objet *FirstInGroup*. L'Objet FirstInGroup est une instance d'un AlarmGroupType qui regroupe plusieurs *Alarmes* associées. Le *FirstInGroupFlag* est défini sur l'instance d'*Alarme* qui était la première *Alarme* à se déclencher dans un *FirstInGroup*. En présence de cette variable, le *FirstInGroup* doit aussi être présent. Ces deux nœuds permettent à un système d'alarme de déterminer quelle *Alarme* de la liste a été le déclencheur. Ils sont communément utilisés dans les situations où les *Alarmes* sont corrélées et où généralement plusieurs *Alarmes* se produisent. En général, tous les capteurs de vibrations d'une turbine, par exemple, se déclenchent si l'un se déclenche, mais l'important, pour un *Opérateur*, est le premier capteur à s'être déclenché.

L'*Objet LatchedState*, s'il est présent, indique que le verrouillage de cette *Alarme* est pris en charge. Le bit "retain" de l'*Alarme* reste True jusqu'à ce qu'il ne soit plus actif, soit acquitté et réinitialisé. Si la *Méthode Reset* est appelée alors qu'elle est active, elle n'a pas d'effet sur l'*Alarme* et est ignorée, et la réponse à l'appel est une erreur Bad_InvalidState. L'*Objet* indique l'état courant, verrouillé ou non verrouillé. Les noms d'états recommandés sont décrits dans l'Annexe A. Si cet *Objet* est fourni, la *Méthode Reset* doit aussi être fournie.

Une instance d'*Alarme* peut contenir une ou plusieurs références HasAlarmSuppressionGroup à des instances d'*AlarmGroupType*. Chaque instance est un *AlarmSuppressionGroup*. Lorsqu'un *AlarmSuppressionGroup* devient actif, le *Serveur* doit définir le *SuppressedState* de l'*Alarme* sur True. Lorsque plus aucun *AlarmSuppressionGroup* référencé n'est actif, le Serveur doit alors définir le *SuppressedState* sur False. Un seul *AlarmSuppressionGroup* peut être attribué à plusieurs *Alarmes*. Les *AlarmSuppressionGroups* sont utilisés pour contrôler les *AlarmFloods* et pour mieux gérer les *Alarmes*.

ReAlarmTime, le cas échéant, définit un temps utilisé pour ramener une *Alarme* au sommet d'une liste d'*Alarmes*. Si une *Alarme* n'est pas revenue à la normale dans le temps donné (à partir du dernier moment où elle a été activée), le *Serveur* génère une nouvelle *Alarme* (comme s'il s'agissait de la première). Si elle a été réduite au silence, elle doit revenir à un état non silencieux, et si elle a été acquittée, elle doit revenir à un état non acquitté. La durée d'activité de l'*Alarme* est fonction du temps de nouvelle alarme.

ReAlarmRepeatCount, le cas échéant, comptabilise le nombre de fois qu'une *Alarme* est réactivée. Certains systèmes d'alarme intelligents utiliseront ce décompte pour élever le degré de priorité ou pour générer des indications supplémentaires ou différentes pour l'*Alarme* donnée. Le décompte est réinitialisé lorsqu'une *Alarme* revient à la normale.

La *Méthode Silence* peut être utilisée pour réduire une instance d'*Alarme* au silence. Elle est définie en 5.8.5.

La *Méthode Suppress* peut être utilisée pour supprimer une instance d'*Alarme*. Le plus souvent, la suppression d'une *Alarme* se produit par l'intermédiaire de la programmation avancée des alarmes, mais cette méthode permet de bénéficier d'un accès supplémentaire ou de supprimer une instance d'*Alarme* particulière. De plus amples détails sont fournis dans la définition en 5.8.6.

La *Méthode Unsuppress* peut être utilisée pour retirer une instance d'*Alarme* de l'état SuppressedState. De plus amples détails sont fournis dans la définition en 5.8.7.

La *Méthode PlaceInService* peut être utilisée pour retirer une instance d'*Alarme* de l'état *OutOfServiceState*. Elle est définie en 5.8.9.

La *Méthode RemoveFromService* peut être utilisée pour placer une instance d'*Alarme* dans l'état *OutOfServiceState*. Elle est définie en 5.8.8.

La *Méthode Réinitialiser* est utilisée pour effacer une *Alarme* verrouillée. Elle est définie en 5.8.4. Si cet *Objet* est fourni, l'*Objet LatchedState* doit aussi être fourni.

Plus de détails concernant le modèle d'*Alarme* et les divers états peuvent être consultés en 4.8 et à l'Annexe E.

5.8.3 AlarmGroupType

L'*AlarmGroupType* fournit une manière simple de regrouper les *Alarmes*. Ce regroupement peut être utilisé pour supprimer une *Alarme* ou pour identifier les *Alarmes* associées. L'usage réel de l'*AlarmGroupType* est spécifié lorsqu'il est utilisé.

L'*AlarmGroupType* est défini de façon formelle dans le Tableau 36.

Tableau 36 – Définition d'AlarmGroupType

Attribut	**Valeur**				
BrowseName	AlarmGroupType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Sous-type du FolderType défini dans l'IEC 62541-5.					
AlarmGroupMember	Objet	<AlarmConditionInstance>		AlarmConditionType	Optional Placeholder

Il convient de nommer l'instance d'un *AlarmGroupType* et de décrire l'objectif du groupe d'*Alarmes*.

L'instance d'*AlarmGroupType* contient un liste d'instances d'*AlarmConditionType* ou d'un sous-type d'*AlarmConditionType* référencé par des références d'*AlarmGroupMember*. Au moins une *Alarme* doit être présente dans une instance d'*AlarmGroupType*.

5.8.4 Méthode Reset

La *Méthode Reset* est utilisée pour réinitialiser une instance d'*Alarme* verrouillée. Elle n'est disponible que sur une instance d'un *AlarmConditionType* présentant le *LatchedState*. Généralement, le *NodeId* de l'instance d'*Objet* est transmis en tant qu'*ObjectId* au *Service* d'*Appel*. Cependant, certains *Serveurs* ne présentent pas d'instances de *Condition* dans l'*AddressSpace*. Par conséquent, les *Serveurs* doivent autoriser les *Clients* à appeler la *Méthode Reset* en spécifiant le *ConditionId* en tant qu'*ObjectId*. La *Méthode* ne peut pas être appelée avec un *ObjectId* du *Nœud* d'*AlarmConditionType*.

Signature

```
Reset();
```

Le Tableau 37 présente les codes de résultats de la *Méthode* (définis dans le Service d'*Appel*).

Tableau 37 – Codes de résultats de la Méthode Reset

Code de résultat	Description
Bad_MethodInvalid	Le *MethodId* fourni ne correspond pas à l'*ObjectId* fourni. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.
Bad_NodeIdInvalid	Utilisé pour indiquer que l'*ObjectId* spécifié n'est pas valide ou que la *Méthode* a été appelée sur le *Nœud* de *ConditionType*. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.
Bad_InvalidState	L'instance d'*Alarme* n'a pas été verrouillée, est encore active ou exige encore d'être acquittée. Pour qu'une instance d'*Alarme* soit réinitialisée, elle doit auparavant avoir été en état d'*Alarme*, être revenue à la normale et avoir été acquittée.

Le Tableau 38 spécifie la représentation de l'*AddressSpace* pour la *Méthode Reset*.

Tableau 38 – Définition de l'AddressSpace pour la Méthode Reset

Attribut	Valeur				
BrowseName	Reset				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionResetEventType	Défini en 5.10.11		

5.8.5 Méthode Silence

La *Méthode Silence* est utilisée pour réduire une instance d'*Alarme* spécifique au silence. Elle n'est disponible que sur une instance d'un *AlarmConditionType* présentant aussi le *SilenceState*. Généralement, le *NodeId* de l'instance d'*Objet* est transmis en tant qu'*ObjectId* au *Service* d'*Appel*. Cependant, certains *Serveurs* ne présentent pas d'instances de *Condition* dans l'*AddressSpace*. Par conséquent, les *Serveurs* doivent autoriser les *Clients* à appeler la *Méthode Silence* en spécifiant le *ConditionId* en tant qu'*ObjectId*. La *Méthode* ne peut pas être appelée avec un *ObjectId* du *Nœud* d'*AlarmConditionType*.

Signature

```
Silence();
```

Le Tableau 39 présente les codes de résultats de la *Méthode* (définis dans le service d'*Appel*).

Tableau 39 – Codes de résultats de la Méthode Silence

Code de résultat	Description
Bad_MethodInvalid	Le *MethodId* fourni ne correspond pas à l'*ObjectId* fourni. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.
Bad_NodeIdInvalid	Utilisé pour indiquer que l'*ObjectId* spécifié n'est pas valide ou que la *Méthode* a été appelée sur le *Nœud* de *ConditionType*. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.

Commentaires

Si l'instance n'est pas actuellement dans un état sonore, la commande est ignorée.

Le Tableau 40 spécifie la représentation de l'*AddressSpace* pour la *Méthode Silence*.

Tableau 40 – Définition de l'AddressSpace pour la Méthode Silence

Attribut	Valeur				
BrowseName	Silence				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionSilenceEventType	Défini en 5.10.10		

5.8.6 Méthode Suppress

La *Méthode Suppress* est utilisée pour supprimer une instance d'*Alarme* spécifique. Elle n'est disponible que sur une instance d'un *AlarmConditionType* présentant aussi le *SuppressedState*. Cette *Méthode* peut être utilisée pour modifier le *SuppressedState* d'une *Alarme* et écraser toute suppression provoquée par un *AlarmSuppressionGroup* associé. Cette *Méthode* fonctionne en parallèle avec toute suppression déclenchée par un *AlarmSupressionGroup*, au sens que si la *Méthode* est utilisée pour supprimer une *Alarme*, un *AlarmSuppressionGroup* peut effacer la suppression.

Généralement, le *NodeId* de l'instance d'objet est transmis en tant qu'*ObjectId* au *Service* d'*Appel*. Cependant, certains *Serveurs* ne présentent pas d'instances de *Condition* dans l'*AddressSpace*. Par conséquent, les *Serveurs* doivent autoriser les *Clients* à appeler la *Méthode Suppress* en spécifiant le *ConditionId* en tant qu'*ObjectId*. La *Méthode* ne peut pas être appelée avec un *ObjectId* du *Nœud* d'*AlarmConditionType*.

Signature

```
Suppress();
```

Le Tableau 41 présente les codes de résultats de la Méthode (définis dans le Service d'Appel).

Tableau 41 – Codes de résultats de la Méthode Suppress

Code de résultat	Description
Bad_MethodInvalid	Le *MethodId* fourni ne correspond pas à l'ObjectId fourni. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.
Bad_NodeIdInvalid	Utilisé pour indiquer que l'*ObjectId* spécifié n'est pas valide ou que la *Méthode* a été appelée sur le *Nœud* de *ConditionType*. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.

Commentaires

La *Méthode Suppress* s'applique à une instance d'*Alarme*, même si elle n'est pas active.

Le Tableau 42 spécifie la représentation de l'*AddressSpace* pour la *Méthode Suppress*.

Tableau 42 – Définition de l'AddressSpace pour la Méthode Suppress

Attribut	Valeur				
BrowseName	Supprimer				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
AlwaysGeneratesEvent	ObjectType	AuditConditionSuppressionEventType	Défini en 5.10.4		

5.8.7 Méthode Unsuppress

La *Méthode Unsuppress* est utilisée pour effacer le *SuppressedState* d'une instance d'*Alarme* spécifique. Elle n'est disponible que sur une instance d'un *AlarmConditionType* présentant aussi le *SuppressedState*. Cette *Méthode* peut être utilisée pour écraser toute suppression provoquée par un AlarmSuppressionGroup associé. Cette *Méthode* fonctionne en parallèle avec toute suppression déclenchée par un AlarmSuppressionGroup, au sens que si la *Méthode* est utilisée pour effacer le *SuppressedState* d'une *Alarme*, toute modification d'un AlarmSuppressionGroup peut supprimer à nouveau l'*Alarme*.

Généralement, le *NodeId* de l'ObjectInstance est transmis en tant qu'*ObjectId* au *Service* d'*Appel*. Cependant, certains *Serveurs* ne présentent pas d'instances de *Condition* dans l'*AddressSpace*. Par conséquent, les *Serveurs* doivent autoriser les *Clients* à appeler la *Méthode Unsuppress* en spécifiant le *ConditionId* en tant qu'*ObjectId*. La *Méthode* ne peut pas être appelée avec un *ObjectId* du *Nœud* d'*AlarmConditionType*.

Signature

```
Unsuppress();
```

Le Tableau 43 présente les codes de résultats de la Méthode (définis dans le Service d'Appel).

Tableau 43 – Codes de résultats de la Méthode Unsuppress

Code de résultat	Description
Bad_MethodInvalid	Le *MethodId* fourni ne correspond pas à l'*ObjectId* fourni. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.
Bad_NodeIdInvalid	Utilisé pour indiquer que l'*ObjectId* spécifié n'est pas valide ou que la *Méthode* a été appelée sur le *Nœud* de *ConditionType*. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.

Commentaires

La *Méthode Unsuppress* s'applique à une instance d'*Alarme*, même si elle n'est pas active.

Le Tableau 44 spécifie la représentation de l'*AddressSpace* pour la *Méthode Unsuppress*.

Tableau 44 – Définition de l'AddressSpace pour la Méthode Unsuppress

Attribut	Valeur				
BrowseName	Unsuppress				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionSuppressionEventType	Défini en 5.10.4		

5.8.8 Méthode RemoveFromService

La *Méthode RemoveFromService* est utilisée pour supprimer une instance d'*Alarme* spécifique. Elle n'est disponible que sur une instance d'un AlarmConditionType présentant aussi l'*OutOfServiceState*. Généralement, le *NodeId* de l'instance d'objet est transmis en tant qu'*ObjectId* au *Service* d'*Appel*. Cependant, certains *Serveurs* ne présentent pas d'instances de *Condition* dans l'*AddressSpace*. Par conséquent, les *Serveurs* doivent autoriser les *Clients* à appeler la *Méthode RemoveFromService* en spécifiant le *ConditionId* en tant qu'*ObjectId*. La *Méthode* peut ne pas être appelée avec un *ObjectId* du *Nœud* d'*AlarmConditionType*.

Signature

```
RemoveFromService ();
```

Le Tableau 45 présente les codes de résultats de la *Méthode* (définis dans le *Service* d'*Appel*).

Tableau 45 – Codes de résultats de la Méthode RemoveFromService

Code de résultat	Description
Bad_MethodInvalid	Le *MethodId* fourni ne correspond pas à l'*ObjectId* fourni. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.
Bad_NodeIdInvalid	Utilisé pour indiquer que l'*ObjectId* spécifié n'est pas valide ou que la *Méthode* a été appelée sur le *Nœud* de *ConditionType*. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.

Commentaires

Les instances qui ne présentent pas l'*Etat OutOfService* doivent rejeter les appels de *RemoveFromService*. La *Méthode RemoveFromService* s'applique à une instance d'*Alarme*, même si elle n'est pas actuellement à l'*Etat Active*.

Le Tableau 46 spécifie la représentation de l'*AddressSpace* pour la *Méthode RemoveFromService*.

Tableau 46 – Définition de l'AddressSpace pour la Méthode RemoveFromService

Attribut	Valeur				
BrowseName	RemoveFromService				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
AlwaysGeneratesEvent	ObjectType	AuditConditionOutOfServiceEventType	Défini en 5.10.12		

5.8.9 Méthode PlaceInService

La *Méthode PlaceInService* est utilisée pour définir l'OutOfServiceState d'une instance d'*Alarme* spécifique sur False. Elle n'est disponible que sur une instance d'un *AlarmConditionType* présentant aussi l'*OutOfServiceState*. Généralement, le *NodeId* de l'ObjectInstance est transmis en tant qu'*ObjectId* au *Service* d'*Appel*. Cependant, certains *Serveurs* ne présentent pas d'instances de *Condition* dans l'*AddressSpace*. Par conséquent, les *Serveurs* doivent autoriser les *Clients* à appeler la *Méthode PlaceInService* en spécifiant le *ConditionId* en tant qu'*ObjectId*. La *Méthode* ne peut pas être appelée avec un *ObjectId* du *Nœud* d'*AlarmConditionType*.

Signature

```
PlaceInService ();
```

Le Tableau 47 présente les codes de résultats de la *Méthode* (définis dans le *Service* d'*Appel*).

Tableau 47 – Codes de résultats de la Méthode PlaceInService

Code de résultat	Description
Bad_MethodInvalid	Le *MethodId* fourni ne correspond pas à l'*ObjectId* fourni. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.
Bad_NodeIdInvalid	Utilisé pour indiquer que l'*ObjectId* spécifié n'est pas valide ou que la *Méthode* a été appelée sur le *Nœud* de *ConditionType*. Voir l'IEC 62541-4 pour la description générale de ce code de résultat.

Commentaires

La *Méthode PlaceInService* s'applique à une instance d'*Alarme*, même si elle n'est pas actuellement à l'*Etat Active*.

Le Tableau 48 spécifie la représentation de l'*AddressSpace* pour la *Méthode PlaceInService*.

Tableau 48 – Définition de l'AddressSpace pour la Méthode PlaceInService

Attribut	Valeur				
BrowseName	PlaceInService				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
AlwaysGeneratesEvent	ObjectType	AuditConditionOutOfServiceEventType	Défini en 5.10.12		

5.8.10 ShelvedStateMachineType

5.8.10.1 Vue d'ensemble

Le *ShelvedStateMachineType* définit un diagramme de sous-états qui représente un modèle avancé de filtrage d'*Alarmes*. Ce modèle est représenté à la Figure 15.

Le modèle d'état prend en charge deux types de *Suspensions*: *OneShotShelving* et *TimedShelving*. Ils sont représentés à la Figure 14. La représentation comporte les transitions admises entre les divers sous-états. La *Suspension* est une activité déclenchée par l'*Opérateur*.

En *OneShotShelving*, un utilisateur demande qu'une *Alarme* soit suspendue pendant son état *Active*. Ce type de *Suspension* est habituellement utilisé lorsqu'une *Alarme* se produit en continu sur une limite (à savoir, une *Condition* oscille entre l'*Alarme* High et l'*Alarme* HighHigh, toujours dans l'état *Active*). La *Suspension* en une seule fois s'efface automatiquement lorsqu'une *Alarme* revient à un état inactif. Une autre utilisation pour ce type de *Suspension* concerne une zone d'installation qui est arrêtée, à savoir une *Alarme* fonctionnant longtemps telle qu'une *Alarme* de niveau bas pour un réservoir qui n'est pas en cours d'utilisation. Lorsque le réservoir recommence à fonctionner, l'état *Shelving* s'efface automatiquement.

En *TimedShelving*, un utilisateur spécifie qu'une *Alarme* soit suspendue pendant une durée donnée. Ce type de *Suspension* est très souvent utilisé pour bloquer les *Alarmes* injustifiées. Par exemple, une *Alarme* qui se produit plus de 10 fois en une minute peut être suspendue pendant quelques minutes.

Dans tous les états, la méthode *Unshelve* peut être appelée pour entraîner une transition vers l'état Unshelve; cela inclut le *Un-shelving* d'une *Alarme* qui est dans l'état *TimedShelve* avant que la durée n'expire et l'état *OneShotShelve* sans transition vers un état inactif.

Toutes les transitions, à l'exception de deux d'entre elles, sont causées par des appels de *Méthode* représentés à la Figure 14. La transition "Time Expired" est simplement une transition générée par le système qui se produit lorsque la valeur de temps définie comme partie du "Timed Shelved Call" a expiré. La transition "Any Transition Occurs" est également une transition générée par le système; cette transition est générée lorsque la *Condition* passe à un état inactif.