IEC 62541-9

Edition 2.0 2015-03

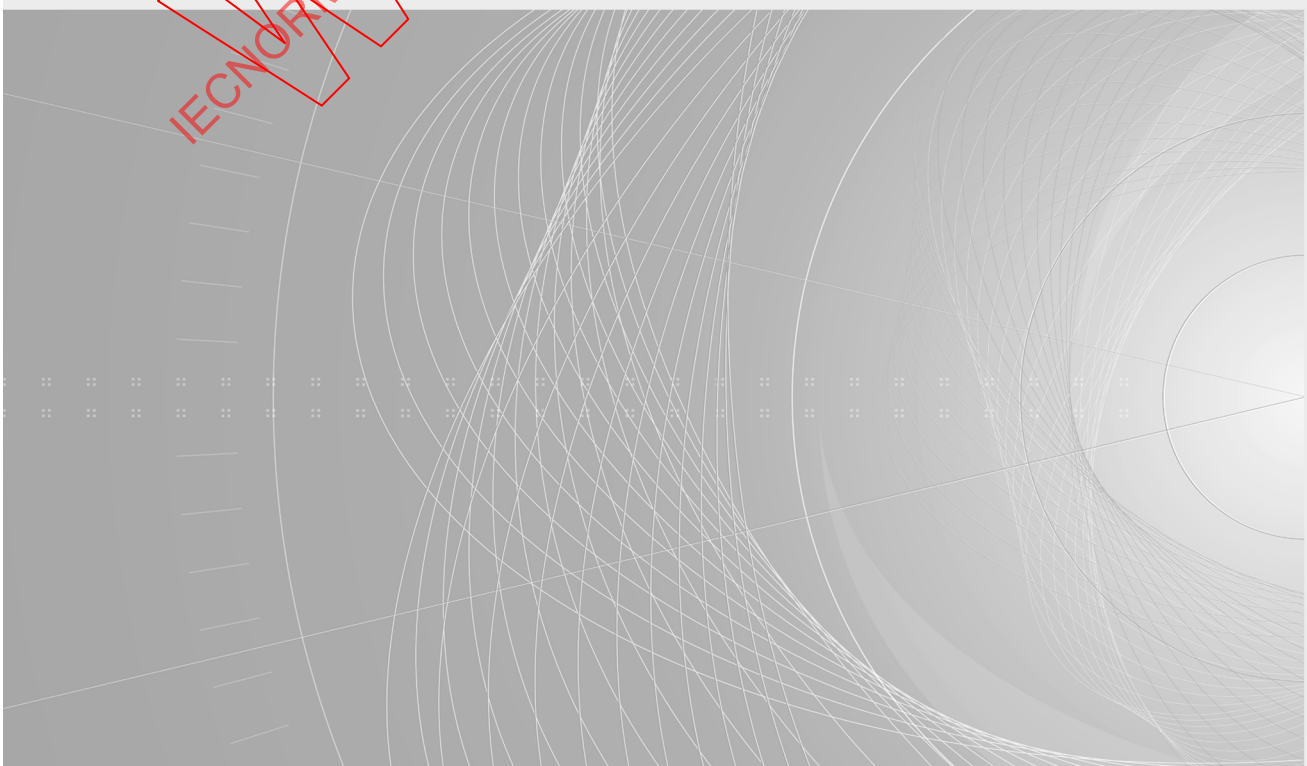
INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC unified architecture – Part 9: Alarms and conditions

Architecture unifiée OPC – Partie 9: Alarmes et conditions



IEC 62541-9:2015-03(en-fr)

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

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

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

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

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

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 It also gives information on projects, replaced and withdrawn publications.

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

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

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

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

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

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

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

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

IEC Just Published - webstore.iec.ch/justpublished
Restez informé sur les nouvelles publications IEC. Just Published détaille les nouvelles publications parues. Disponible en ligne et aussi une fois par mois par email.

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

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

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

IEC 62541-9

Edition 2.0 2015-03

INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC unified architecture – Part 9: Alarms and conditions

Architecture unifiée OPC – Partie 9: Alarmes et conditions

INTERNATIONAL ELECTROTECHNICAL COMMISSION

COMMISSION ELECTROTECHNIQUE INTERNATIONALE

ICS 25.040.40; 35.100

ISBN 978-2-8322-2382-6

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

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

_____

OPC UNIFIED ARCHITECTURE –

Part 9: Alarms and conditions

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
6) All users should ensure that they have the latest edition of this publication.
7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

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

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

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

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

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

d) Fixed ExclusiveLimitStateMachineType and ShelvedStatemachineType to be sub-types of FinitStateMachineType not StateMachineType. See 5.8.3 and 5.8.5.2 for additional details [ref 2091].

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

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

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

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

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

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

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

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

OPC UNIFIED ARCHITECTURE –

Part 9: Alarms and conditions

1 Scope

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

2 Normative references

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

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

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

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

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

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

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

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

3 Terms, definitions, and abbreviations

3.1 Terms and definitions

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

3.1.1
acknowledge
operator action that indicates recognition of a new *Alarm*

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

3.1.2
active
state for an Alarm that indicates that the situation the *Alarm* is representing currently exists

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

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

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

3.1.4
ConditionBranch
specific state of a *Condition*

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

3.1.5
ConditionSource
element which a specific *Condition* is based upon or related to

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

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

3.1.6
confirm
operator action informing the *Server* that a corrective action has been taken to address the cause of the *Alarm*

3.1.7
disable
system is configured such that the *Alarm* will not be generated even though the base *Alarm Condition* is present

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

3.1.8
operator
special user who is assigned to monitor and control a portion of a process

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

3.1.9
refresh
act of providing an update to an *Event Subscription* that provides all *Alarms* which are considered to be *Retained*

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

3.1.10
retain
alarm in a state that is interesting for a *Client* wishing to synchronize its state of *Conditions* with the *Server*'s state.

3.1.11
shelving
facility where the *Operator* is able to temporarily prevent an *Alarm* from being displayed to the *Operator* when it is causing the *Operator* a nuisance

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

3.1.12
suppress
act of determining whether an *Alarm* should not occur

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

3.2 Abbreviations and symbols

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

COM (Microsoft Windows) Component Object Model

DA Data Access

UA Unified Architecture

3.3 Used data types

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

Table 1 – Parameter Types defined in IEC 62541-3

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

Table 2 – Parameter Types defined in IEC 62541-4

Parameter Type
IntegerId
StatusCode

4 Concepts

4.1 General

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

4.2 Conditions

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

- a temperature exceeding a configured limit;

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

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

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

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

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

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

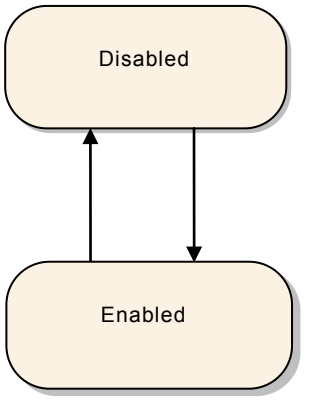


Figure 1 – Base Condition State Model

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

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

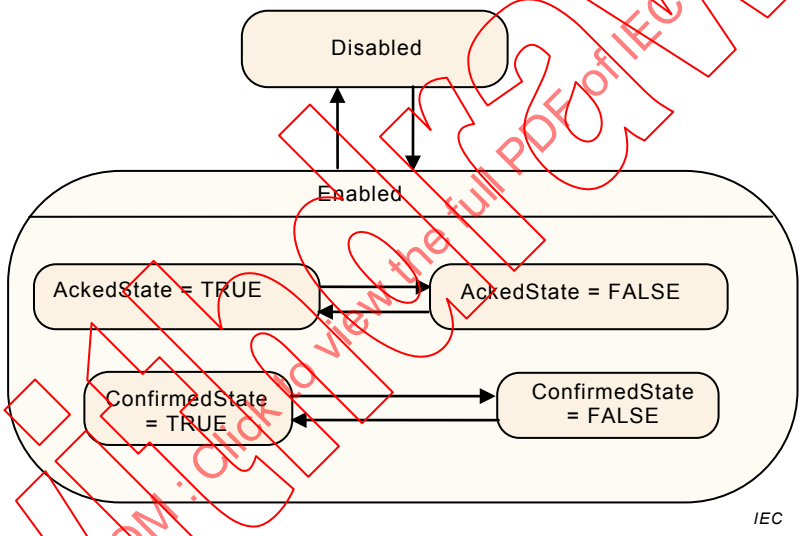


Figure 2 – AcknowledgeableConditions State Model

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

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

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

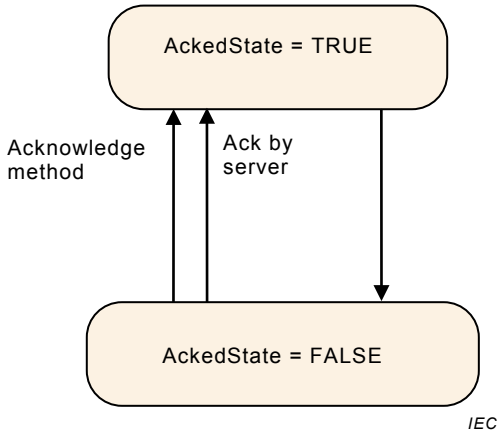


Figure 3 – Acknowledge State Model

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

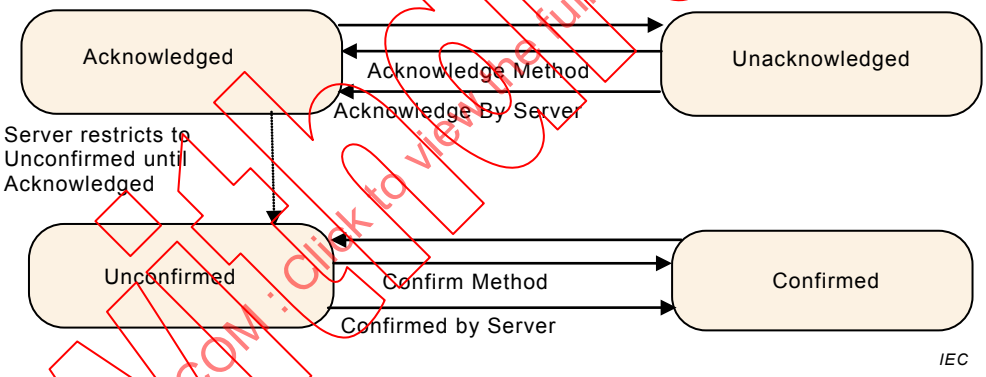


Figure 4 – Confirmed Acknowledge State Model

4.4 Previous states of Conditions

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

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

4.5 Condition state synchronization

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

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

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

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

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

- As described in 4.4 some systems require that previous states of a *Condition* are preserved for some time. Some *Servers* – in particular if they require acknowledgement of previous states – will maintain separate *ConditionBranches* for prior states that still need attention.

  ConditionRefresh shall issue *Event Notifications* for all interesting states (current and previous) of a *Condition* instance and *Clients* can therefore receive more than one *Event* for a *Condition* instance with different *BranchIds*.

- Under some circumstances a *Server* may not be capable of ensuring the *Client* is fully in sync with the current state of *Condition* instances. For example if the underlying system represented by the *Server* is reset or communications are lost for some period of time the *Server* may need to resynchronize itself with the underlying system. In these cases the *Server* shall send an *Event* of the *RefreshRequiredEventType* to advise the *Client* that a *Refresh* may be necessary. A *Client* receiving this special *Event* should initiate a *ConditionRefresh* as noted in this clause.

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

4.6 Severity, Quality, and Comment

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

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

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

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

4.7 Dialogs

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

4.8 Alarms

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

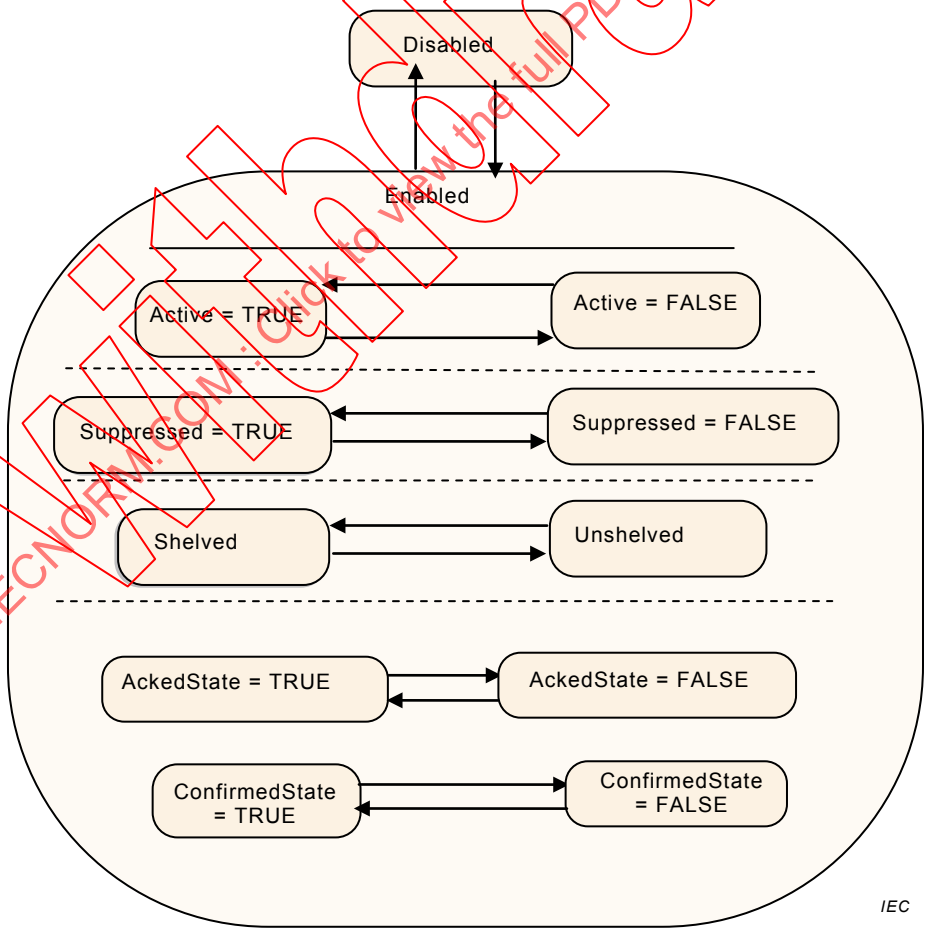


Figure 5 – Alarm State Machine Model

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

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

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

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

4.9 Multiple Active States

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

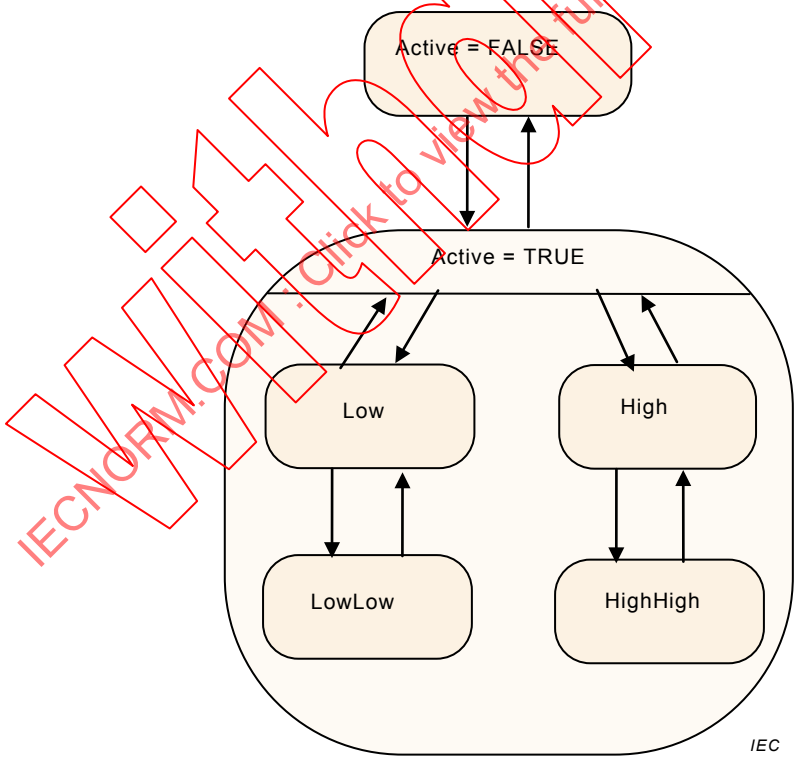


Figure 6 – Multiple Active States Example

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

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

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

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

4.10 *Condition* Instances in the Address Space

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

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

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

4.11 Alarm and Condition Auditing

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

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

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

5 Model

5.1 General

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

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

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

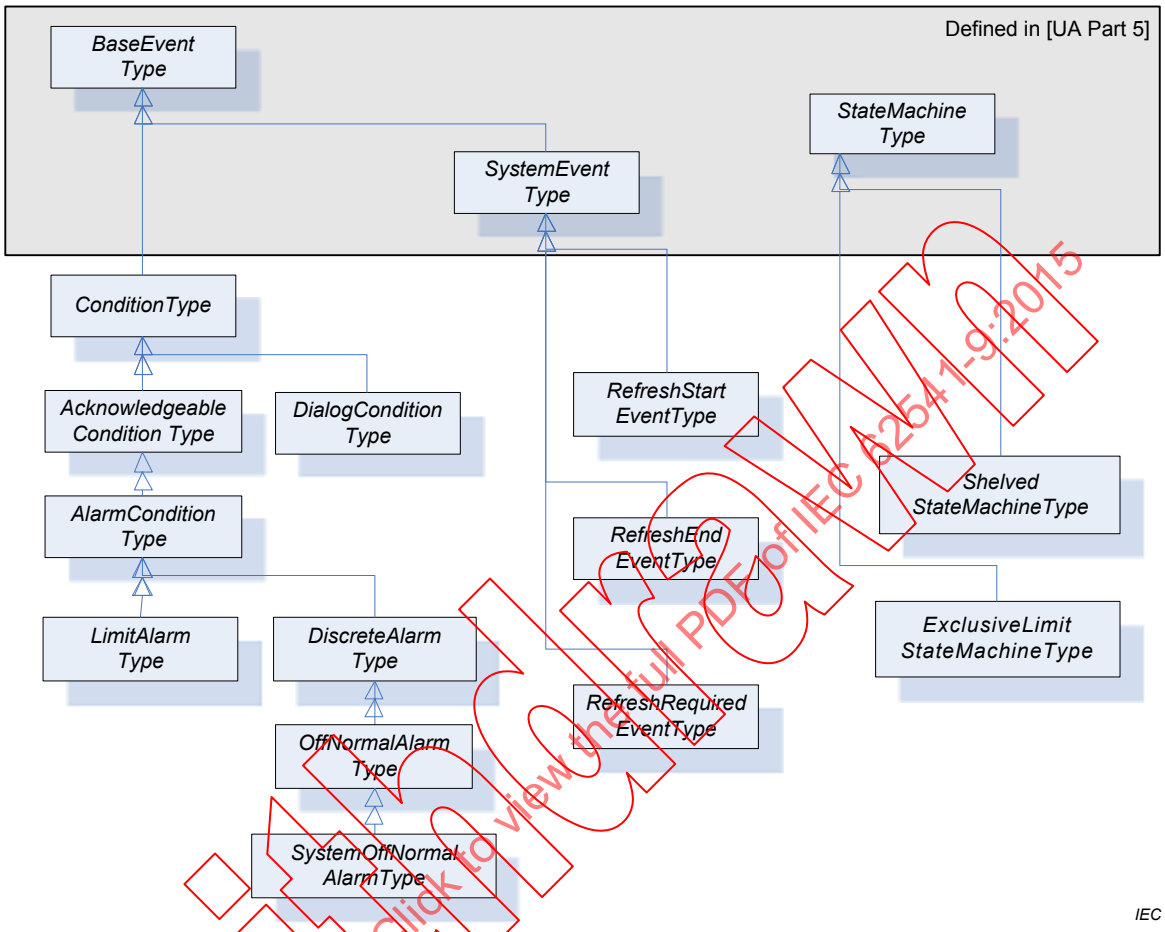


Figure 7 – ConditionType Hierarchy

5.2 Two-State State Machines

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

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

Table 3 – TwoStateVariableType Definition

Attribute	Value				
BrowseName	TwoStateVariableType				
DataType	LocalizedText				
ValueRank	-1 (-1 = Scalar)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *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 5.4.2		Optional
HasFalseSubState	StateMachine or TwoStateVariableType	<StateIdentifier>	Defined in 5.4.3		Optional

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

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

TransitionTime specifies the time when the current state was entered.

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

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

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

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

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

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

5.3 Condition Variables

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

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

Table 4 – ConditionVariableType Definition

Attribute	Value				
BrowseName	ConditionVariableType				
DataType	BaseDataType				
ValueRank	-2 (-2 = Any)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *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 Reference Types

5.4.1 General

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

5.4.2 HasTrueSubState ReferenceType

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

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

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

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

Table 5 – HasTrueSubState ReferenceType

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

5.4.3 HasFalseSubState ReferenceType

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

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

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

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

Table 6 – HasFalseSubState ReferenceType

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

5.5 Condition Model

5.5.1 General

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

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

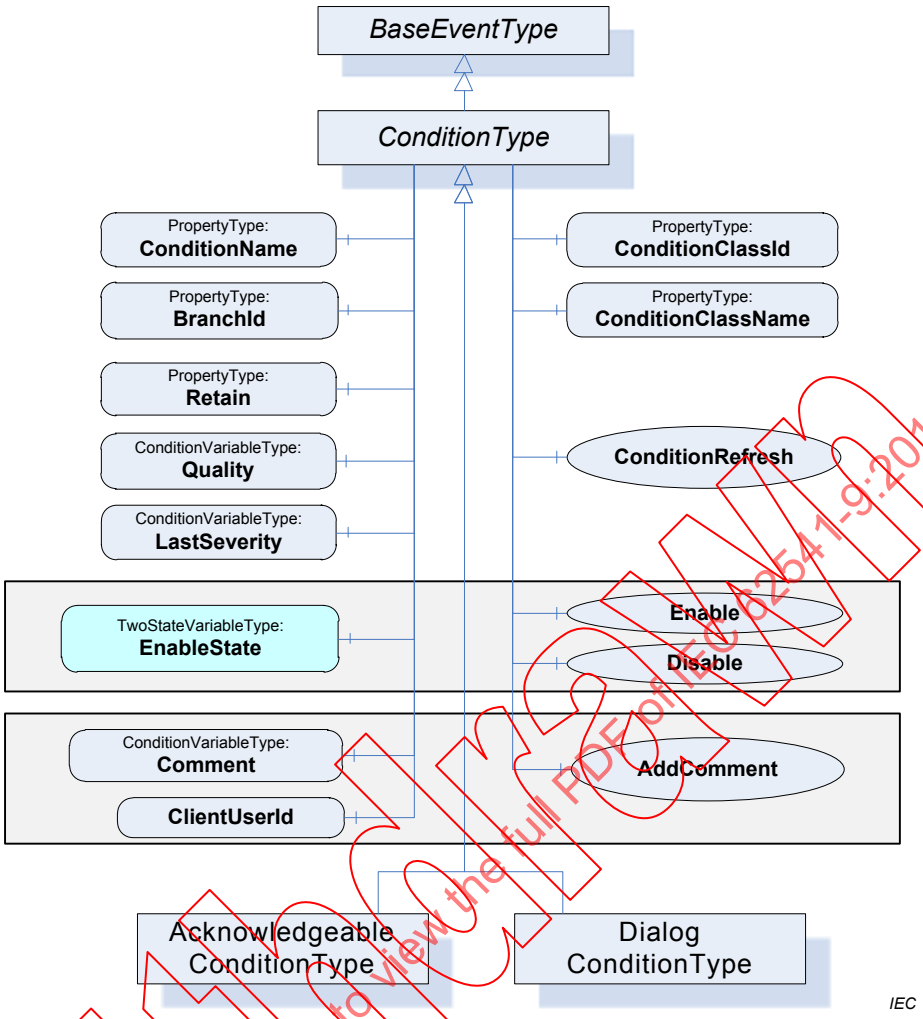


Figure 8 – Condition Model

5.5.2 ConditionType

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

Table 7 – ConditionType Definition

Attribute	Value				
BrowseName	ConditionType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**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	ConditionName	String	PropertyType	Mandatory
HasProperty	Variable	BranchId	NodeId	PropertyType	Mandatory
HasProperty	Variable	Retain	Boolean	PropertyType	Mandatory
HasComponent	Variable	EnabledState	LocalizedText	TwoStateVariableType	Mandatory
HasComponent	Variable	Quality	StatusCode	ConditionVariableType	Mandatory
HasComponent	Variable	LastSeverity	UInt16	ConditionVariableType	Mandatory
HasComponent	Variable	Comment	LocalizedText	ConditionVariableType	Mandatory
HasProperty	Variable	ClientUserId	String	PropertyType	Mandatory
HasComponent	Method	Disable	Defined in 5.5.4		Mandatory
HasComponent	Method	Enable	Defined in 5.5.5		Mandatory
HasComponent	Method	AddComment	Defined in 5.5.6		Mandatory
HasComponent	Method	ConditionRefresh	Defined in 5.5.7		None

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

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

ConditionClassName provides the display name of the *ConditionClassType*.

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

BranchId is Null for all *Event Notifications* that relate to the current state of the *Condition* instance. If *BranchId* is not Null it identifies a previous state of this *Condition* instance that still needs attention by an *Operator*. If the current *ConditionBranch* is transformed into a previous *ConditionBranch* then the Server needs to assign a non-null BranchId. An initial *Event* for the branch will generated with the values of the *ConditionBranch* and the new BranchId. The *ConditionBranch* can be updated many times before it is no longer needed. When the *ConditionBranch* no longer requires *Operator* input the final *Event* will have *Retain* set to FALSE. The retain bit on the current Event is TRUE, as long as any ConditionBranches require Operator input. See 4.4 for more information about the need for creating and maintaining previous *ConditionBranches* and Clause B.1 for an example using branches. The BranchId *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 *Clients*.
- When the *Condition* instance enters the enabled state, the *Condition* shall be evaluated and all of its *Properties* updated to reflect the current values. If this evaluation causes the *Retain Property* to transition to TRUE for any *ConditionBranch*, then an *Event Notification* shall be generated for that *ConditionBranch*.
- The *Server* may choose to continue to test for a *Condition* instance while it is *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.

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

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

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

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

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

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

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

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

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

Table 8 – Simple Attribute Operand

Name	Type	Description
SimpleAttributeOperand		
typeId	NodeId	*NodeId* of the *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 standard does not define a standard way to make them visible.

5.5.4 Disable Method

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

Signature

```
Disable();
```

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

Table 9 – Disable Result Codes

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

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

Table 10 – Disable Method AddressSpace Definition

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

5.5.5 Enable Method

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

Signature

```
Enable();
```

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

Table 11 – Enable Result Codes

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

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

Table 12 – Enable Method AddressSpace Definition

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

5.5.6 AddComment Method

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

Signature

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

The parameters are defined in Table 13

Table 13 – AddComment Arguments

Argument	Description
EventId	EventId identifying a particular *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 14 (defined in Call Service).

Table 14 – AddComment result Codes

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

Comments

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

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

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

Table 15 – AddComment Method AddressSpace Definition

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

5.5.7 ConditionRefresh Method

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

Signature

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

The parameters are define in Table 16

Table 16 – ConditionRefresh Parameters

Argument	Description
SubscriptionId	A valid *Subscription* Id of the *Subscription* to be refreshed.

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

Table 17 – ConditionRefresh ReturnCodes

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

Comments

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

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

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

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

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

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

Table 18 – ConditionRefresh Method AddressSpace Definition

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

5.6 Dialog Model

5.6.1 General

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

5.6.2 DialogConditionType

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

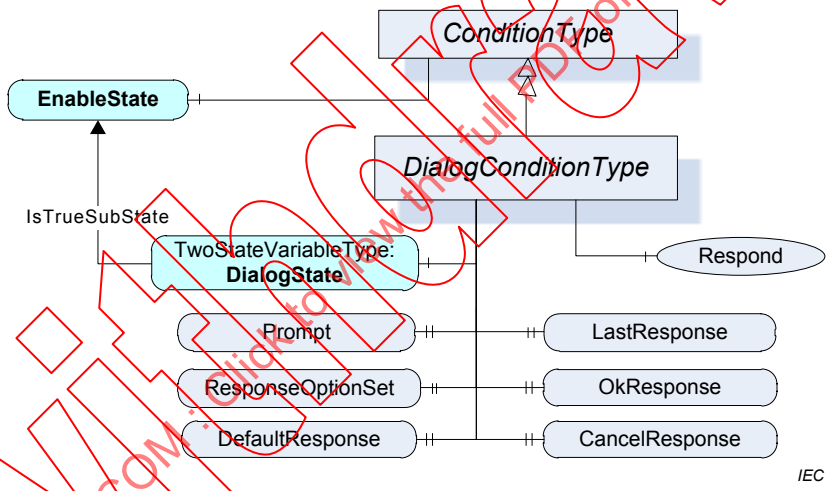


Figure 9 – DialogConditionType Overview

Table 19 – DialogConditionType Definition

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

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

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

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

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

Typical combinations of response options are

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

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

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

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

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

5.6.3 Respond Method

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

Signature

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

The parameters are defined in Table 20.

Table 20 – Repond Parameters

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

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

Table 21 – Respond ResultCodes

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

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

Table 22 – Respond Method AddressSpace Definition

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

5.7 Acknowledgeable Condition Model

5.7.1 General

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

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

5.7.2 AcknowledgeableConditionType

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

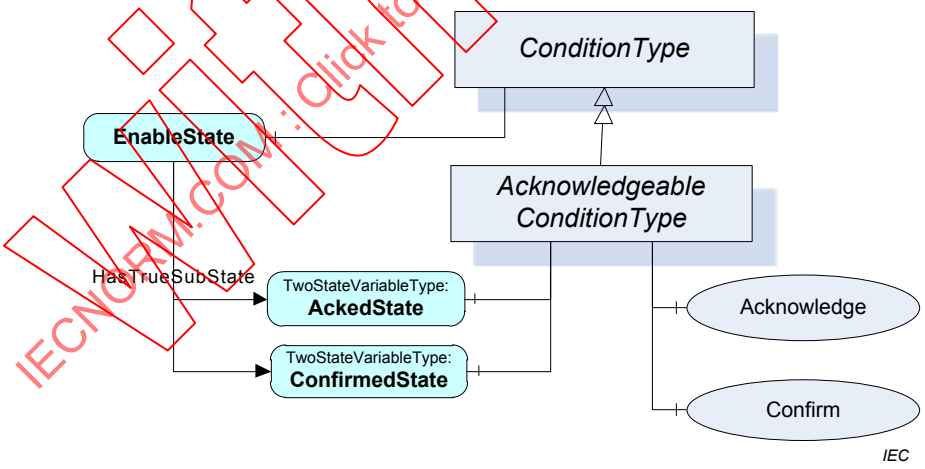


Figure 10 – AcknowledgeableConditionType Overview

Table 23 – AcknowledgeableConditionType Definition

Attribute	Value				
BrowseName	AcknowledgeableConditionType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *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 sub-states of the TRUE *EnabledState*. See 4.3 for more information about acknowledgement and confirmation models. The *EventId* used in the *Event Notification* is considered the identifier of this state and has to be used when calling the *Methods* for acknowledgement or confirmation.

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

5.7.3 Acknowledge Method

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

Signature

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

The parameters are defined in Table 24.

Table 24 – Acknowledge Parameters

Argument	Description
EventId	EventId identifying a particular *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 25 (defined in Call Service).

Table 25 – Acknowledge result codes

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

Comments

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

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

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

Table 26 – Acknowledge Method AddressSpace Definition

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

5.7.4 Confirm Method

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

Signature

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

The parameters are defined in Table 27.

Table 27 – Confirm Method Parameters

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

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

Table 28 – Confirm Result Codes

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

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* is set to TRUE and the *Comment Property* contains the text of the optional comment argument. If a previous state is confirmed, the *BranchId* and all *Condition* values of this branch will be reported.

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

Table 29 – Confirm Method AddressSpace Definition

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

5.8 Alarm Model

5.8.1 General

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

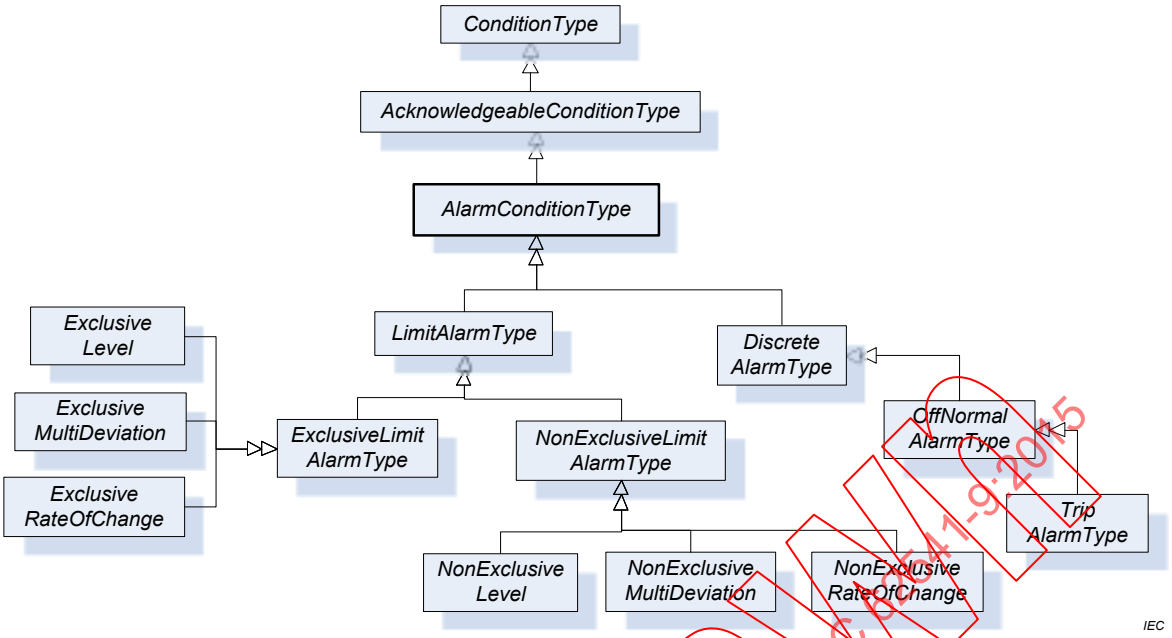


Figure 11 – AlarmConditionType Hierarchy Model

5.8.2 AlarmConditionType

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

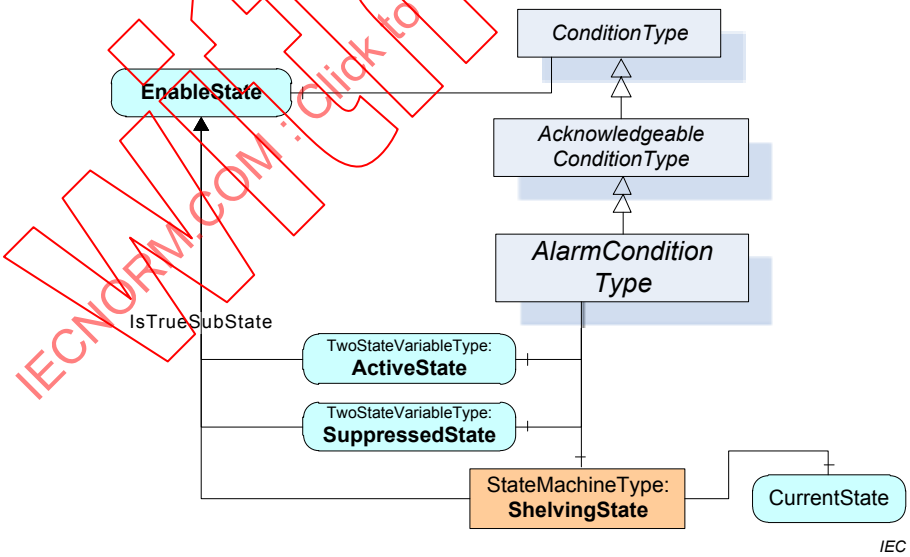


Figure 12 – Alarm Model

Table 30 – AlarmConditionType Definition

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

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

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

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

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

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

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

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

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

5.8.3 ShelvedStateMachineType

5.8.3.1 Overview

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

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

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

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

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

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

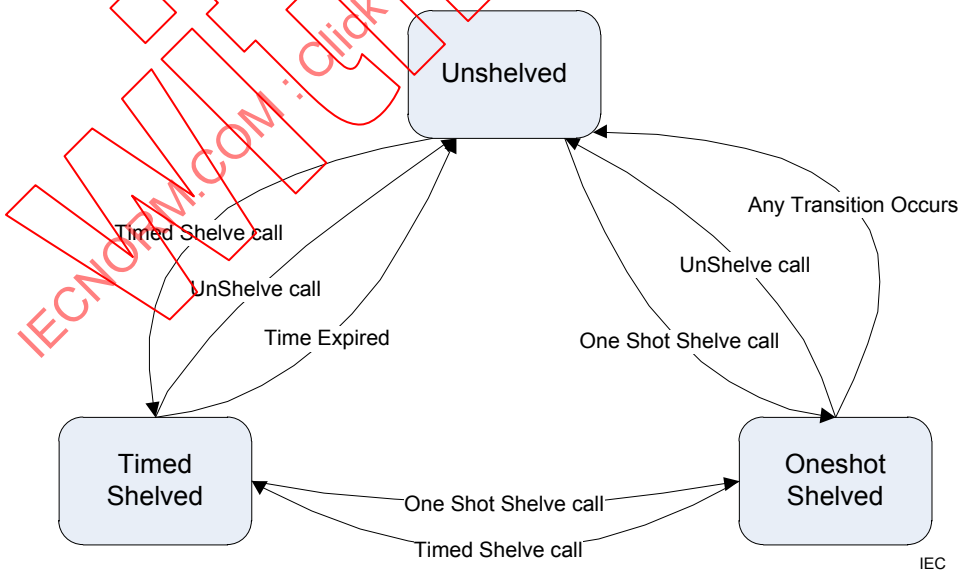


Figure 13 – Shelve state transitions

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

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

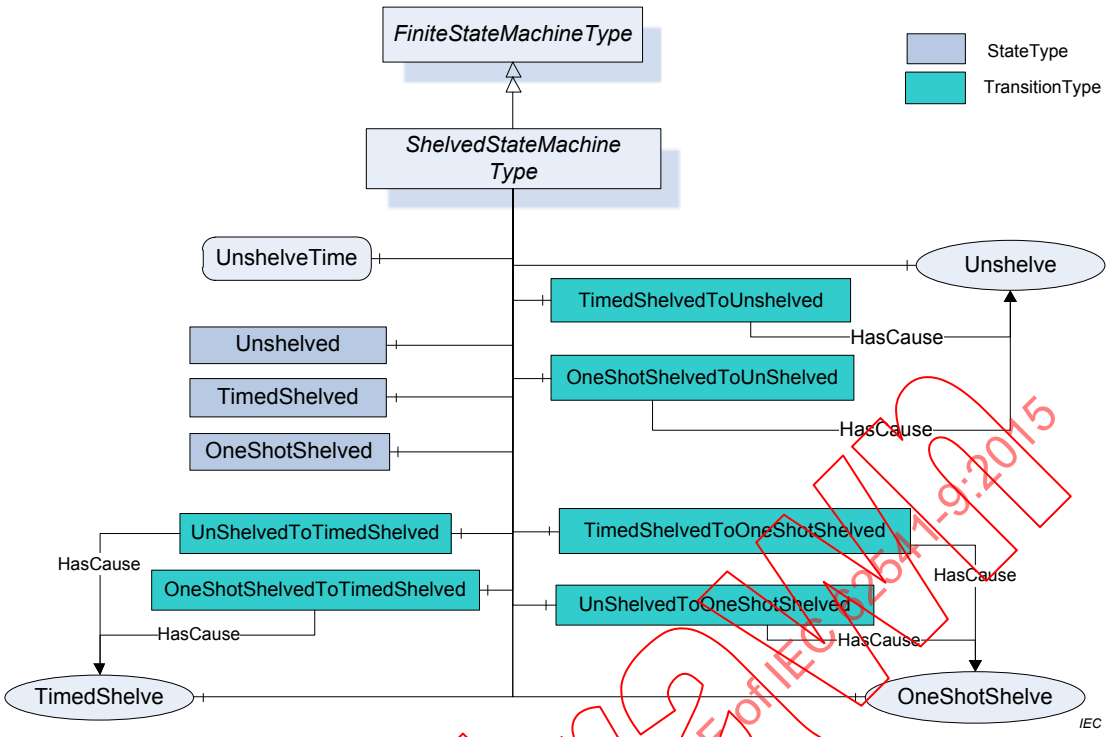


Figure 14 – Shelved State Machine Model

Table 31 –ShelvedStateMachine Definition

Attribute	**Value**				
BrowseName	ShelvedStateMachineType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the 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 5.8.3.3		Mandatory
HasComponent	Method	OneShotShelve	Defined in 5.8.3.4		Mandatory
HasComponent	Method	Unshelve	Defined in 5.8.3.2		Mandatory

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

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

Table 32 – ShelvedStateMachine Transitions

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

5.8.3.2 Unshelve Method

Unshelve sets the *AlarmCondition* to the Unshelved state.

Signature

```
Unshelve( );
```

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

Table 33 – Unshelve Result Codes

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

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

Table 34 – Unshelve Method AddressSpace Definition

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

5.8.3.3 TimedShelve Method

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

Signature

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

Table 35 – TimedShelve Parameters

Argument	Description
ShelvingTime	Specifies a fixed time for which the *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 36 – TimedShelve Result Codes

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

Comments

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

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

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

Table 37 – TimedShelve Method AddressSpace Definition

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

5.8.3.4 OneShotShelve Method

OneShotShelve sets the *AlarmCondition* to the OneShotShelved state.

Signature

```
OneShotShelve( );
```

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

Table 38 – OneShotShelve Result Codes

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

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

Table 39 – OneShotShelve Method AddressSpace Definition

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

5.8.4 LimitAlarmType

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

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

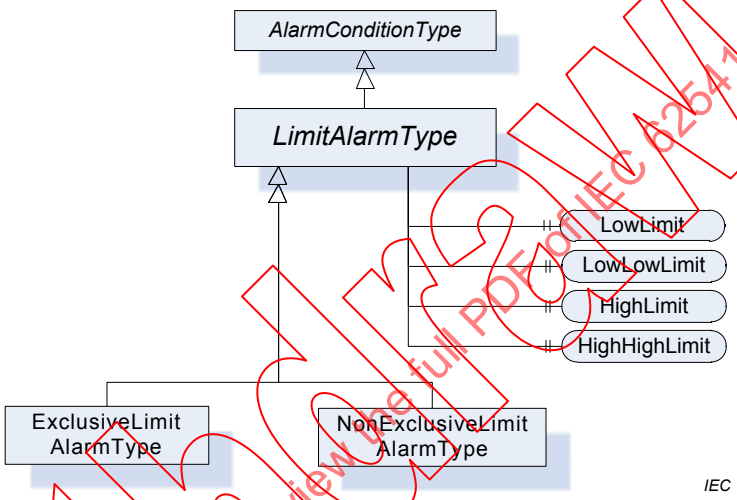


Figure 15 – LimitAlarmType

The *LimitAlarmType* is formally defined in Table 40.

Table 40 – LimitAlarmType Definition

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

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

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

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

5.8.5 ExclusiveLimit Types

5.8.5.1 Overview

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

5.8.5.2 ExclusiveLimitStateMachineType

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

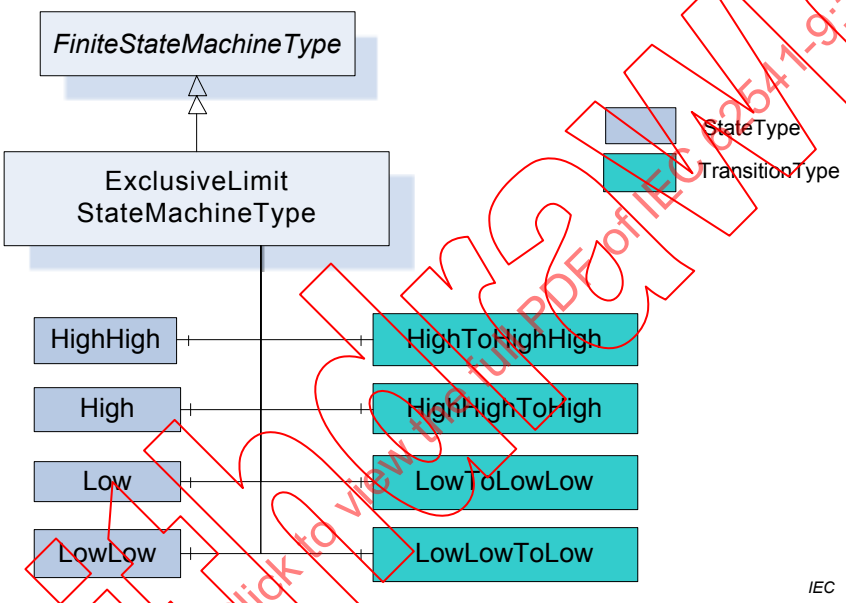


Figure 16 – ExclusiveLimitStateMachine

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

Table 41 – ExclusiveLimitStateMachineType Definition

Attribute	**Value**				
BrowseName	ExclusiveLimitStateMachineType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *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 42 – ExclusiveLimitStateMachineType Transitions

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

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

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

5.8.5.3 ExclusiveLimitAlarmType

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

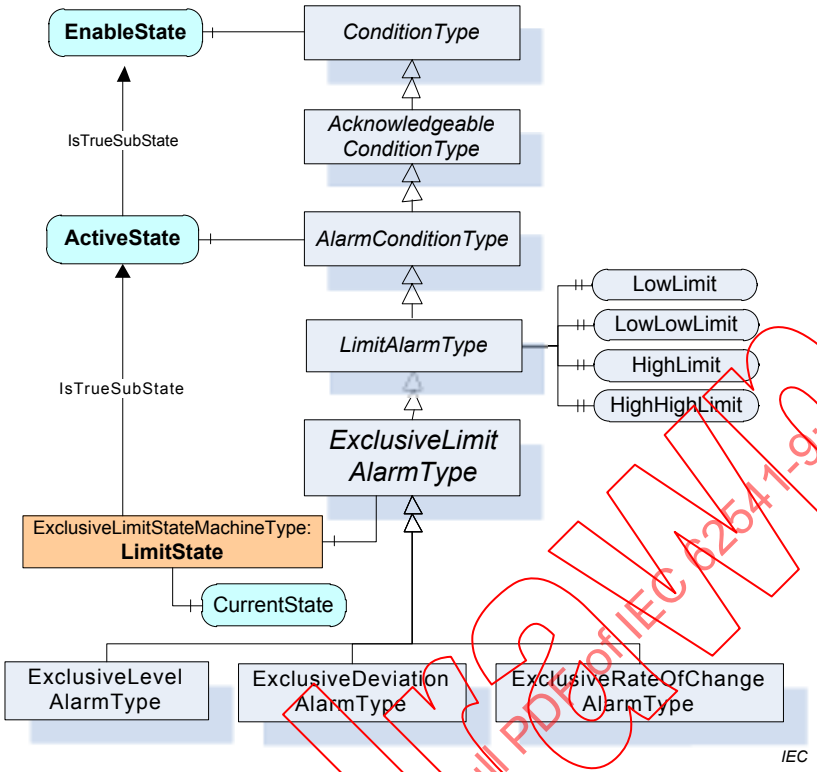


Figure 17 – ExclusiveLimitAlarmType

The *ExclusiveLimitAlarmType* is formally defined in Table 43.

Table 43 – ExclusiveLimitAlarmType Definition

Attribute	**Value**				
BrowseName	ExclusiveLimitAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the LimitAlarmType defined in 5.8.4.					
HasSubtype	ObjectType	ExclusiveLevelAlarmType	Defined in 5.8.7.3		
HasSubtype	ObjectType	ExclusiveDeviationAlarm Type	Defined in 5.8.8.3		
HasSubtype	ObjectType	ExclusiveRateOfChange AlarmType	Defined in 5.8.9.3		
HasComponent	Object	LimitState		*ExclusiveLimitStateMachineType*	Mandatory

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

5.8.6 NonExclusiveLimitAlarmType

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

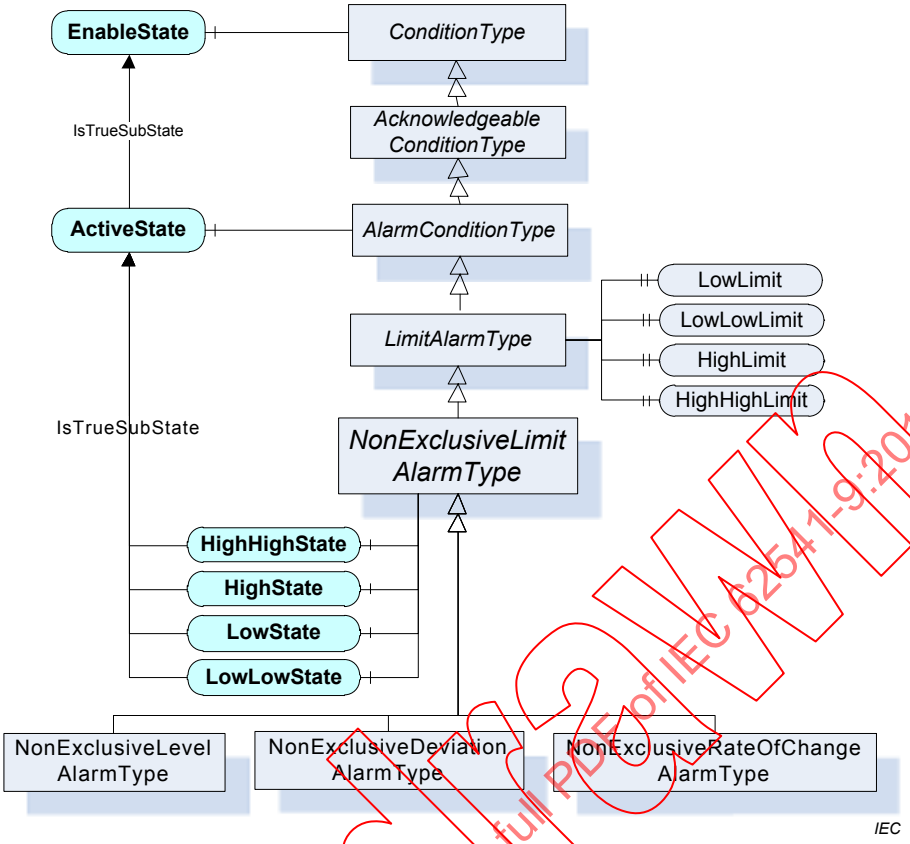


Figure 18 – NonExclusiveLimitAlarmType

The *NonExclusiveLimitAlarmType* is formally defined in Table 44.

Table 44 – NonExclusiveLimitAlarmType Definition

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

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

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

5.8.7 Level Alarm

5.8.7.1 Overview

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

5.8.7.2 NonExclusiveLevelAlarmType

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

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

Table 45 – NonExclusiveLevelAlarmType Definition

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

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

5.8.7.3 ExclusiveLevelAlarmType

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

Table 46 – ExclusiveLevelAlarmType Definition

Attribute	Value				
BrowseName	ExclusiveLevelAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherits the Properties of the ExclusiveLimitAlarmType defined in 5.8.5.3.					

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

5.8.8 Deviation Alarm

5.8.8.1 Overview

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

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

5.8.8.2 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* should be used.

The *NonExclusiveDeviationAlarmType* is based on the *NonExclusiveLimitAlarmType*. It is formally defined in Table 47.

Table 47 – NonExclusiveDeviationAlarmType Definition

Attribute	Value				
BrowseName	NonExclusiveDeviationAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the NonExclusiveLimitAlarmType defined in 5.8.6.					
HasProperty	Variable	SetpointNode	NodeId	PropertyType	Mandatory

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.

5.8.8.3 ExclusiveDeviationAlarmType

The *ExclusiveDeviationAlarmType* is utilized with multiple mutually exclusive limits. It is formally defined in Table 48.

Table 48 – ExclusiveDeviationAlarmType Definition

Attribute	Value				
BrowseName	ExclusiveDeviationAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Inherits the *Properties* of the *ExclusiveLimitAlarmType* defined in 5.8.5.3.					
HasProperty	Variable	SetpointNode	NodeId	PropertyType	Mandatory

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.

5.8.9 Rate of Change

5.8.9.1 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 meter. For example a tank may have a High limit for the *Rate* of *Change* of its level (measured in meters) which would be 4 m/min. If the tank level changes at a rate that is greater than 4 m/min then the High sub state is entered.

5.8.9.2 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* should be used

The *NonExclusiveRateOfChangeAlarmType* is based on the *NonExclusiveLimitAlarmType*. It is formally defined in Table 49.

Table 49 – NonExclusiveRateOfChangeAlarmType Definition

Attribute	Value				
BrowseName	NonExclusiveRateOfChangeAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the NonExclusiveLimitAlarmType defined in 5.8.6.					

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

5.8.9.3 ExclusiveRateOfChangeAlarmType

ExclusiveRateOfChangeAlarmType is utilized with multiple mutually exclusive limits. It is formally defined in Table 50.

Table 50 – ExclusiveRateOfChangeAlarmType Definition

Attribute	Value				
BrowseName	ExclusiveRateOfChangeAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherits the *Properties* of the *ExclusiveLimitAlarmType* defined in 5.8.5.3.					

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

5.8.10 Discrete Alarms

5.8.10.1 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 sub types defined in this standard is illustrated in Figure 19. It is formally defined in Table 51.

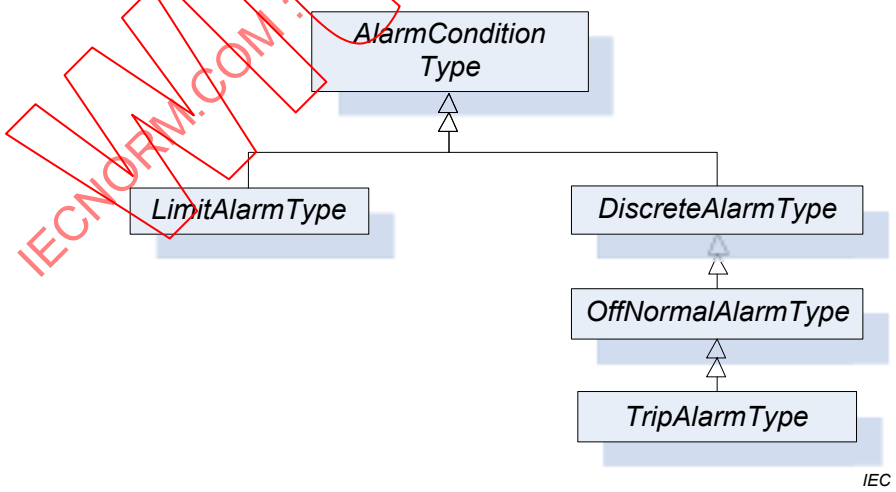


Figure 19 – DiscreteAlarmType Hierarchy

Table 51 – 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 5.8.8		

5.8.10.2 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 52. This sub type 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 52 – OffNormalAlarmType Definition

Attribute	Value				
BrowseName	OffNormalAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the DiscreteAlarmType defined in 5.8.10.1					
HasSubtype	*ObjectType*	TripAlarmType	Defined in 5.8.10.4		
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.

5.8.10.3 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 53.

Table 53 – SystemOffNormalAlarmType Definition

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

5.8.10.4 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 54. This *Type* is mainly used for categorization.

Table 54 – TripAlarmType Definition

Attribute	Value				
BrowseName	TripAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *OffNormalAlarmType* defined in 5.8.10.2.					

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 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 20 informally describes the hierarchy of *ConditionClass Types* defined in this standard.

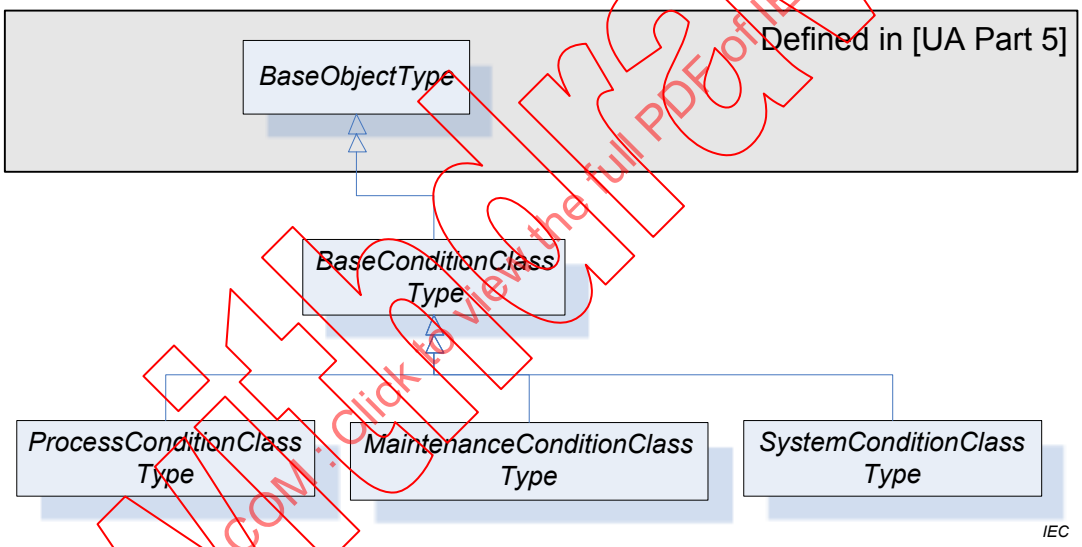


Figure 20 – ConditionClass Type Hierarchy

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

5.9.2 Base ConditionClassType

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 55.

Table 55 – 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 56.

Table 56 – 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 57. No further definition is provided here. It is expected that other standards groups will define domain-specific sub-types.

Table 57 – 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 58. 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 groups or vendors will define domain-specific sub-types.

Table 58 – SystemConditionClassType Definition

Attribute	Value				
BrowseName	SystemConditionClassType				
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 sub-types of *AuditUpdateMethodEventTypes* that will be generated in response to the *Methods* defined in this standard. They are illustrated in Figure 21.

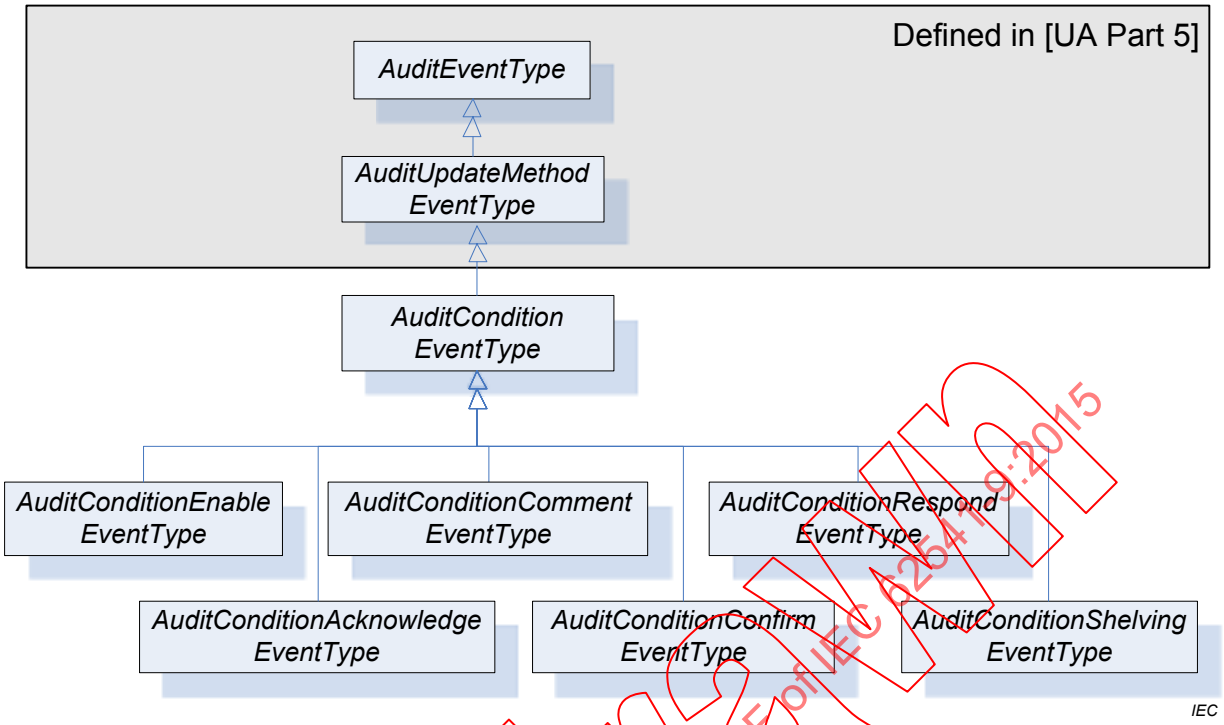


Figure 21 – AuditEvent Hierarchy

Audit Condition EventTypes 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 *Event* type.

5.10.2 AuditConditionEventType

This *EventType* is used to subsume all *Audit Condition EventTypes*. It is formally defined in Table 59.

Table 59 – AuditConditionEventType Definition

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

Audit Condition EventTypes 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 *Event* Type 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 60.

Table 60 – 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 61.

Table 61 – AuditConditionCommentEventType Definition

Attribute	Value				
BrowseName	AuditConditionCommentEventType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	EventId	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 *EventId* 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. It is formally defined in Table 62.

Table 62 – 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 63.

Table 63 – AuditConditionAcknowledgeEventType Definition

Attribute		Value			
BrowseName		AuditConditionCommentEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	EventId	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 *EventId* 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 64.

Table 64 – AuditConditionConfirmEventType Definition

Attribute		Value			
BrowseName		AuditConditionCommentEventType			
IsAbstract		False			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	EventId	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 *EventId* 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 65.

Table 65 – 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.

5.11 Condition Refresh Related Events

5.11.1 Overview

Following are sub-types of *SystemEventTypes* that will be generated in response to a *Refresh Methods* call. They are illustrated in Figure 22.

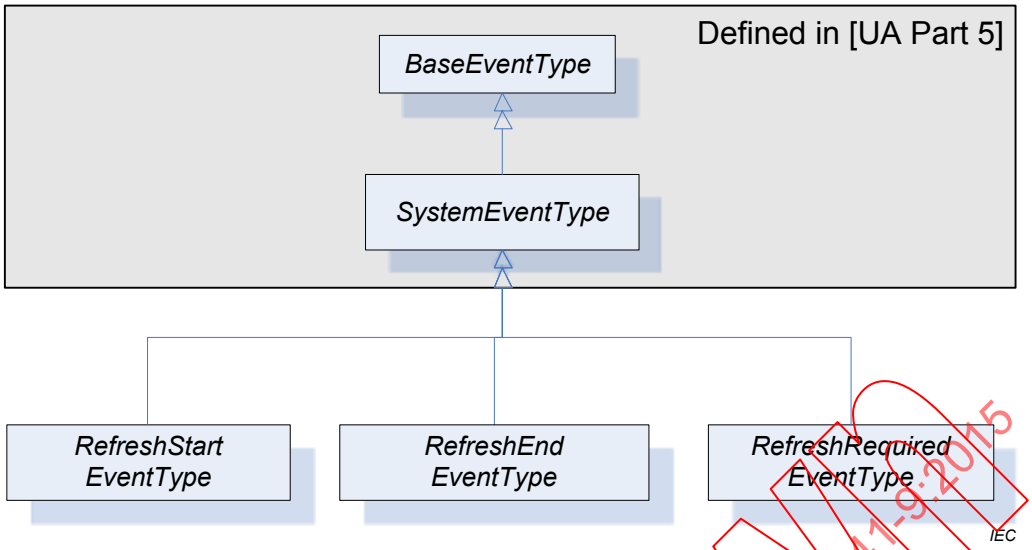


Figure 22 – Refresh Related Event Hierarchy

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 66.

Table 66 – 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 67.

Table 67 – 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 68.

Table 68 – 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 69.

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 sub type 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 be found in 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 69 – HasCondition ReferenceType

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

5.13 Alarm and Condition Status Codes

Table 70 defines the *StatusCodes* defined for *Alarm* and *Conditions*.

Table 70 – Alarm and 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 section describes behaviour that is expected from an OPC UA *Server* that is implementing the *A&C Information Model*. In particular this clause 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 do 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 *Event*s, *Alarms* and *Conditions*. This may be accomplished by sharing of information between redundant *Server* (such as actual *Event*s) 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 notifies 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 Event Notifier 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 standard. 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 23 shows such a hierarchy. Note that *HasNotifier* is a sub-type of *HasEventSource*. I.e. the target *Node* of a *HasNotifier Reference* (an *Event* notifying *Object*) may 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*.

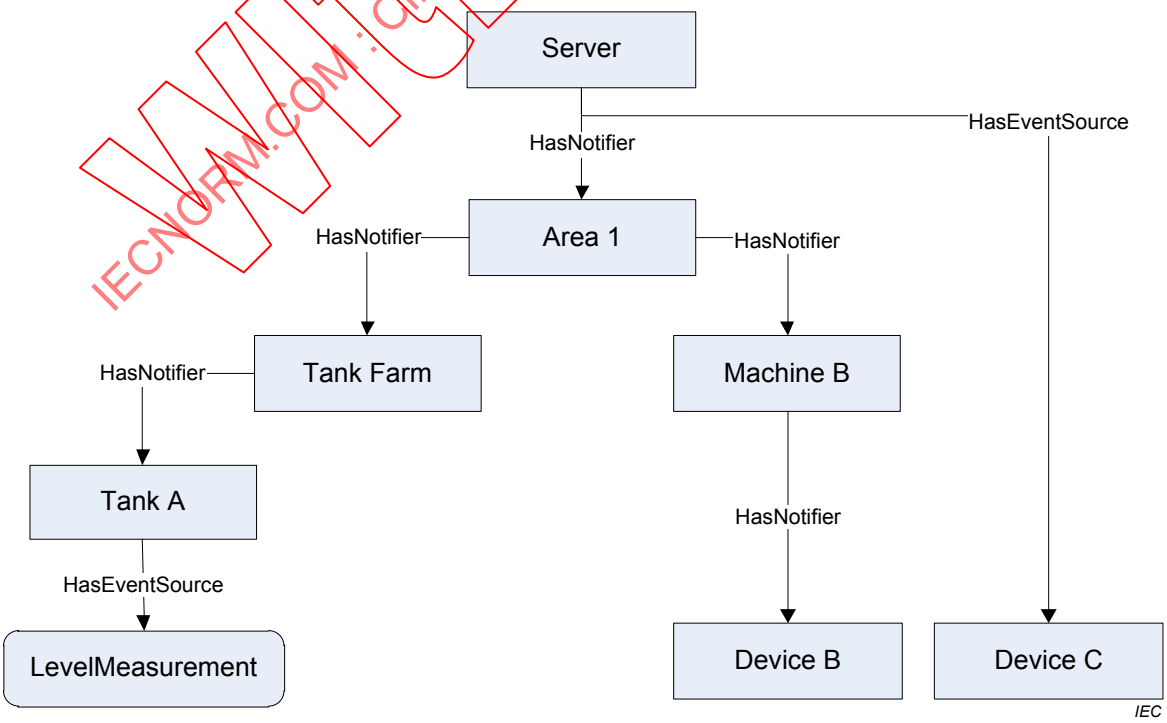


Figure 23 – Typical Event Hierarchy

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 sub-types like the *HasNotifier Reference*) and then browsing for *HasCondition References* from all target *Nodes* of the discovered *References*.

Figure 24 shows the application of the *HasCondition Reference* in an *Event* 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*.

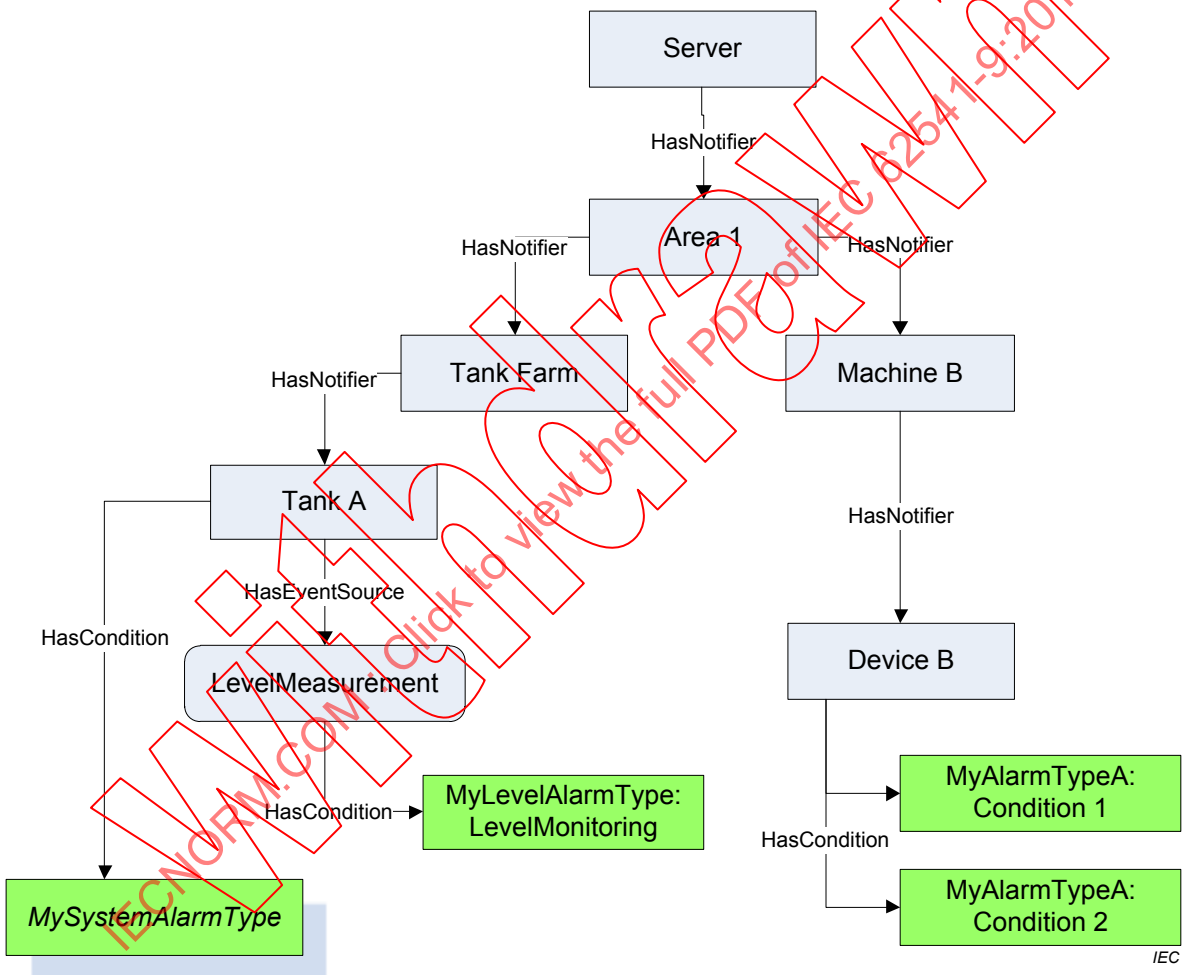


Figure 24 – Use of HasCondition in an Event Hierarchy

6.4 Conditions in InstanceDeclarations

Figure 25 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.

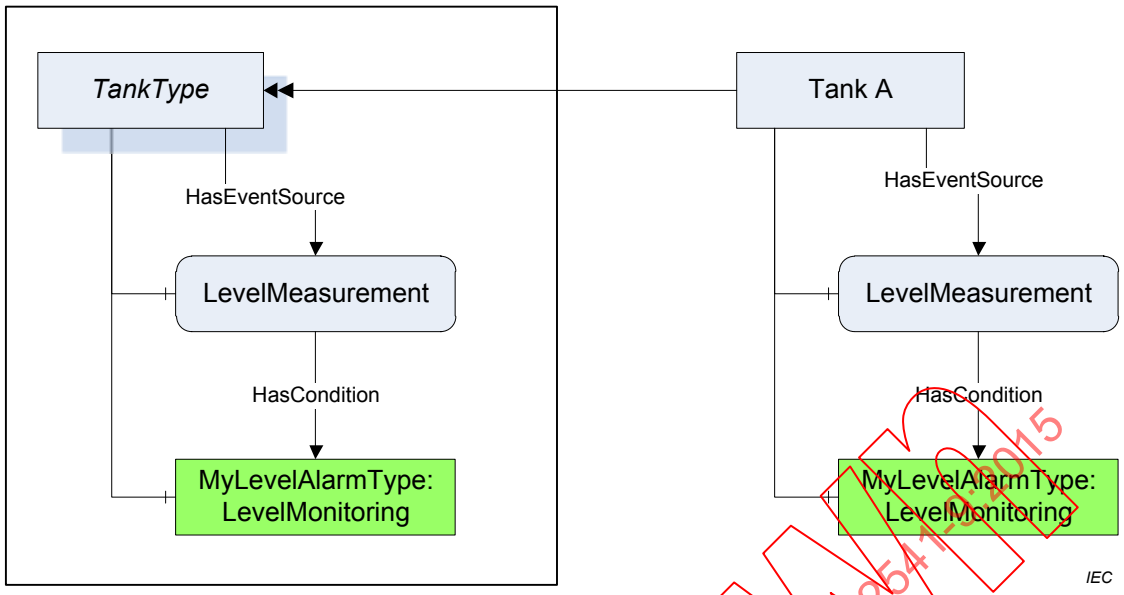


Figure 25 – Use of HasCondition in an InstanceDeclaration

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 26 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*.

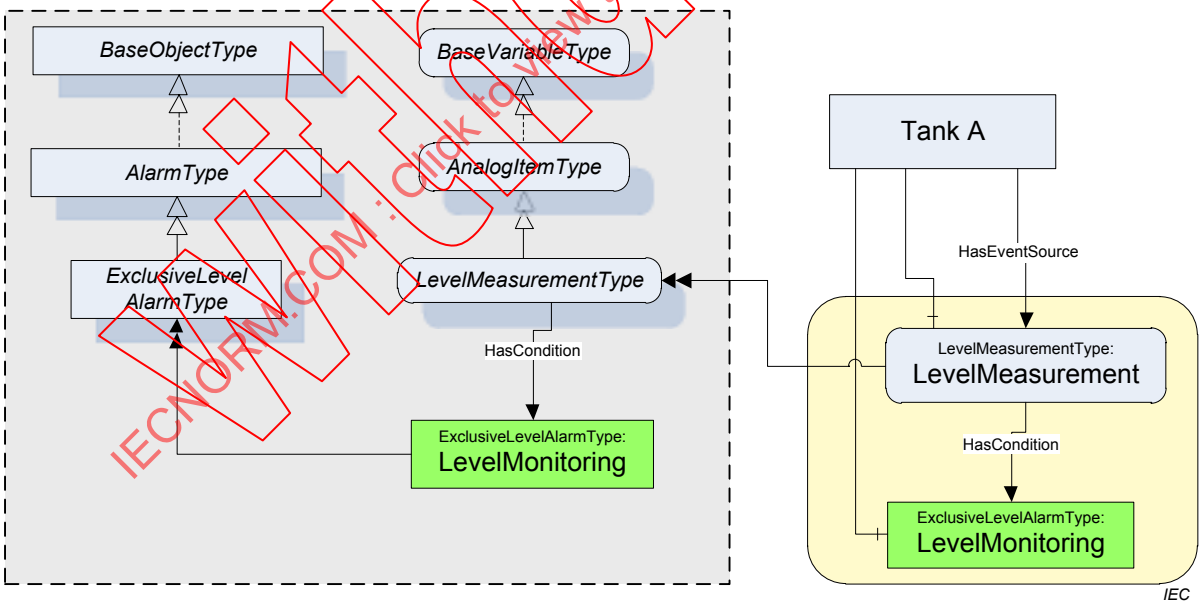


Figure 26 – Use of HasCondition in a VariableType

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
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
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
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. The tables 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*.

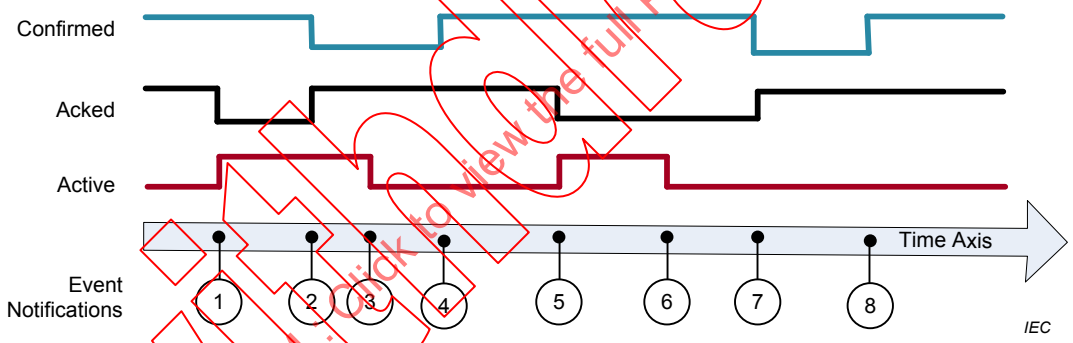


Figure B.1 – Single State Example

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*.

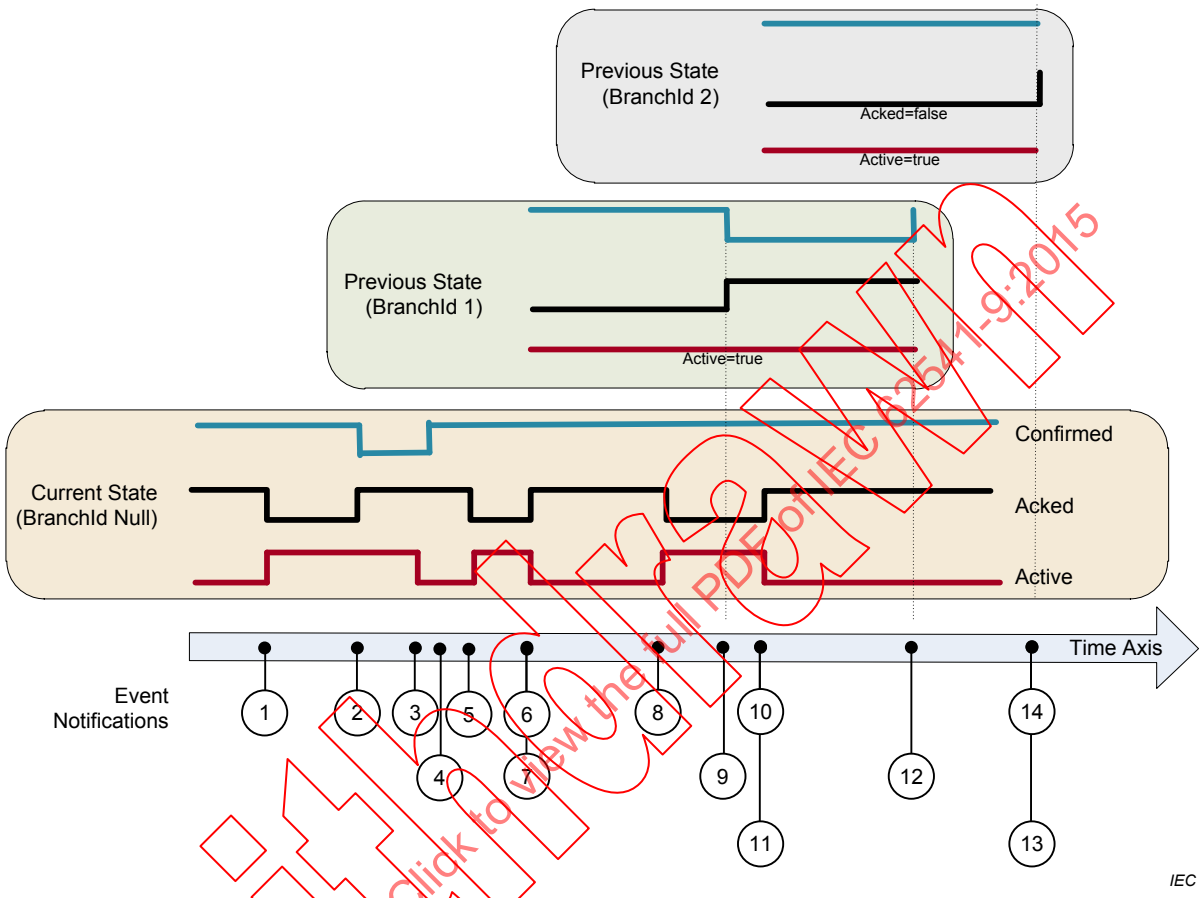


Figure B.2 – Previous State Example

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, Branch2 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
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 Address Space 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.

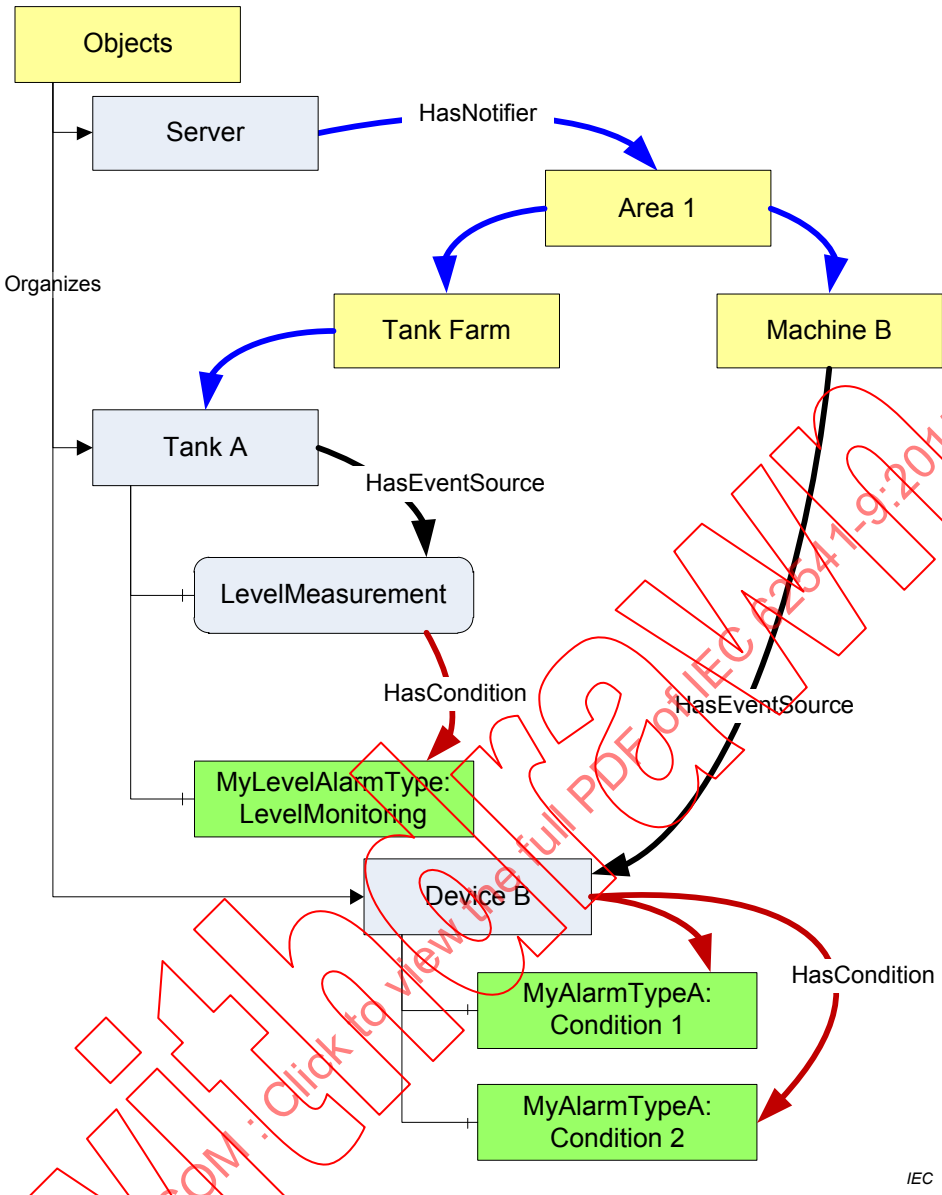


Figure B.3 – HasCondition used with Condition instances

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

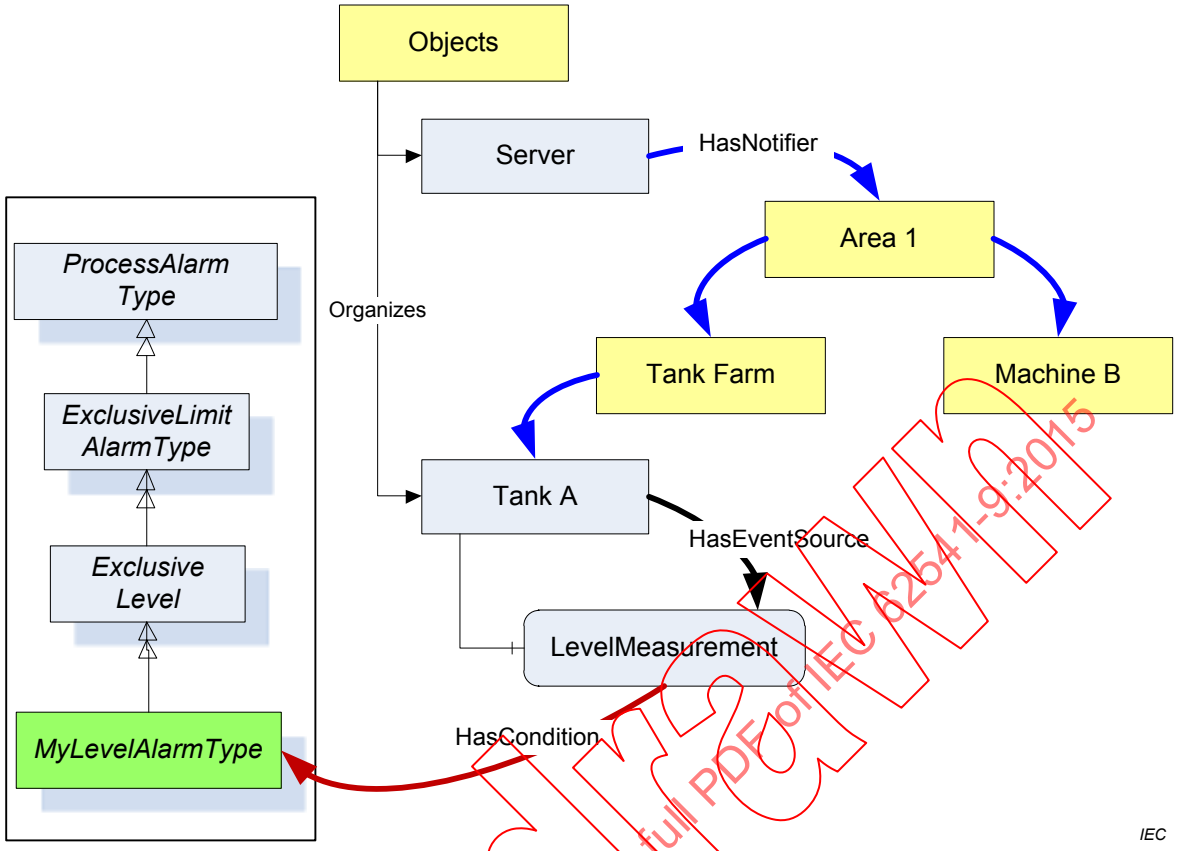


Figure B.4 – HasCondition reference to a Condition Type

Figure B.5 provides an example where the *HasCondition Reference* is already defined in the *Type* system. The *Reference* can 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

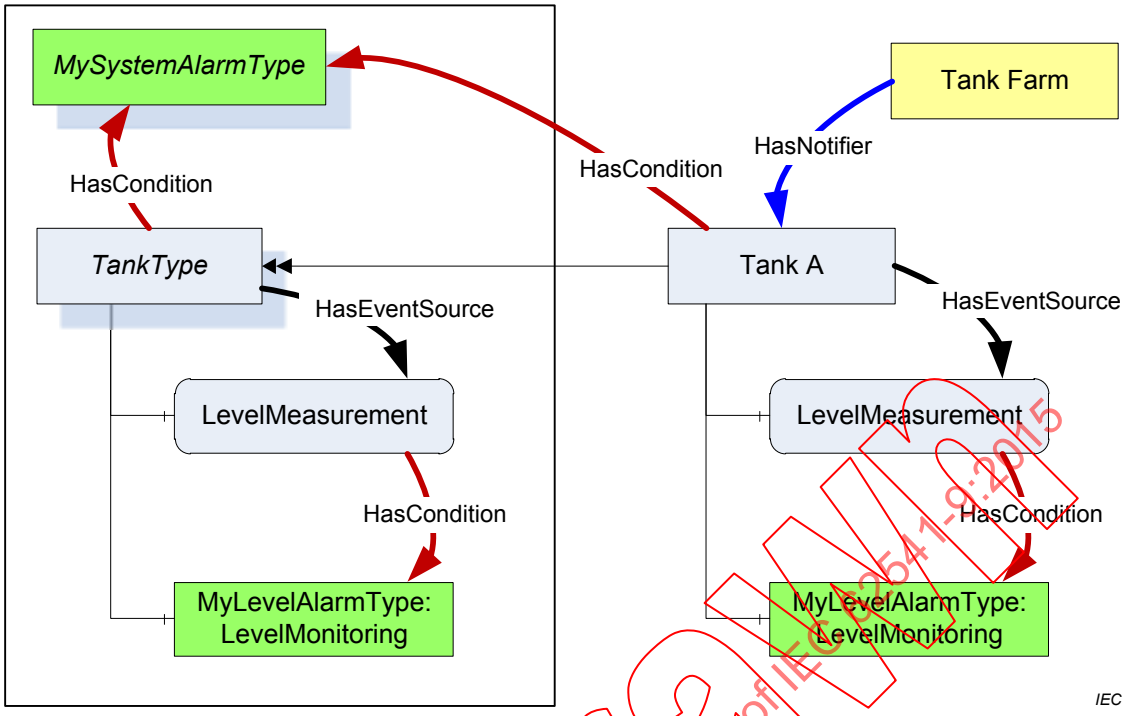


Figure B.5 – HasCondition used with an instance declaration

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 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 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 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 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* hierarchy.

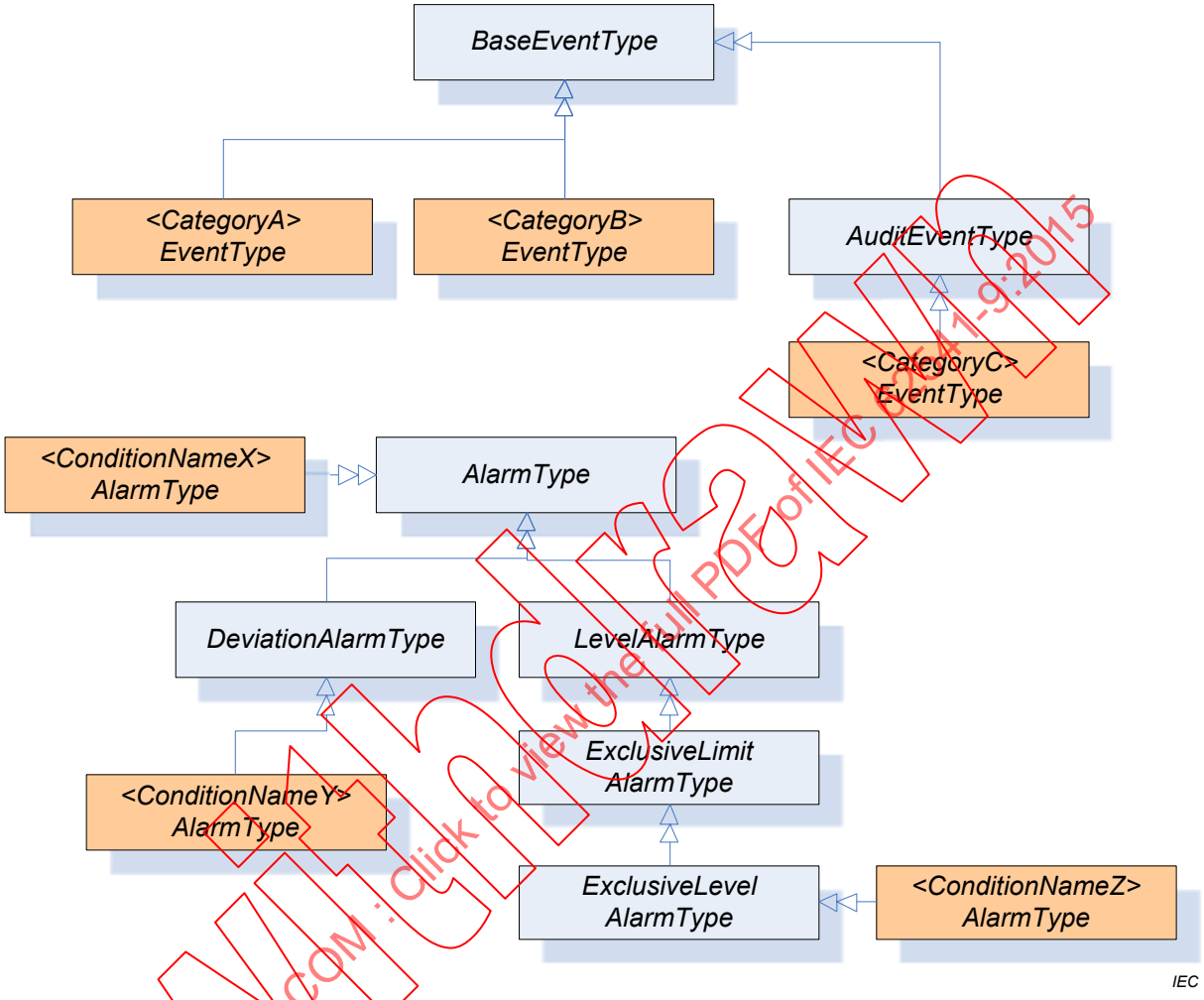


Figure D.1 – The Type Model of a Wrapped COM AE Server

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 *Event*s 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 that 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 that 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 that 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	-

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 can be acknowledged by passing the EventId to the *Acknowledge Method* defined on the *AcknowledgeableConditionType*.

*Condition*s cannot 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 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 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 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.

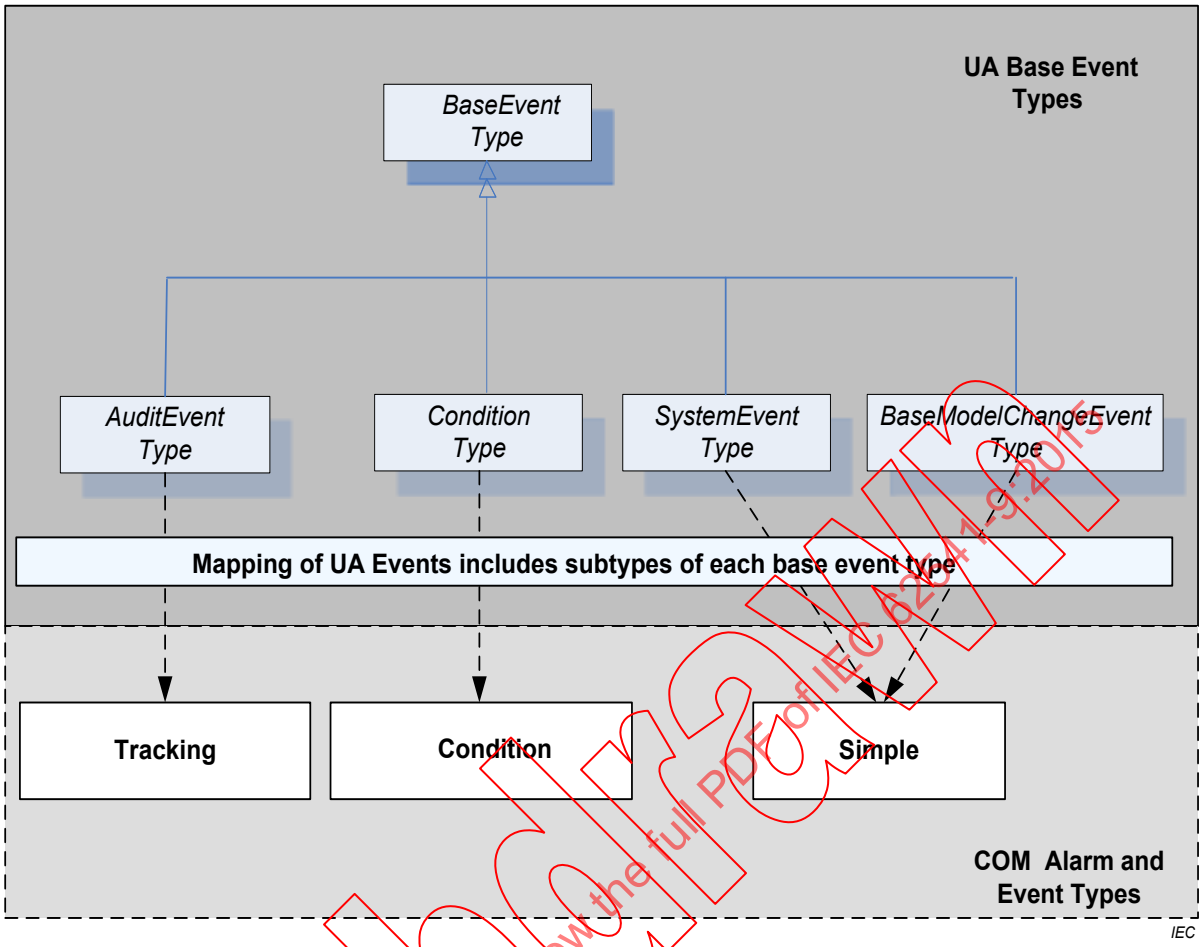


Figure D.2 – Mapping UA Event Types to COM A&E Event Types

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.

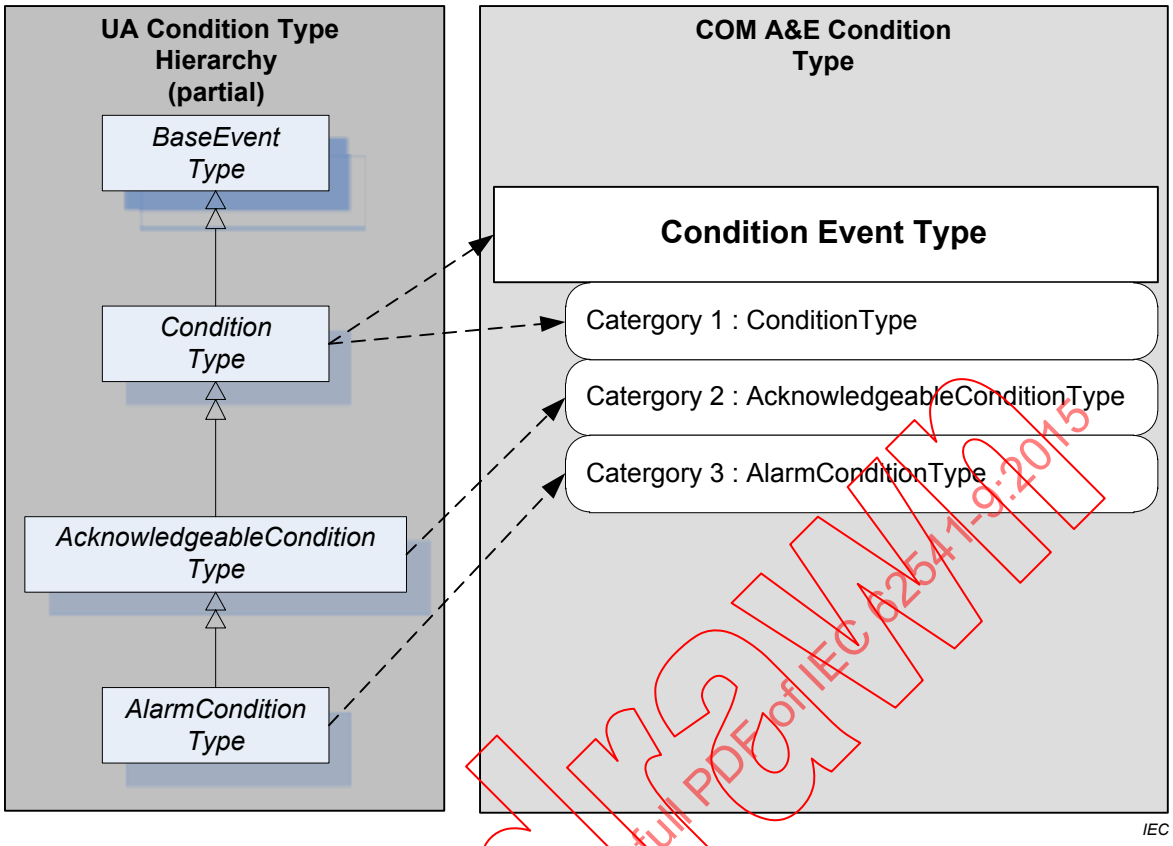


Figure D.3 – Example Mapping of UA Event Types to COM A&E Categories

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 *Attributes* 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 Notifications* 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 *Attributes* 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): <u>Bad status codes</u> 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 <u>Uncertain status codes</u> 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 <u>Good status codes</u> 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.
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 monitored item 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 monitored item – 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 monitored items 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.

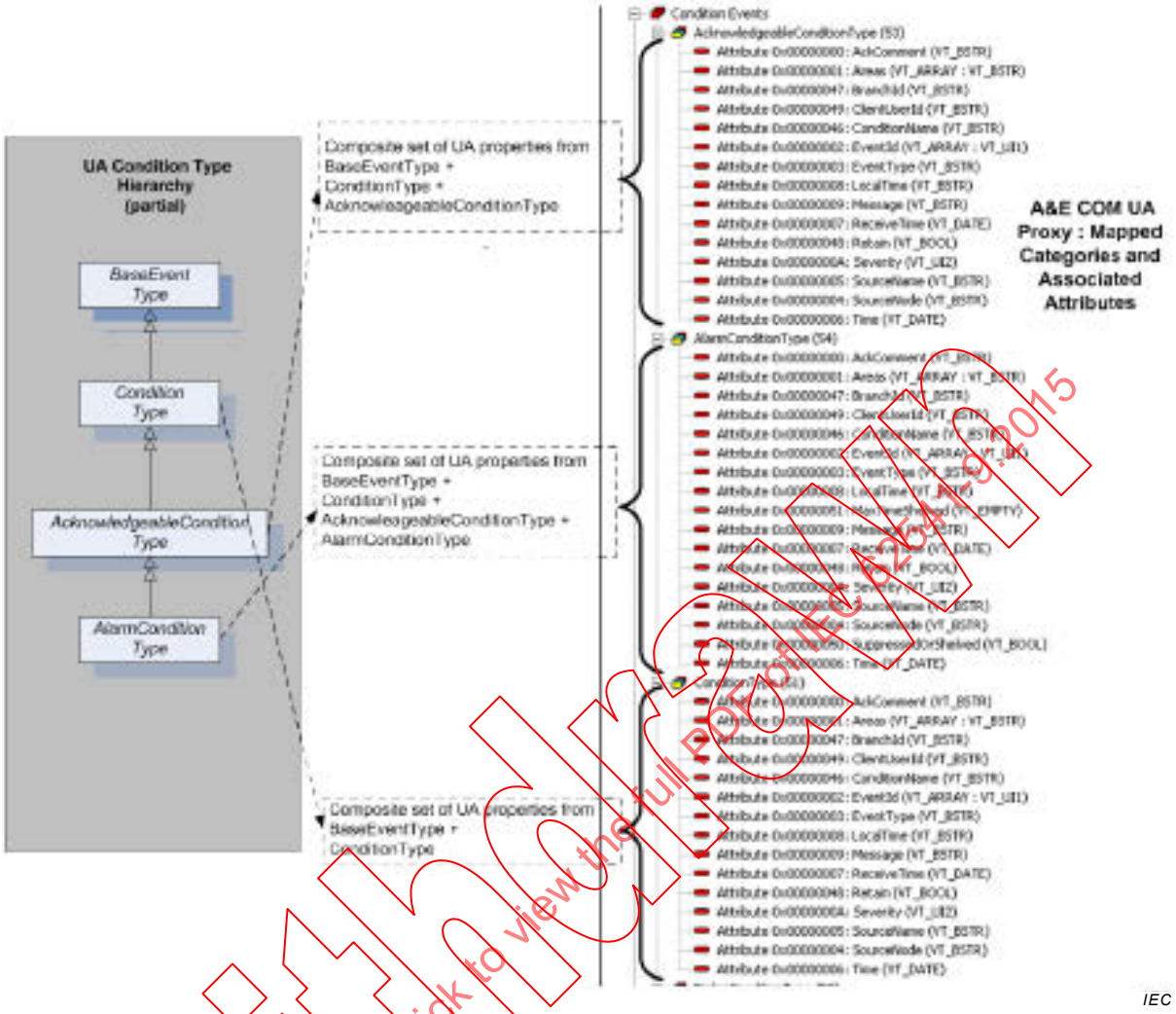


Figure D.4 – Example Mapping of UA Event Types to A&E Categories with Attributes

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* hierarchy is the *Server Object*. Starting at the *Server Object*, eventNotifier *Reference*s 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 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 below.

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 monitored items 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 monitored item, 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 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 monitored items 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 monitored items 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 monitored item corresponding to the area name is decremented. When the ref count goes to zero, the monitored item is removed from the UA *Subscription*

As with source filter strings, area filter strings can 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 monitored item 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 monitored items 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 monitored item 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 monitored items to the UA *Subscription*.

Bibliography

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

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

SOMMAIRE

COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

_____

ARCHITECTURE UNIFIEE 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 deuxième édition annule et remplace la première édition parue en 2012. Cette édition constitue une révision technique.

Cette édition inclut les modifications techniques majeures suivantes par rapport à l'édition précédente:

a) ajout d'une section pour décrire le comportement attendu pour les serveurs A&C et le modèle d'informations associé en cas de redondance ou de défauts de communication, voir 5.14 pour des détails supplémentaires [réf 698 & 967];

b) modification du DialogConditionType afin qu'il ne soit pas abstrait, étant donné qu'il est prévu qu'une instance de ce type existe dans le système, voir Tableau 19 pour des détails supplémentaires [réf 1622];

c) mise à jour de la méthode ConditionRefresh pour permettre l'utilisation des NodeIds bien connus associés aux types pour MethodId et ConditionId au lieu d'exiger l'utilisation de MethodId et ConditionId uniquement, qui font partie d'une instance. Sans cette modification, les serveurs qui ne présentent pas d'instance peuvent avoir des problèmes avecConditionRefresh, voir 5.5.7 pour des détails supplémentaires [réf 2091];

d) ExclusiveLimitStateMachineType et ShelvedStatemachineType déclarés être des sous-types de FinitStateMachineType, et non de StateMachineType. Voir 5.8.3 et 5.8.5.2 pour des détails supplémentaires [réf 2091].

Le texte de cette norme est issu des documents suivants:

CDV	Rapport de vote
65E/382/CDV	65E/408/RVC

Le rapport de vote indiqué dans le tableau ci-dessus donne toute information sur le vote ayant abouti à l'approbation de cette norme.

Cette publication a été rédigée selon les Directives ISO/IEC, Partie 2.

Une liste de toutes les parties de la série IEC 62541, publiées sous le titre général *Architecture unifiée OPC*, peut être consultée sur le site web de l'IEC.

Le comité a décidé que le contenu de cette publication 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 à la publication recherchée. À cette date, la publication sera

- reconduite,
- supprimée,
- remplacée par une édition révisée, ou
- amendée.

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 UNIFIEE OPC –

Partie 9: Alarmes et conditions

1 Domaine d'application

La présente partie de l'IEC 62541 spécifie la représentation des *Alarms* (alarmes) et des *Conditions* dans l'architecture unifiée OPC, y compris la représentation du *Information Model (*modèle d'informations) relative aux *Alarms* et *Conditions* dans l'espace d'adresses d'OPC UA.

2 Références normatives

Les documents suivants sont cités en référence de manière normative, en intégralité ou en partie, dans le présent document et sont indispensables pour son application. 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 de l'espace d'adressage*

IEC 62541-4, *Architecture unifiée OPC – Partie 4: Services*

IEC 62541-5, *Architecture unifiée OPC – Partie 5: Modèle d'information*

IEC 62541-6, *Architecture unifiée OPC – Partie 6: Correspondances*

IEC 62541-8, *Architecture unifiée OPC – Part 8: Accès aux données*

EEMUA: 2nd Edition EEMUA 191 – *Alarm System – A guide to design, management and procurement (Appendixes 6, 7, 8, 9)*, disponible à http://www.eemua.co.uk/ (disponible en anglais seulement)

3 Termes, définitions et abréviations

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 et l'IEC 62541-5, ainsi que les suivants s'appliquent.

3.1.1
acquittement
acknowledge
action de l'*Opérateur* qui indique la reconnaissance d'une nouvelle *Alarm*

Note 1 à l'article: Cette définition est tirée de l'EEMUA. Le terme "Accept'' (''Accepter'') est un autre terme courant utilisé pour décrire l'acquittement (*Acknowledge*). Les deux termes sont interchangeables. La présente norme utilise "*Acknowledge*".

3.1.2
actif
active
état d'une *Alarm* qui indique que la situation que l'*Alarm* représente existe actuellement

Note 1 à l'article: D'autres termes courants définis par l'EEMUA sont “Standing” (en cours) pour une *Active Alarm* (Alarme active) et "Cleared” (effacée) lorsque la *Condition* est retournée à la normale et n'est plus *active*.

3.1.3
classe de condition
ConditionClass
ensemble de *Condition* qui indique le domaine ou le but pour lequel une certaine *Condition* est utilisée

Note 1 à l'article: Un certain nombre de *ConditionClasses* (classes de conditions) de haut-niveau sont définies dans la présente spécification. Les fournisseurs ou les organisations peuvent dériver des classes plus concrètes ou définir des classes de haut niveau différentes.

3.1.4
branche de condition
ConditionBranch
état spécifique d'une *Condition*

Note 1 à l'article: Le *Server* (serveur) peut maintenir des *ConditionBranches* (branches de conditions) pour l'état courant et aussi pour des états antérieurs.

3.1.5
source de condition
ConditionSource
élément sur lequel repose une *Condition* spécifique ou auquel celle-ci se rapporte

Note 1 à l'article: Typiquement, il s'agira d'une *Variable* représentant un marqueur de processus (par exemple FIC101) ou d'un *Object* (objet) représentant un dispositif ou un sous-système.

Dans *des Events (Événements)* générés pour des *Conditions, la SourceNode Property (Propriété SourceNode)* (héritée du BaseEventType) contient le *NodeId (identificateur de nœud)* de la *ConditionSource* (source de condition).

3.1.6
confirmer
confirm
action de l'*opérateur* informant le *Server* (*Serveur*) qu'une action corrective a été entreprise pour traiter la cause de l'*Alarm*

3.1.7
désactiver
disable
système qui est configuré de telle manière que l'*Alarm* ne sera pas générée même si l'*Alarm Condition* (*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.

3.1.8
opérateur
operator
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 la présente norme, un Opérateur est un utilisateur spécial. Toutes les descriptions qui s'appliquent aux utilisateurs généraux s'appliquent aussi aux Opérateurs.

3.1.9
rafraîchir; rafraîchissement
refresh
action de mise à jour d'un *Event Subscription (Abonnement d'événement*) qui fournit toutes les *Alarms* qui sont considérées devoir être *Retained* (retenues)

Note 1 à l'article: Cette notion est décrite plus en détail dans l'EEMUA.

3.1.10
retenir
retain
Alarm dans un état qui est intéressant pour un *Client* qui souhaite synchroniser son état de *Conditions* à l'état du *Server*

3.1.11
suspension
shelving
moyen par lequel l'*Operator* est capable d'empêcher temporairement qu'une *Alarm* ne s'affiche à l'attention de l'*Operator* lorsqu'elle cause une gêne pour l'*Operator*

Note 1 à l'article: Une Shelved *Alarm* (alarme suspendue) sera retirée de la liste et ne sera pas annoncée de nouveau tant qu'elle ne sera pas un-shelved (reprise). Cette définition est tirée de l'EEMUA.

3.1.12
supprimer; suppression
suppress
action de déterminer qu'il convient qu'une *Alarm* ne se produise pas

Note 1 à l'article: "Une *Alarm* est supprimée lorsque des critères logiques sont appliqués pour déterminer qu'il convient que l'*Alarm* ne se produise pas, même si l'*Alarm Condition* de base (valeur de consigne d'*Alarm* dépassée par exemple) est présente" selon la définition de l'EEMUA.

3.2 Abréviations et symboles

A&E Alarm&Event (comme utilisé pour 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 ci-après décrivent les types de données qui sont utilisés dans tout la présente norme. 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 paramètre définis dans l'IEC 62541-3

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

Tableau 2 – Types de paramètres définis dans l'IEC 62541-4

Type de paramètre
IntegerId
StatusCode

4 Concepts

4.1 Généralités

La présente norme définit un *Information Model* (modèle d'informations) pour les *Conditions*, les *Conditions* de dialogue et les *Alarms*, y compris les fonctionnalités d'acquittement. Ce modèle s'appuie sur la gestion d'événements de base définie dans l'IEC 62541-3, IEC 62541-4 et l'IEC 62541-5 et les complète. Cet *Information Model* peut aussi être étendu pour prendre en charge les autres besoins de domaines spécifiques. Les détails des aspects du modèle d'informations pris en charge sont fournis via les profils (voir IEC 62541-7). Les systèmes sont censés fournir les événements et conditions historiques via le cadre d'accès aux historiques (Historical Access framework) normalisé (voir IEC 62541-11).

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:

- température dépassant une limite configurée;
- dispositif ayant besoin de maintenance;
- processus par lots qui exige que l'utilisateur confirme une certaine étape du processus avant de se poursuivre.

Chaque instance *Condition* est d'un *ConditionType* (type de condition) spécifique. Le *ConditionType* et les types dérivés sont des sous-types de *BaseEventType* (Type d'événement de base) (voir IEC 62541-3 et IEC 62541-5). La présente partie définit les types qui sont communs à de nombreuses industries. Il est prévu que les fournisseurs ou autres groupes de normalisation définissent des *ConditionTypes* supplémentaires dérivés des types de base communs définis dans la présente partie. Les *ConditionTypes* pris en charge par un *Server* sont présentés dans l'*AddressSpace* (*espace d'adresses*) du *Server*.

Les instances de *Condition* sont des mises en œuvre spécifiques d'un *ConditionType*. Il incombe au *Server* de décider si, oui ou non, ces instances sont également présentées dans l'*AddressSpace* du *Server*. Les spécifications de 4.10 fournissent un contexte supplémentaire concernant les instances de *Condition*. Ces instances de *Condition* doivent avoir un

identificateur unique pour les distinguer des autres instances, et ce, 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 nécessitant encore une attention sont également tenus d'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 (identificateur de branche) qui la différencie des autres branches de la même instance *Condition*. La *ConditionBranch* qui représente l'état courant de la *Condition* (le tronc) a un BranchId Null (vide). Les *servers* peuvent générer des *Event Notifications* (*Notifications d'événements)* pour chaque branche. Lorsque l'état représenté par une *ConditionBranch* ne nécessite pas d'attention supplémentaire, une *Event Notification* finale pour cette branche aura sa *Retain Property* définie sur False (Faux). Les spécifications de 4.4 fournissent plus d'informations et des cas d'utilisation. La conservation d'états antérieurs et, donc, également la prise en charge de plusieurs branches sont facultatives pour les *Servers*.

D'un point de vue conceptuel, la durée de vie de l'instance *Condition* est indépendante de son état. Cependant, des *Servers* peuvent fournir l'accès à des instances *Condition* uniquement pendant le temps que les *ConditionBranches* existent.

Le modèle d'état de base pour *Condition* est illustré à la Figure 1. Il est étendu par les divers sous-types de *Condition* définis dans la présente norme et peut être étendu encore plus par les fournisseurs ou autres groupes de normalisation. Les états principaux d'une *Condition* sont "*disabled*" (*désactivé*) et "*enabled*" (*activé*). L'état "*disabled*" est destiné à permettre de désactiver des *Conditions* au niveau du *Server* ou en dessous du *Server* (dans un dispositif ou un certain système sous-jacent). L'état "*enabled*" est normalement étendu par l'addition de sous-états.

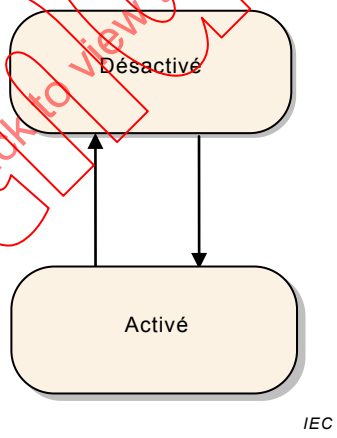


Figure 1 – Modèle d'état de base pour Condition

Un passage à l'état "*Disabled*" se traduit par un *Condition Event (événement de condition*), mais il n'est pas généré de *Event Notifications* consécutives tant que la *Condition* n'est pas retournée à l'état "Enabled".

Lorsqu'une *Condition* entre dans l'état "Enabled", ce passage et tous les passages consécutifs se traduisent par la création de *Condition Events* (*événements de condition*) par le *Server*.

Lorsqu'un *Auditing* (*Conduite d'audit*) est pris en charge par un *Server*, l'action suivante liée à un *Auditing* doit être réalisée. Le *Server* crée des *AuditEvents* (*événements d'audit*) pour les opérations "enable" (activer) et "disable" (désactiver) (soit par invocation d'une *Method* (*Méthode*) ou par certains moyens spécifiques au serveur/fournisseur), plutôt que de créer une *AuditEvent Notification* (notification d'événements d'audit) pour chaque instance

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* (conditions acquittables) sont des sous-types du *ConditionType* de base. Les *AcknowledgeableConditions* présentent les états pour indiquer qu'une *Condition* est à acquitter ou à confirmer.

Un *AckedState* (état acquitté) et un *ConfirmedState* (état confirmé) étendent l'état "*Enabled*" défini par la *Condition*. Le modèle d'état est illustré à la Figure 2. L'état activé est étendu en ajoutant l'*AckedState* et (facultativement) le *ConfirmedState*.

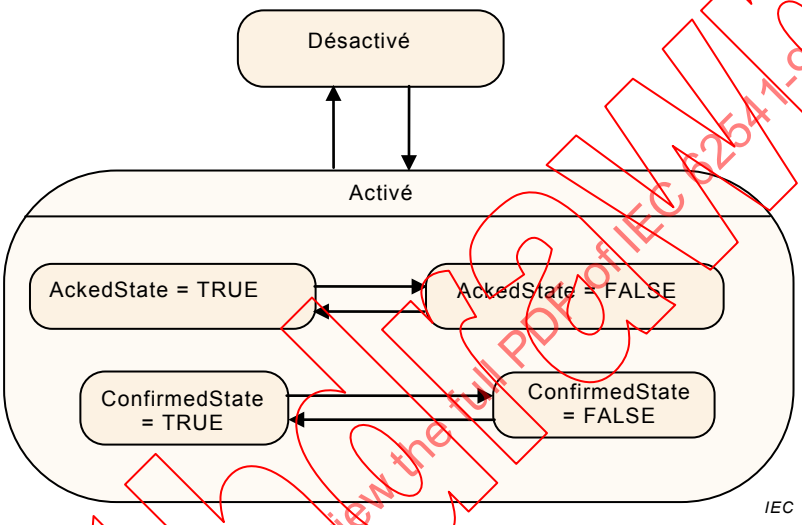


Figure 2 – Modèle d'état pour AcknowledgeableConditions

L'acquittement de la transition peut venir du *Client* ou peut être dû à une certaine logique interne au *Server*. Par exemple, l'acquittement d'une *Condition* connexe peut faire que cette *Condition* devienne acquittée ou la *Condition* peut être réglée pour s'acquitter automatiquement elle-même lorsque la situation acquittable disparaît.

Deux modèles d'états pour *Acknowledge* sont pris en charge dans la présente norme. 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 pour *Acknowledge* est illustré 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'*Alarm* 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 prévoir des changements vers AckedState lorsque la *Condition* passe à un état inactif.

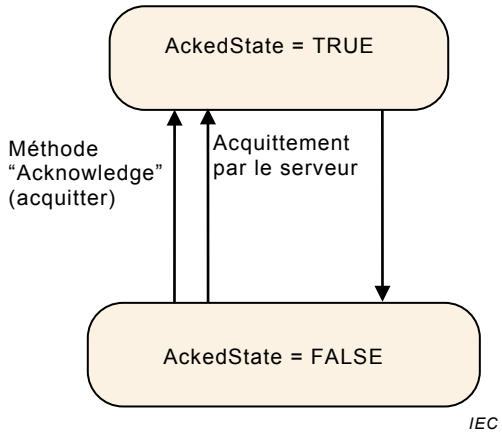


Figure 3 – Modèle d’état pour Acknowledge

Un modèle d’état plus complexe qui ajoute une confirmation à l'*Acknowledge* de base est illustré à la Figure 4. Le modèle *Confirmed Acknowledge* (*acquittement confirmé*) est typiquement utilisé pour établir une différence entre le fait d'acquitter la présence d'une *Condition* et le fait d'avoir entrepris une certaine action pour traiter la *Condition*. Par exemple, un *Operator* recevant une *Notification* d’une température haute pour le moteur appelle la *Acknowledge Method (*méthode "acquitter") pour signaler au *Server* qu’une température haute a été observée. L'*Operator* entreprend ensuite une certaine action, telle que diminuer la charge sur le moteur afin de faire baisser la température. L'*Operator* appelle ensuite la *Confirm Method (*méthode "confirmer") pour signaler au *Server* qu'une action corrective a été entreprise.

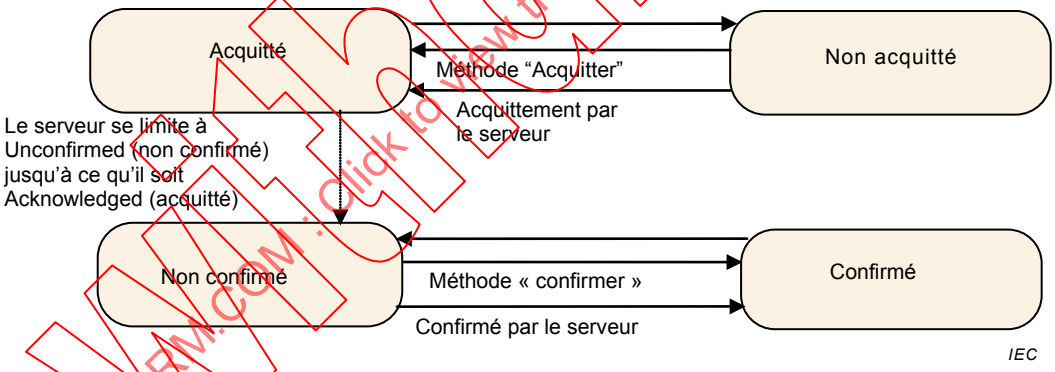


Figure 4 – Modèle d’état pour Confirmed Acknowledge (acquittement confirmé)

4.4 États 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, il est exigé d'acquitter tant le passage à l'état *Active* que le passage à l'état inactif. 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 *Server* est tenu de maintenir les *ConditionBranches* pour tous les états antérieurs non acquittés. Ces branches seront supprimées dès qu'elles auront été acquittées ou si elles ont atteint leur état final.

Les *ConditionBranches* multiples peuvent être également 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'est abonné à des *Events*, la *Notification* des transitions commence à l'heure de *Subscription (abonnement)*. L'état courant existant n'est pas consigné. Cela signifie par exemple que des *Clients* ne sont pas informés des *Alarms* actuellement actives jusqu'à ce qu'un nouveau changement d'état se produise.

Les *Clients* peuvent obtenir l'état courant de toutes les instances *Condition* qui sont dans un état intéressant en demandant un *Refresh* pour un *Subscription*. Il convient de noter que le *Refresh* n'est pas un moyen systématique de rejouer car le *Server* n'est pas tenu de maintenir un historique des *Event*.

Les *Clients* demandent un *Refresh* (*rafraîchissement*) en appelant la *Method* "ConditionRefresh" (*Méthode* ‘’Rafraîchissement de condition’’). Le *Server* répond par un *RefreshStartEvent*. Cet *Event* est suivi des *Retained Conditions* (*conditions retenues*). Le *Server* peut aussi envoyer de nouvelles *Event Notifications* (notifications d'événements) parsemées d'*Event Notifications* de rafraîchissement. Lorsque le *Server* en a terminé avec le rafraîchissement, un *RefreshEndEvent* est produit et marque la fin du rafraîchissement. Les *Clients* doivent vérifier plusieurs *Event Notifications* pour détecter une *ConditionBranch* afin d'éviter d'écraser un nouvel état délivré avec 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. Ceci permet à un *Client* de représenter avec précision l'état courant de *Condition*.

Un *Client* souhaitant afficher le statut courant des *Alarms* et des *Conditions* (appelé “current *Alarm* display”, c'est-à-dire affichage des alarmes courantes) utiliserait la logique suivante pour traiter les *Refresh Event Notifications* (notifications d'événements de rafraîchissement). À la réception de l'*Event* du *RefreshStartEvent*, le *Client* marque toutes les *Retained Conditions* comme suspectes. Le *Client* ajoute tous les éventuels nouveaux *Events* qui ont été reçus pendant le rafraîchissement sans les marquer comme suspects. Le *Client* enlève également le fanion "suspect" de toutes les éventuelles *Retained Conditions* qui sont retournées comme partie intégrante du rafraîchissement. Lorsque le *Client* reçoit un *RefreshEndEvent*, le *Client* retire tous les éventuels *Events* 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 *Servers*, en particulier ceux exigeant l'acquittement des états antérieurs, maintiendront des *ConditionBranches* distinctes pour les états antérieurs qui exigent encore une attention.

  ConditionRefresh doit produire des *Event Notifications* pour tous les états intéressants (courants et antérieurs) d'une instance *Condition* et les *Clients* peuvent par conséquent recevoir plus d'un *Event* pour une même instance *Condition* avec des *BranchIds* (identificateurs de branche) différents.

- Dans certaines circonstances, un *Server* peut ne pas être capable d'assurer que le *Client* est totalement synchronisé avec l'état courant des instances *Condition*. Par exemple, si le système sous-jacent représenté par le *Server* est réinitialisé ou les communications sont perdues pendant un certain temps, le *Server* peut avoir besoin de se resynchroniser au système sous-jacent. Dans ces cas, le *Server* doit envoyer un *Event* de type *RefreshRequiredEventType* pour annoncer au *Client* qu'un *Refresh* peut être nécessaire. Il convient qu'un *Client* recevant cet *Event* spécial déclenche un *ConditionRefresh* comme indiqué dans le présent article.

- Afin de s'assurer qu'un *Client* est toujours informé, les trois *EventTypes* spéciaux (*RefreshEndEventType, RefreshStartEventType* et *RefreshRequiredEventType*) ignorent le filtrage de contenu d'*Event* associé à un *Subscription* et sont toujours délivrés au *Client*.

4.6 Sévérité, qualité et commentaire

Comment (commentaire), Severity (sévérité) et Quality (qualité) sont des éléments importants de *Conditions* et tout changement qui leur est apporté engendre des *Event Notifications*.

La "Severity" (sévérité) d'une *Condition* est héritée du modèle *Event* 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 *Alarms* appartenant à la *ProcessConditionClass*.

Un "Comment" (commentaire) est une chaîne créée par l'utilisateur qui est à associer à un certain état d'une *Condition*.

''Quality''(qualité) se réfère à la qualité de la ou des valeurs de données sur lesquelles cette *Condition* est basée. Étant donné qu'une *Condition* est habituellement basée 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* "LevelAlarm" est également "questionable" (douteuse). Lorsque deux variables ou plus sont représentées par une condition donnée ou si la condition provient d'un système sous-jacent, et lorsqu'aucune correspondance directe avec une variable est disponible, c'est à l'application de déterminer quel niveau de qualité est affiché comme partie intégrante de la condition.

4.7 Dialogues

Les Dialogs (dialogues) sont des *ConditionTypes* utilisés par un *Server* pour demander des données d'utilisateur. Ils sont typiquement utilisés lorsqu'un *Server* est entré dans un état qui exige l'intervention d'un *Client*. Par exemple, un *Server* surveillant une machine à papier indique qu'un rouleau de papier a été enroulé et est prêt à l'inspection. Le *Server* activerait une *Condition* de Dialog indiquant à l'utilisateur qu'une inspection est exigée. Une fois que l'inspection a eu lieu, l'utilisateur répond en informant le *Server* d'une inspection acceptée ou non acceptée permettant au processus de se poursuivre.

4.8 Alarmes

Les *Alarms* sont des spécialisations des *AcknowledgeableConditions* qui ajoutent à une *Condition* des concepts d'état *Active (actif)*, *Shelving* (*suspension*) et *Suppressed* (supprimé*)*. Le modèle d'état est illustré à la Figure 5.

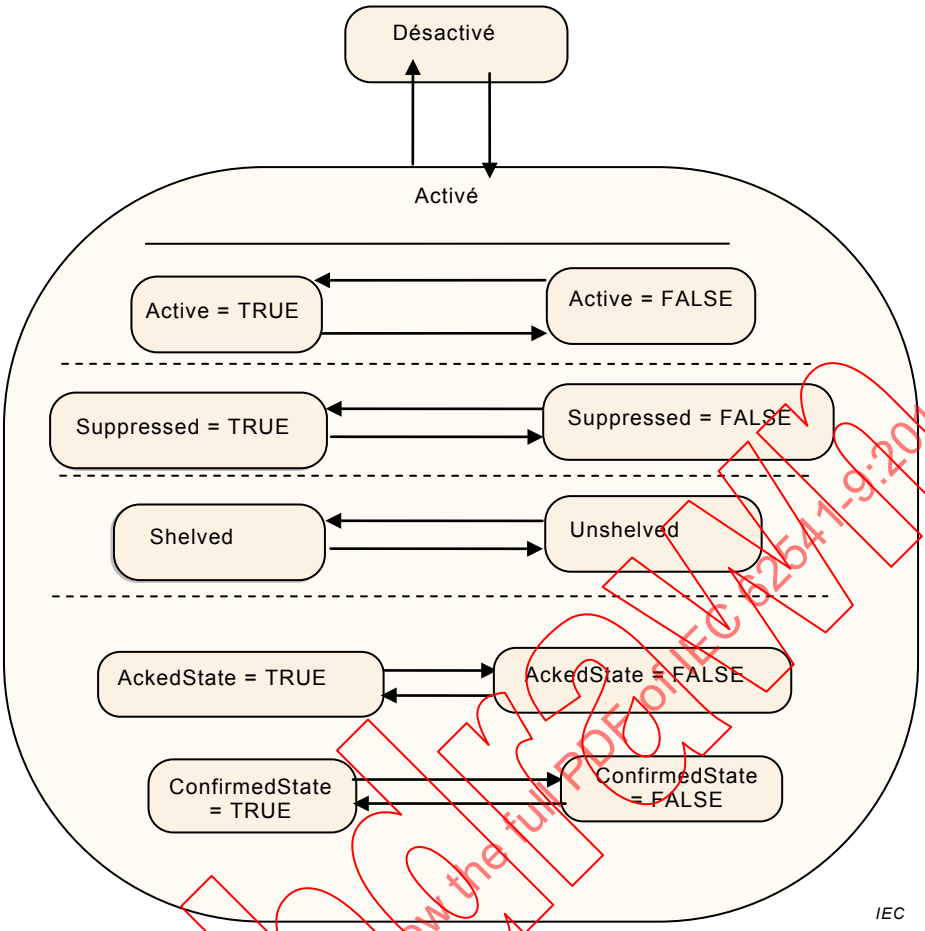


Figure 5 – Modèle de diagramme d'états pour les alarmes

Une *Alarm* dans l'état *Active* indique que la situation décrite par la *Condition* est actuellement présente. Lorsqu'une *Alarm* est dans l'état inactif, elle indique une situation qui est retournée à un état normal.

Certains sous-types d'*Alarm* introduisent des sous-états de l'état *Active*. Par exemple, une *Alarm* représentant une température peut fournir aussi bien un état de niveau haut qu'un état de niveau haut critique (voir l'article suivant).

L'état *Shelving* (*suspension*) peut être établi par un *Operator* via des *Methods* d'OPC UA. L'état *Suppressed* (supprimé) est établi en interne par le *Server* pour des raisons spécifiques au système. Les systèmes d'*Alarm* mettent typiquement en œuvre les caractéristiques *Suppress* (supprimer) et Shelve (suspendre) pour éviter que les *Operators* ne soient submergés par des flots d'*Alarms* en limitant le nombre d'*Alarms* qu'un *Operator* voit sur un affichage d'*Alarms* courantes. Cela est accompli par le réglage du fanion *SuppressedOrShelved* sur des *Alarms* dépendantes de second ordre et/ou des *Alarms* de moindre sévérité, amenant l'*Operator* à se concentrer sur les questions les plus critiques.

Les états *Shelved* et *Suppressed* diffèrent de l'état *Disabled* parce que les *Alarms* sont encore totalement fonctionnelles et peuvent être incluses dans des *Subscription Notifications* (*notifications d'abonnement*) adressées à un *Client*.

4.9 Etats actifs multiples

Dans certains cas, il est souhaitable de définir plus complètement l'état *Active* d'une *Alarm* en fournissant un diagramme de sous-états pour l'état *Active*. Par exemple, une *Alarm* 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 illustré à la Figure 6.

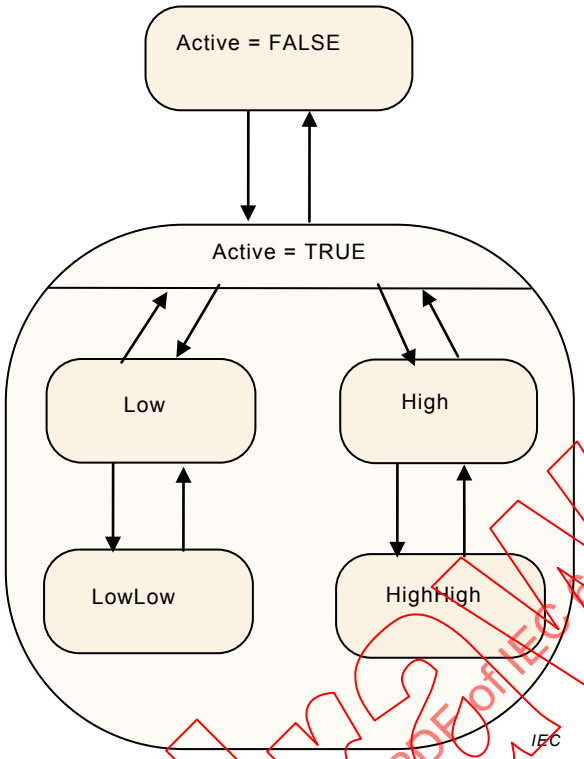


Figure 6 – Exemple d'états actifs multiples

Avec le modèle d'*Alarm* à états multiples, les transitions d'états parmi les sous-états de *Active* sont autorisées 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'*Alarm* ne peut être que dans un seul sous-état à la fois. Si, par exemple, la température dépasse la limite HighHigh, l'alarme LevelAlarm exclusive associée est dans le sous-état HighHigh et non dans le sous-état High.

Certains systèmes d'*Alarm* 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 LevelAlarm non exclusive est à la fois dans le sous-état High et dans le sous-état HighHigh.

4.10 Instances *Condition* dans l'espace d'adresses

Sachant que les *Conditions* ont toujours un état (*Enabled* ou *Disabled*) et éventuellement plusieurs sous-états, cela a un sens d'avoir des instances *Conditions* présentes dans l'*AddressSpace*. Si le *Server* présente des instances *Condition*, celles-ci apparaissent habituellement dans l'*AddressSpace* comme étant des composants des *Objects* qui les "possèdent". Par exemple, un transmetteur de température qui comporte une *Alarm* haute température intégrée apparaîtrait dans l'*AddressSpace* comme une instance d'un certain *Object* transmetteur de température avec une *Référence HasComponent* (possède un composant) à une instance d'un *LevelAlarmType*.

La disponibilité d'instances permet à des *Clients* d'accès aux données de surveiller l'état courant de la *Condition* en s'abonnant aux valeurs d'*Attribute* (c'est-à-dire attribut) des *Nodes Variable*.

Alors que le fait de présenter des instances *Condition* dans l'*AddressSpace* n'est pas toujours possible, le faire permet l'interaction directe (lecture, écriture et invocation de *Method*) avec

une instance spécifique *Condition*. Par exemple, si une instance *Condition* n'est pas présentée, il n'y a aucun moyen d'invoquer la *méthode Enable* ou *Disable* pour l'instance spécifique *Condition*.

4.11 Conduite d'audits pour les alarmes et les conditions

Les normes d'OPC UA incluent 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 relatives au “Who”, “When” and “What” (“Qui”, “Quand” et “Quoi”) concernant les interactions utilisateur avec un système. Ces enregistrements d'audit sont particulièrement importants lorsque la gestion des *Alarms* est envisagée. Les *Alarms* constituent l'instrument type pour fournir des informations à un utilisateur et lui indiquer que quelque chose requiert 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 *Method* qui ont une incidence sur l'état du système, par exemple un appel de la méthode *Acknowledge* générerait un événement *AuditConditionAck*.

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, la présente norme définit un certain nombre de sous-types des *AuditEventTypes,* mais sans leur ajouter de nouveaux champs.

La présente norme décrit l'*AuditEventType* que chaque *Method* est tenue de générer. Par exemple, la méthode *Disable* comporte une *référence AlwaysGeneratesEvent* à un *AuditConditionEnableEventType*. Un *Event* de ce type doit être généré pour chaque invocation de la *Method*. L'*Event* d'audit décrit l'interaction de l'utilisateur avec le système; dans certains cas, cette interaction peut affecter plus d'une *Condition* ou être liée à plus d'un état.

5 Modèle

5.1 Généralités

Le modèle d'*Alarm* et de *Condition* étend le modèle *Event* de base d'OPC UA en définissant divers *Event Types* (types d'événements) basés sur le *BaseEventType* (type d'événement de base). Tous les *Event Types* définis dans la présente norme peuvent être étendus encore plus pour former des *Types* d'*Alarm* et de *Condition* spécifiques au domaine ou au *Server*.

Des instances des *Types* d'*Alarm* et de *Condition* peuvent être facultativement présentées dans l'*AddressSpace* afin de permettre un accès direct à l'état d'une *Alarm* ou d'une *Condition*.

Les paragraphes suivants définissent les *Types* d'*Alarm* et de *Condition* selon OPC UA. La Figure 7 décrit de manière informelle la hiérarchie de ces *Types*. Les sous-types de *LimitAlarmType* et de *DiscreteAlarmType* n'y sont pas montrés. La hiérarchie complète de l'*AlarmConditionType* peut être vue à la Figure 11.

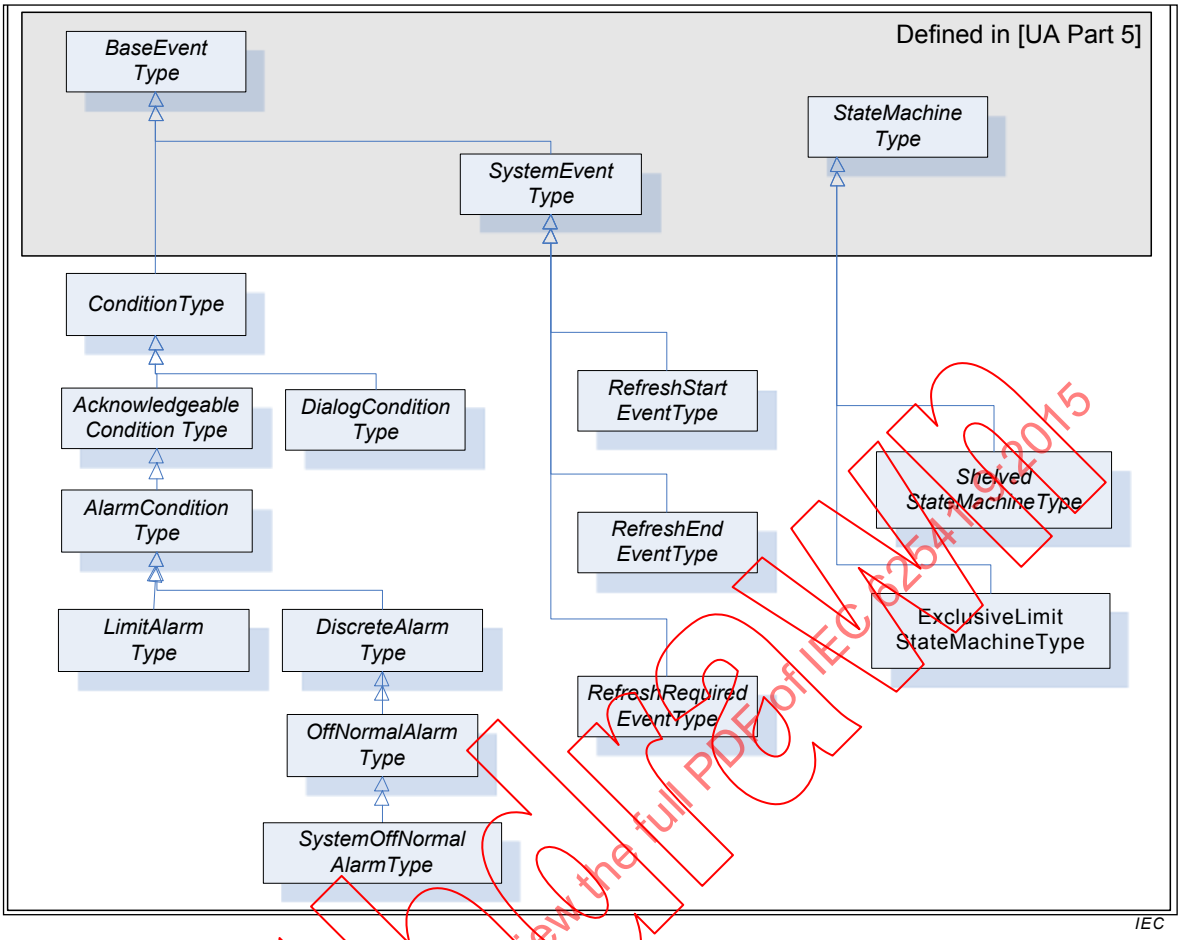


Figure 7 – Hiérarchie de ConditionType

5.2 Diagrammes d'états à deux états

La plupart des états définis dans la présente norme sont simples à connaître, ils sont soit TRUE (vrais), soit FALSE (faux). Le *TwoStateVariableType* (type de variable à deux états) est introduit spécialement pour ce cas d'utilisation. Des états plus complexes sont modélisés en utilisant un *StateMachineType* (type de diagramme d'états) défini dans l'IEC 62541-5.

Le *TwoStateVariableType* est dérivé du *StateVariableType* (type de variable à états) défini dans l'IEC 62541-5 et défini de manière formelle au Tableau 3.

Tableau 3 – Définition de TwoStateVariableType

Attribut	Valeur				
BrowseName	TwoStateVariableType				
DataType	LocalizedText (Texte Localisé)				
ValueRank	-1 (-1 = Scalar)				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
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	Boolean	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 Value* d'une *TwoStateVariable* (*variable à deux états*) contient l'état courant sous la forme d'un nom lisible sans aide. 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 neutralisé pour refléter le *DataType* (Booléen) exigé. La valeur doit être l'état courant, à savoir soit TRUE, soit FALSE.

TransitionTime (*heure de transition*) spécifie l'heure de l'entrée dans l'état courant.

EffectiveTransitionTime (*heure de transition effective*) 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 – pendant qu'elle est active– commute plusieurs fois entre High et HighHigh, l'heure *TransitionTime* reste à l'instant auquel l'*Alarm* devenait active alors que l'heure *EffectiveTransitionTime* change avec chaque changement d'un sous-état.

La *Property (propriété)* facultative *EffectiveDisplayName* issue du *StateVariableType* est utilisée si un état comporte des sous-états. Elle contient un nom lisible sans aide pour l'état courant après prise en compte de l'état de tous les éventuels *SubStateMachines*. À 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 *Properties (propriétés)* facultatives du *StateVariableType* n'ont pas de signification définie pour *TwoStateVariables* (*variables à deux états*).

TrueState et *FalseState* contiennent la chaîne localisée pour la valeur de *TwoStateVariable* lorsque sa *Id Property* (*propriété Id)* a respectivement la valeur TRUE ou FALSE. Sachant que les deux *Properties* fournissent des métadonnées pour le *Type*, les *Servers* peuvent ne pas permettre de sélectionner ces *Properties* dans le filtre d'événements pour un élément surveillé. Les *Clients* peuvent utiliser le *Service* Read (lecture) pour récupérer des informations à partir du *ConditionType* spécifique.

Une *Référence HasTrueSubState* (*a des sous-états de vrai*) est utilisée pour indiquer que l'état TRUE a des sous-états.

Une *Référence HasFalseSubState* (*a des sous-états de faux*) est utilisée pour indiquer que l'état FALSE a des sous-états.

5.3 Variables de Condition

Les divers éléments d'information d'une *Condition* ne sont pas considérés comme étant des états. Cependant, une modification de leur valeur est considérée importante et supposée déclencher une *Event Notification*. Ces éléments d'information sont appelés *ConditionVariables* (*variables de condition*).

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**	**Règles de modélisation**
Sous-type du *BaseDataVariableType* défini dans l'IEC 62541-5.					
HasProperty	Variable	SourceTimestamp	UtcTime	PropertyType	Obligatoire

Le *SourceTimestamp* (*horodatage de source*) indique l'heure du dernier changement de la *Value* de cette *ConditionVariable*. Il doit s'agir de la même heure qui serait retournée du *Service Read* à l'intérieur de la structure *DataValue* pour l'attribut *Value* de la *ConditionVariable.*

5.4 Types de référence des sous-états

5.4.1 Généralités

Cet article définit les *ReferenceTypes* qui sont nécessaires au-delà de ceux déjà spécifiés dans le cadre de l'IEC 62541-3 et de l'IEC 62541-5 pour étendre les diagrammes d'états *TwoState* (à deux états) avec des sous-états. Ces *References* n'existent que lorsque des sous-états sont disponibles. Par exemple, si l'état d'un diagramme TwoState est FALSE, les sous-états dont la référence est l'état TRUE ne sont alors pas disponibles. Lorsqu'un événement 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 *TwoStateVariables* peuvent être étendues 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* (références non hiérarchiques).

La sémantique indique que le sous-état (le *Node* 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 *References HasTrueSubState* existent pour le même super état), ils sont tous traités comme des sous-états indépendants. La représentation dans l'*AddressSpace* est spécifiée au Tableau 5.

Le *SourceNode* (*nœud source*) de la *Reference* doit être une instance d'un *TwoStateVariableType* et le *TargetNode* (*nœud cible*) doit soit être une instance d'un *TwoStateVariableType*, soit une instance d'un sous-type d'un *StateMachineType*.

Il n'est pas exigé de fournir la *Reference HasTrueSubState* d'un super état à un sous-état, mais il est exigé que le sous-état fournisse la *Reference* inverse (IsTrueSubStateOf, c'est-à-dire "est sous-état de vrai de") 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* (références non hiérarchiques).

La sémantique indique que le sous-état (le *Node* 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 *References HasFalseSubState* existent pour le même super état), ils sont tous traités comme des sous-états indépendants. La représentation dans l'*AddressSpace* est spécifiée au Tableau 6.

Le *SourceNode* (*nœud source*) de la *Reference* doit être une instance d'un *TwoStateVariableType* et le *TargetNode* (*nœud cible*) doit soit être une instance d'un *TwoStateVariableType*, soit une instance d'un sous-type d'un *StateMachineType*.

Il n'est pas exigé de fournir la *Reference HasFalseSubState* d'un super état à un sous-état, mais il est exigé que le sous-état fournisse la *Reference* inverse (IsFalseSubStateOf, c'est-à-dire "est sous-état de faux de") à son super état.

Tableau 6 – ReferenceType HasFalseSubState

Attributs	**Valeur**		
BrowseName	HasFalseSubState		
InverseName	IsFalseSubStateOf		
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'*Event* en définissant le *ConditionType*. Le *ConditionType* introduit le concept d'états qui le différencie du modèle d'*Event* de base. Contrairement aux *BaseEventTypes*, les *Conditions* ne sont pas transitoires. Le *ConditionType* est encore plus étendu en Dialog et *AcknowledgeableConditionTypes*, chacun de ceux-ci ayant leurs propres sous-types.

Le modèle de *Condition* est illustré à la Figure 8 et il est 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. En revanche, les figures illustrent uniquement des informations fournies par les définitions formelles.

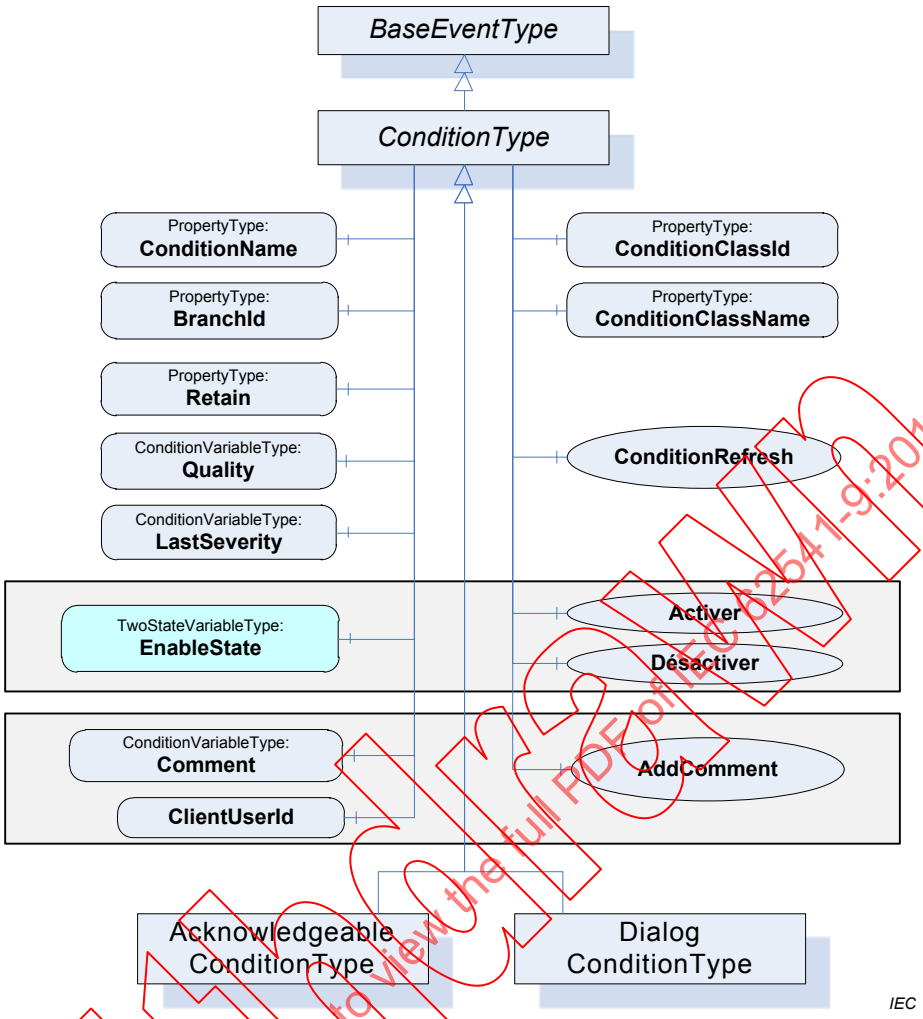


Figure 8 – Modèle de Condition

5.5.2 ConditionType

Le *ConditionType* définit toutes les caractéristiques générales d'une *Condition*. Tous les autres *ConditionTypes* en dérivent. Il est défini de façon formelle dans le Tableau 7. L'état FALSE de l'*EnabledState* ne doit pas être étendu avec un diagramme de sous-états.

Tableau 7 – Définition de ConditionType

Attribut	Valeur				
BrowseName	ConditionType				
IsAbstract	True				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
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	ConditionName	String	PropertyType	Obligatoire
HasProperty	Variable	BranchId	NodeId	PropertyType	Obligatoire
HasProperty	Variable	Retain	Boolean	PropertyType	Obligatoire
HasComponent	Variable	EnabledState	LocalizedText	TwoStateVariableType	Obligatoire
HasComponent	Variable	Quality	StatusCode	ConditionVariableType	Obligatoire
HasComponent	Variable	LastSeverity	UInt16	ConditionVariableType	Obligatoire
HasComponent	Variable	Comment	LocalizedText	ConditionVariableType	Obligatoire
HasProperty	Variable	ClientUserId	String	PropertyType	Obligatoire
HasComponent	Method	Disable	Défini en 5.5.4		Obligatoire
HasComponent	Method	Enable	Défini en 5.5.5		Obligatoire
HasComponent	Method	AddComment	Défini en 5.5.6		Obligatoire
HasComponent	Method	ConditionRefresh	Défini en 5.5.7		Aucune

Le *ConditionType* hérite de toutes les *Properties* du *BaseEventType*. Leur sémantique est définie dans l'IEC 62541-5. *SourceNode* identifie la *ConditionSource*. Voir 5.12 pour plus de détails. Si la *ConditionSource* n'est pas un *Node* (nœud) dans l'*AddressSpace*, le *NodeId* est mis à null. Le *SourceNode* est le *Node* auquel la condition est associée, il peut être identique au InputNode (Nœud d'entrée) pour une alarme, mais peut être un nœud séparé. Par exemple un moteur, qui est une variable avec une valeur équivalant au r/min (RPM), peut être la *ConditionSource* pour les *Conditions* associées au moteur, ainsi qu'un capteur de température également associé au moteur. Dans le premier cas, l'InputNode pour l'alarme High RPM est la valeur de r/min du moteur, tandis que dans le dernier cas, l'InputNode de la High Alarm (Alarme Haute) serait la valeur du capteur de température associé au moteur.

ConditionClassId spécifie le domaine dans lequel cette *Condition* est utilisée. Il s'agit du *NodeId* du *ConditionClassType* correspondant. Voir 5.9 pour la définition de *ConditionClass* et un jeu de *ConditionClasses* définies dans la présente norme. Lorsqu'ils utilisent cette *Property* à des fins de filtrage, les *Clients* sont tenus de spécifier tous les NodeIds de *ConditionClassType* individuels. L'opérateur OfType ne peut pas être appliqué. *BaseConditionClassType* est utilisé comme une classe chaque fois qu'une *Condition* ne peut pas être affectée à une classe plus concrète.

ConditionClassName fournit la désignation d'affichage du *ConditionClassType*.

ConditionName identifie l'instance *Condition* qui est à l'origine de l'*Event*. Il peut être utilisé avec le *SourceName* dans un affichage d'utilisateur pour distinguer entre différentes instances *Condition*. Si une *ConditionSource* n'a qu'une seule instance d'un *ConditionType*, et le *Server* n'a pas de nom d'instance, le *Server* doit fournir le nom de navigation de *ConditionType*.

BranchId est Null pour toutes les *Event Notifications* qui se rapportent à l'état courant de l'instance *Condition*. Si le *BranchId* n'est pas Null, il identifie un état antérieur de cette instance *Condition* qui a encore besoin d'attention de la part d'un *Operator*. Si la *ConditionBranch* courante est transformée en une *ConditionBranch* antérieure, il est nécessaire que le Serveur affecte un BranchId non null. Un *Event* initial pour la branche sera 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 injectée par l'*Operator*, l'*Event* final aura le *Retain* mis

à FALSE. Le bit retain sur l'Event courant est TRUE, tant que les ConditionBranches exigent un entrée de l'Opérateur. Voir 4.4 pour plus d'informations relatives à la nécessité de créer et de maintenir des *ConditionBranches* antérieures et Article B.1 pour un exemple utilisant des branches. Le *DataType BranchId* est égal à *NodeId* bien que le *Server* ne soit pas tenu d'avoir des *ConditionBranches* dans l'*Address Space*. L'utilisation d'un *NodeId* permet au *Server* d'utiliser de simples identificateurs numériques, chaînes ou blobs (Binary Large Object) (matrices d'octets).

Retain lorsqu'il est TRUE décrit une *Condition* (ou *ConditionBranch*) comme étant dans un état qui est intéressant pour un *Client* souhaitant synchroniser son état avec l'état du *Server*. La logique permettant de déterminer la manière dont le fanion est réglé est spécifique au *Server*. Typiquement, toutes les *Alarms actives* auraient leur fanion *Retain* réglé; cependant, il est également possible pour les *Alarms* inactives d'avoir leur fanion *Retain* mis à TRUE.

En traitement normal, lorsqu'un *Client* reçoit un *Event* avec le fanion *Retain* mis à FALSE, il convient que le *Client* considère cela comme une *ConditionBranch* qui n'a plus d'intérêt; dans le cas d'un "affichage courant d'*Alarm*", 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 d'*Event Notifications* et, à ce titre, se traduit par le comportement spécifique suivant:

- lorsque l'instance *Condition* entre dans l'état *Disabled*, la *propriété Retain* de cette *Condition* doit être réglée sur FALSE par le *Server* afin d'indiquer au *Client* que l'instance *Condition* ne présente pas actuellement d'intérêt pour les *Clients*;
- lorsque l'instance *Condition* entre dans l'état activé, la *Condition* doit être évaluée et toutes ses *Propriétés* mises à jour pour refléter les valeurs courantes. Si cette évaluation fait passer la *propriété Retain* à TRUE pour n'importe quelle *ConditionBranch*, une *Event Notification* doit être générée pour la *ConditionBranch* en question;
- le *Server* peut choisir de continuer à effectuer des essais pour une instance *Condition* pendant qu'elle est désactivée. Cependant, aucune *Event Notification* ne sera générée pendant que l'instance *Condition* est désactivée;
- pour n'importe quelle *Condition* qui existe dans l'*AddressSpace*, les *attributs* et les *Variables* suivantes continueront 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 sont plus fournies doivent retourner un statut de Bad_ConditionDisabled.

Dans l'état activé, les changements apportés aux composants suivants doivent entraîner une *Event Notification* de *ConditionType*:

- Quality;
- Severity (héritée de *BaseEventType*);
- Comment.

Il peut ne pas s'agir de la liste exhaustive. Les sous-types peuvent définir des *Variables* supplémentaires qui déclenchent des *Event Notifications*. En général, les changements apportés aux *Variables* des types *TwoStateVariableType* ou *ConditionVariableType* déclenchent des *Event Notifications*.

Quality révèle le statut des valeurs de processus ou autres ressources sur lesquelles cette instance *Condition* est basée. Si, par exemple, une valeur de processus est "Uncertain", la *Condition* "LevelAlarm" associée est également "questionable" (douteuse). Les valeurs pour

Quality peuvent être n'importe lesquelles des *StatusCodes* d'OPC définis dans l'IEC 62541-8 ainsi que *Good*, *Uncertain* et *Bad* tels que définis dans l'IEC 62541-4. Ces *StatusCodes* sont semblables à la description de la qualité de données, mais légèrement plus génériques que celle-ci, dans les diverses spécifications relatives aux bus de terrain. Il est de la responsabilité du *Server* de mettre en correspondance les informations de statut interne avec ces codes. Un *Server* qui ne prend en charge aucune information relative à la qualité doit retourner *Good*. Cette qualité peut également refléter le statut de communication associé au système sur lequel est basée cette valeur ou ressource et duquel cette *Alarm* a été reçue. Pour les erreurs de communication du système sous-jacent, notamment celles donnant lieu à l'indisponibilité de certains champs *Event*, la qualité doit être l'erreur Bad_NoCommunication.

Les *Events* ne sont générés que pour les *Conditions* dont le champ *Retain* est mis à true.

LastSeverity fournit la sévérité antérieure de la *ConditionBranch*. Initialement, cette *Variable* contient une valeur de zéro; elle retournera une valeur uniquement après un changement de sévérité. La nouvelle sévérité est fournie via la *propriété Severity* qui est héritée du *BaseEventType*.

Comment contient le dernier commentaire fourni pour un certain état (*ConditionBranch*). Il peut avoir été fourni par une *Method* AddComment, une autre *Method* ou d'une tout autre manière. La valeur initiale de cette *Variable* est null (vide), à moins qu'elle ne soit fournie d'une tout autre manière. Si une méthode fournit comme option la possibilité d'établir un Commentaire, la valeur de cette *Variable* est alors réinitialisée à null (vide) si un commentaire facultatif n'est pas fourni.

ClientUserId est lié au champ *Comment* et contient l'identité de l'utilisateur qui a inséré le *Comment* le plus récent. La logique pour obtenir le *ClientUserId* est définie dans l'IEC 62541-5.

Le *NodeId* de l'instance *Condition* est utilisé comme *ConditionId*. Il n'est pas explicitement modélisé comme un composant du *ConditionType*. Il peut toutefois faire l'objet d'une requête avec le *SimpleAttributeOperand* suivant, Tableau 8, dans le *SelectClause* de l'*EventFilter*.

Tableau 8 – SimpleAttributeOperand

Designation	Type	Description
SimpleAttributeOperand		
typeId	NodeId	*NodeId* du *ConditionType Node*
browsePath[]	QualifiedName	vide
attributeId	IntegerId	Id du *NodeId Attribute*

5.5.3 Instances Condition et Branch

Les *conditions* sont des *Objects* qui ont un état qui change au fil du temps. Chaque instance *Condition* a un *ConditionId* qui l'identifie de manière unique au sein du *Server*.

Une instance *Condition* peut être un *Object* qui apparaît dans l'*espace d'adresses* du *Server*. Si tel est le cas, le *ConditionId* est le *NodeId* pour l'*Object*.

L'état d'une instance *Condition* à un instant donné correspond aux valeurs établies pour les *Variables* qui appartiennent à l'instance *Condition*. En cas de changement d'une ou plusieurs valeurs des *Variables*, le *Server* génère un *Event* avec un *EventId* unique.

Si un *Client* appelle *Refresh*, le *Server* rendra compte de l'état courant d'une instance *Condition* en envoyant de nouveau le dernier *Event* (à savoir, les mêmes *EventId* et *Time* sont envoyés).

Une *ConditionBranch* est une copie de l'état de l'instance *Condition* qui peut changer indépendamment de l'état courant de l'instance *Condition*. Chaque *Branch* a un identificateur

appelé un *BranchId* qui est unique parmi toutes les *Branches* actives pour une instance *Condition*. De manière générale, les *Branches* ne sont pas visibles dans l'*Address Space* et la présente norme ne définit pas de moyen normalisé de les rendre visibles.

5.5.4 Méthode Disable

Disable est utilisé pour faire passer une instance *Condition* à l'état *Disabled*. Normalement, le *MethodId* passé au *Service* Call est trouvé en parcourant l'instance *Condition* dans l'*AddressSpace*. Cependant, certains *Servers* ne présentent pas d'instances *Condition* dans l'*AddressSpace.* Par conséquent, tous les *Servers* doivent permettre aux *Clients* d'appeler la méthode *Disable* en spécifiant le *ConditionId* comme étant l'*ObjectId* et le *NodeId* bien connu de la déclaration de la *Method* sur le *ConditionType* comme étant le *MethodId.*

Signature

```
Disable();
```

Codes de résultats de la méthode dans le Tableau 9 (définis dans le Service Call)

Tableau 9 – Code de résultats pour la méthode Disable

ResultCode	Description
Bad_ConditionAlreadyDisabled	Voir Tableau 70 pour la description de ce code de résultat.

Le Tableau 10 spécifie la représentation de l'*AddressSpace* pour la méthode *Disable.*

Tableau 10 – Définition de l'AddressSpace pour la méthode Disable

Attribut	Valeur				
BrowseName	Disable				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
AlwaysGeneratesEvent	Définie en 5.10.2			AuditConditionEnableEventType	

5.5.5 Méthode Enable

Enable est utilisé pour faire passer une instance *Condition* à l'état activé. Normalement, le *MethodId* passé au *Service* Call est trouvé en parcourant l'instance *Condition* dans l'*AddressSpace*. Cependant, certains *Servers* ne présentent pas d'instances *Condition* dans l'*AddressSpace*. Par conséquent, tous les *Servers* doivent permettre aux *Clients* d'appeler la méthode *Enable* en spécifiant le *ConditionId* comme étant l'*ObjectId* et le *NodeId* bien connu de la déclaration de la *Method* sur le *ConditionType* comme étant le *MethodId.* Si l'instance *Condition* n'est pas présentée, il peut alors être difficile pour un *Client* de déterminer le *ConditionId* pour une *Condition* désactivée.

Signature

```
Enable();
```

Codes de résultats de la méthode dans le Tableau 11 (définis dans le Service Call)

Tableau 11 –Codes de résultats de la méthode Enable

ResultCode	Description
Bad_ConditionAlreadyEnabled	Voir Tableau 70 pour la description de ce code de résultat.

Le Tableau 12 spécifie la représentation de l'*AddressSpace* pour la méthode *Enable*.

Tableau 12 – Définition de l'AddressSpace pour la méthode Enable

Attribut	Valeur				
BrowseName	Enable				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
AlwaysGeneratesEvent	Définie en 5.10.2			AuditConditionEnableEventType	

5.5.6 Méthode AddComment

AddComment est utilisé pour appliquer un Commentaire à un état spécifique d'une instance *Condition*. Normalement, le *MethodId* passé au *Service* Call est trouvé en parcourant l'instance *Condition* dans l'*AddressSpace*. Cependant, certains *Servers* ne présentent pas d'instances *Condition* dans l'*AddressSpace*. Par conséquent, tous les *Servers* doivent permettre aux *Clients* d'appeler la méthode *AddComment* en spécifiant le *ConditionId* comme étant l'*ObjectId* et le *NodeId* bien connu de la déclaration de la *Method* sur le *ConditionType* comme étant le *MethodId*. La *Method* ne peut pas être appelée sur le nœud *ConditionType*.

Signature

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

Les paramètres sont définis dans le Tableau 13

Tableau 13 – Arguments AddComment

Argument	Description
EventId	EventId identifiant une *Event Notification* particulière où un état a été consigné pour une *Condition*.
Comment	Un texte localisé à appliquer à la *Condition*.

Codes de résultats de la méthode dans le Tableau 14 (définis dans le Service Call).

Tableau 14 – Codes de résultats de AddComment

ResultCode	Description
Bad_MethodInvalid	Voir IEC 62541-4 pour la description de ce code de résultat. La *Condition* adressée ne prend pas en charge l'ajout de commentaires.
Bad_EventIdUnknown	Voir Tableau 70 pour la description de ce code de résultat.
Bad_NodeIdUnknown	Voir IEC 62541-4 pour la description de ce code de résultat. Utilisé pour indiquer que la *Condition* spécifiée n'est pas valide ou que la *méthode* a été appelée sur le *nœud* ConditionType.

Commentaires

Des *Commentaires* sont ajoutés aux occurrences d'*Event* identifiées via un *EventId*. Les *EventIds* où l'*EventType* correspondant ne prend pas en charge de *Comments* sont tous rejetés.

Un *ConditionEvent* – où la *variable Comment* contient ce texte – sera 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 seront consignés.

Le Tableau 15 spécifie la représentation de l'*AddressSpace* pour la méthode *AddComment*.

Tableau 15 – Définition de l'AddressSpace pour la méthode AddComment

Attribut	Valeur				
BrowseName	AddComment				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
HasProperty	*Variable*	InputArguments	Argument[]	PropertyType	Obligatoire
AlwaysGenerates Event	Définie en 5.10.4			AuditConditionCommentEventType	

5.5.7 Méthode ConditionRefresh

ConditionRefresh permet à un *Client* de demander un *Refresh* de toutes les instances *Condition* qui sont actuellement dans un état intéressant (elles ont leur fanion *Retain* réglé). Cela inclut les états antérieurs d'une instance *Condition* pour lesquels le *Server* maintient des *Branches*. Un *Client* invoque généralement cette *Method* lorsqu'il se connecte initialement à un *Server* et suite à n'importe quelle situation, telle que les interruptions de communication, dans lesquelles il exige la resynchronisation au *Server*. Cette *Method* est disponible uniquement avec le *ConditionType* ou ses sous-types. Pour appeler cette *Method*, l'appel doit transmettre le *MethodId* bien connu de la *Method* au *ConditionType* et l'*ObjectId* doit être l'*ObjectId* bien connu du *ConditionType Object*.

Signature

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

Les paramètres sont définis dans le Tableau 16.

Tableau 16 – Paramètres ConditionRefresh

Argument	Description
SubscriptionId	*Subscription* Id valide pour le*Subscription* à rafraîchir.

Codes de résultats de la méthode dans le Tableau 17 (définis dans le Service Call).

Tableau 17 – ReturnCodes de ConditionRefresh

ResultCode	Description
Bad_SubscriptionIdInvalid	Voir IEC 62541-4 pour la description de ce code de résultat
Bad_RefreshInProgress	Voir Tableau 70 pour la description de ce code de résultat

Commentaires

Le 4.5 détaille le concept, les cas d'utilisation et le flux d'informations.

L'argument d'entrée fournit un identificateur de *Subscription* indiquant le *Subscription* (abonnement) de *Client* qui doit être rafraîchi. Si le *Subscription* est accepté, le *Server* réagira comme suit:

1) Le *Server* émet un *RefreshStartEvent* (défini en 5.11.2) marquant le début de *Refresh.* Une copie du *RefreshStartEvent* est mise en file d'attente dans le flux d'*Event* pour chaque *MonitoredItem* (élément surveillé) de *Notifier* dans *Subscription*. Chacune des copies de l'*Event* doit contenir le même *EventId*.
2) Le *Server* émet des *Event Notifications* de n'importe quelles *Retained Conditions* et *Retained Branches* des *Conditions* qui satisfont aux critères du filtre de contenu des *Subscriptions*. Noter que l'EventId pour une telle *notification* rafraîchie doit être identique à celui pour la *Notification* initiale.

3) Le *Server* peut intercaler de nouvelles *Event Notifications* qui n'ont pas été émises précédemment à l'attention du notificateur avec celles envoyées comme partie de la demande de *Refresh*. Les *Clients* doivent vérifier toutes les *Event Notifications* pour détecter une *ConditionBranch* afin d'éviter d'écraser un nouvel état délivré en même temps avec un état plus ancien par le processus *Refresh*.

4) Le *Server* émet un *RefreshEndEvent* (défini en 5.11.3) pour signaler la fin du *Refresh.* Une copie du *RefreshEndEvent* est mise en file d'attente dans le flux d'*Event* pour chaque *MonitoredItem* de *Notifier* dans *Subscription*. Chacune des copies des *Events* doit contenir le même *EventId*.

S'il faut rafraîchir plus d'un *Subscription*, le traitement de matrice de *Service* d'appel normalisé peut alors être utilisé.

Comme mentionné ci-dessus, *ConditionRefresh* doit aussi émettre des *Event Notifications* pour les états antérieurs s'ils requièrent encore l'attention. Cela est particulièrement vrai pour les instances *Condition* dans lesquelles les états antérieurs nécessitent toujours un acquittement ou une confirmation.

Le Tableau 18 spécifie la représentation de l'*AddressSpace* pour la méthode *ConditionRefresh*.

Tableau 18 – Définition de l'AddressSpace pour la méthode ConditionRefresh

Attribut	**Valeur**				
BrowseName	ConditionRefresh				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
HasProperty	*Variable*	InputArguments	Argument[]	PropertyType	Obligatoire
AlwaysGeneratesEvent	Définie en 5.11.2			RefreshStartEvent	
AlwaysGeneratesEvent	Définie en 5.11.3			RefreshEndEvent	

5.6 Modèle Dialog

5.6.1 Généralités

Le modèle Dialog est une extension du modèle *Condition* utilisé par un *Server* pour demander des données d'utilisateur. Il fournit une fonctionnalité semblable aux dialogues de *message* normalisés rencontrés dans la plupart des systèmes d'exploitation. Le modèle peut être facilement personnalisé en fournissant des options de réponse spécifiques au *Server* dans le *ResponseOptionSet* (jeu d'options de réponse) et en ajoutant une fonctionnalité supplémentaire à des *Condition Types* dérivés.

5.6.2 DialogConditionType

Le *DialogConditionType* est utilisé pour représenter des *Conditions* sous la forme de dialogues. Il est illustré à la Figure 9 et défini de façon formelle dans le Tableau 19.

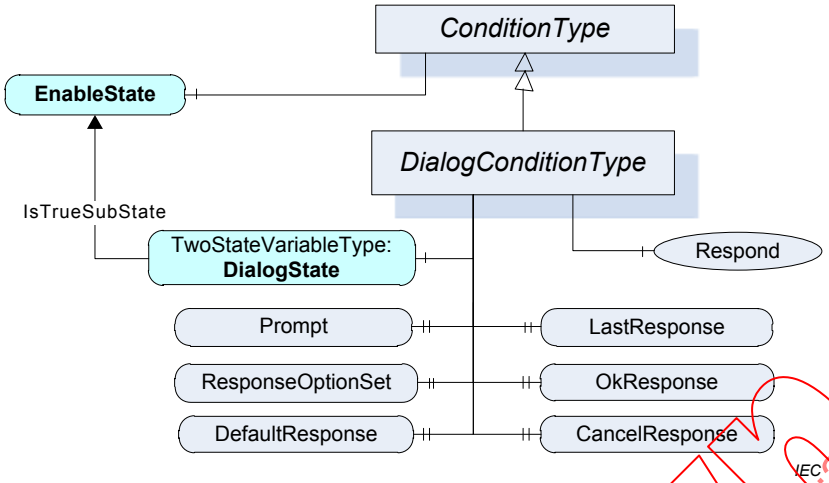


Figure 9 – Vue d'ensemble de DialogConditionType

Tableau 19 – Définition de DialogConditionType

Attribut	**Valeur**				
BrowseName	DialogConditionType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
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	Method	Respond	Défini en 5.6.3.		Obligatoire

Le *DialogConditionType* hérite de toutes les *Properties* du *ConditionType*.

DialogState/Id lorsqu'il est mis à TRUE indique que le *Dialog* est actif et attend une réponse. Les noms d'états recommandés sont décrits dans l'Annexe A.

Prompt (invite de commande) 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, Cancel (Annuler)
- Yes, No, Cancel
- Abort, Retry, Ignore (Abandonner, Réessayer, Ignorer)
- Retry, Cancel

- Yes, No (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 *Property* 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 *Property* est -1.

OkResponse fournit l'indice de l'option OK dans la matrice *ResponseOptionSet*. Ce choix est la réponse qui permettra 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 pour cette option est disponible. Si aucune option OK n'est disponible, la valeur de cette *Property* est -1.

CancelResponse fournit l'indice de réponse dans la matrice *ResponseOptionSet* qui fera passer le Dialog à l'état inactif sans procéder à l'opération décrite par l'invite de commande. Cela permet au *Client* d'identifier l'option *Cancel* si une gestion spéciale pour cette option est disponible. Si aucune option *Cancel* n'est disponible, la valeur de cette *Property* est -1.

5.6.3 Méthode Respond

Respond est utilisé pour passer l'option de réponse sélectionnée et terminer le dialogue. *DialogState/Id* retournera à FALSE.

Signature

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

Les paramètres sont définis dans le Tableau 20.

Tableau 20 – Paramètres Respond

Argument	Description
SelectedResponse	Indice sélectionné dans la matrice *ResponseOptionSet*.

Codes de résultats de la méthode dans le Tableau 21 (définis dans le Service Call).

Tableau 21 – ResultCodes de Respond

ResultCode	Description
Bad_DialogNotActive	Voir Tableau 70 pour la description de ce code de résultat.
Bad_DialogResponseInvalid	Voir Tableau 70 pour la description de ce code de résultat.

Le Tableau 22 spécifie la représentation de l'*AddressSpace* pour la méthode *Respond*.

Tableau 22 – Définition de l'AddressSpace pour la méthode Respond

Attribut	Valeur				
BrowseName	Respond				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
HasProperty	*Variable*	InputArguments	Argument[]	PropertyType	Obligatoire
AlwaysGeneratesEvent	Définie en 5.10.5			AuditConditionRespondEventType	

5.7 Modèle Condition acquittable

5.7.1 Généralités

Le modèle *Condition* acquittable (Acknowledgeable *Condition*) étend le modèle *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 dans les paragraphes ci-après.

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 illustré à la Figure 10 et défini de façon formelle dans le Tableau 23.

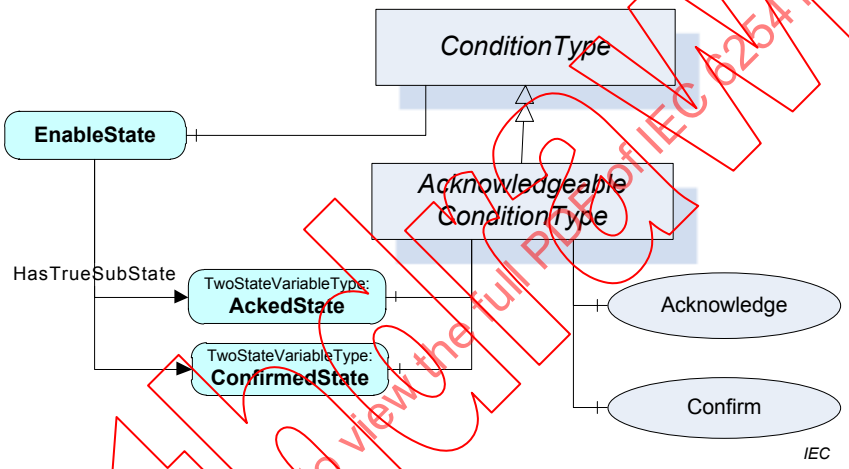


Figure 10 – Vue d'ensemble d'AcknowledgeableConditionType

Tableau 23 – Définition d'AcknowledgeableConditionType

Attribut	**Valeur**				
BrowseName	AcknowledgeableConditionType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
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	Method	Acknowledge	Défini en 5.7.3		Obligatoire
HasComponent	Method	Confirm	Défini en 5.7.4		Facultative

L'*AcknowledgeableConditionType* hérite de toutes les *Properties* du *ConditionType*.

AckedState lorsqu'il est FALSE indique que l'instance *Condition* requiert l'acquittement pour l'état *Condition* consigné. Lorsque l'instance *Condition* est acquittée, l'*AckedState* est mis à 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 relatives aux modèles d'acquittement

et de confirmation. L'*EventId* utilisé dans l'*Event Notification* est considéré être l'identificateur de cet état et est à utiliser pour appeler les *Methods* pour l'acquittement ou la confirmation.

Un *Server* 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 *Server* doit créer une branche de l'instance *Condition* comme spécifié en 4.4. Les *Clients* sont censés garder trace de toutes les *ConditionBranches* où *AckedState*/Id est FALSE pour permettre leur acquittement. Voir également en 5.5.2 pour plus d'informations relatives aux *ConditionBranches* et les exemples en B.1. La gestion de l'*AckedState* et des branches s'applique aussi au *ConfirmState*.

5.7.3 Méthode Acknowledge

Acknowledge est utilisé pour acquitter une *Event Notification* pour un état de l'instance *Condition* où *AckedState* avait été mis à FALSE. Normalement, le *MethodId* passé au *Service* Call est trouvé en parcourant l'instance *Condition* dans l'*AddressSpace*. Cependant, certains *Servers* ne présentent pas d'instances *Condition* dans l'*AddressSpace*. Par conséquent, tous les *Servers* doivent permettre aux *Clients* d'appeler la méthode *Acknowledge* en spécifiant le *ConditionId* comme étant l'*ObjectId* et le *NodeId* bien connu de la déclaration de la *Method* sur l'*AcknowledgeableConditionType* comme étant le *MethodId*. La *Method* ne peut pas être appelée sur le nœud *AcknowledgeableConditionType*.

Signature

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

Les paramètres sont définis dans le Tableau 24.

Tableau 24 – Paramètres Acknowledge

Argument	Description
EventId	EventId identifiant une *Event Notification* particulière. Seules les *Event Notifications* où AckedState/Id était FALSE peuvent être acquittées.
Comment	Un texte localisé à appliquer à la *Condition*.

Codes de résultats de la méthode dans le Tableau 25 (définis dans le Service Call).

Tableau 25 – Codes de résultats de Acknowledge

ResultCode	Description
Bad_ConditionBranchAlreadyAcked	Voir Tableau 70 pour la description de ce code de résultat.
Bad_EventIdUnknown	Voir Tableau 70 pour la description de ce code de résultat.
Bad_NodeIdUnknown	Voir IEC 62541-4 pour la description de ce code de résultat. Utilisé pour indiquer que la *Condition* spécifiée n'est pas valide ou que la *Method* a été appelée sur le *nœud* ConditionType.

Commentaires

Un serveur est chargé d'assurer que chaque *Event* a un *EventId* unique. Cela permet aux *Clients* d'identifier et d'acquitter une *Event Notification* particulière.

L'*EventId* identifie une *Event Notification* spécifique où un état à acquitter avait été consigné. L'acquittement et le commentaire facultatif seront appliqués à l'état identifié par l'*EventId*. Si le champ 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 traduira par une *Event Notification* où *AckedState*/Id est mis à TRUE et la *propriété Comment* contient le texte de l'argument "comment" facultatif. Si un état antérieur est acquitté, le *BranchId* et toutes les valeurs de *Condition* pour cette branche seront consignés. Le Tableau 26 spécifie la représentation de l'*AddressSpace* pour la méthode *Acknowledge*.

Tableau 26 – Définition de l'AddressSpace pour la méthode Acknowledge

Attribut	**Valeur**				
BrowseName	Acknowledge				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
HasProperty	*Variable*	InputArguments	Argument[]	PropertyType	Obligatoire
AlwaysGeneratesEvent	Définie en 5.10.5			AuditConditionAcknowledgeEventType	

5.7.4 Méthode Confirm

Confirm est utilisé pour confirmer une *Event Notification* pour un état de l'instance *Condition* où *ConfirmedState* avait été mis à FALSE. Normalement, le *MethodId* passé au *Service* Call est trouvé en parcourant l'instance *Condition* dans l'*AddressSpace*. Cependant, certains *Servers* ne présentent pas d'instances *Condition* dans l'*AddressSpace*. Par conséquent, tous les *Servers* doivent permettre aux *Clients* d'appeler la méthode *Confirm* en spécifiant le *ConditionId* comme étant l'*ObjectId* et le *NodeId* bien connu de la déclaration de la *Method* sur l'*AcknowledgeableConditionType* comme étant le *MethodId*. La *Method* ne peut pas être appelée sur le nœud *AcknowledgeableConditionType*.

Signature

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

Les paramètres sont définis dans le Tableau 27.

Tableau 27 – Paramètres de la méthode Confirm

Argument	**Description**
EventId	*EventId* identifiant une *Event Notification* particulière. Seules les *Event Notifications* où *ConfirmedState*/Id était TRUE peuvent être confirmées.
Comment	Texte localisé à appliquer aux *Conditions*.

Codes de résultats de la méthode dans le Tableau 28 (définis dans le Service Call).

Tableau 28 – Codes de résultats de la méthode Confirm

ResultCode	**Description**
Bad_ConditionBranchAlreadyConfirmed	Voir Tableau 70 pour la description de ce code de résultat.
Bad_EventIdUnknown	Voir Tableau 70 pour la description de ce code de résultat.
Bad_NodeIdUnknown	Voir IEC 62541-4 pour la description de ce code de résultat. Utilisé pour indiquer que la *Condition* spécifiée n'est pas valide ou que la *Method* a été appelée sur le nœud ConditionType.

Commentaires

Un serveur est chargé d'assurer que chaque *Event* a un *EventId* unique. Cela permet aux *Clients* d'identifier et de confirmer une *Event Notification* particulière.

L'*EventId* identifie une *Event Notification* spécifique où un état à confirmer avait été consigné. Il peut être fourni un *Comment* qui sera appliqué à l'état identifié par *EventId*.

Un *EventId* valide se traduira par une *Event Notification* où *ConfirmedState*/Id est mis à TRUE et la *propriété Comment* contient le texte de l'argument "comment" facultatif. Si un état antérieur est confirmé, le *BranchId* et toutes les valeurs de *Condition* pour cette branche seront consignés.

Le Tableau 29 spécifie la représentation de l'*AddressSpace* pour la méthode *Confirm*.

Tableau 29 – Définition de l'AddressSpace pour la méthode Confirm

Attribut	**Valeur**				
BrowseName	Confirm				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
HasProperty	*Variable*	InputArguments	Argument[]	PropertyType	Obligatoire
AlwaysGeneratesEvent	Définie en 5.10.7			AuditConditionConfirmEventType	

5.8 Modèle Alarm

5.8.1 Généralités

La Figure 11 décrit de manière informelle l'AlarmConditionType, ses sous-types et sa position dans la hiérarchie des *Event Types*.

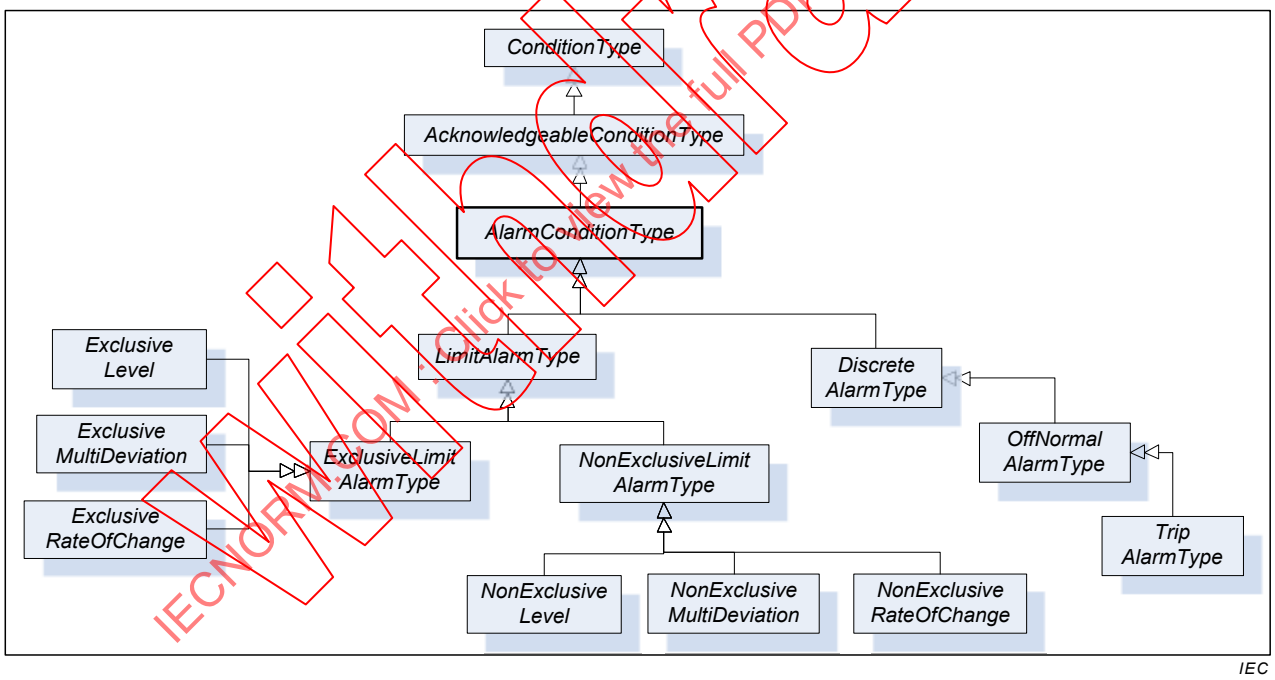


Figure 11 – Modèle de la hiérarchie d'AlarmConditionType

5.8.2 AlarmConditionType

L'*AlarmConditionType* est un type abstrait qui étend l'*AcknowledgeableConditionType* en introduisant un *ActiveState*, un *SuppressedState* et un *ShelvingState*. Le modèle *Alarm* est illustré à la Figure 12. Cette représentation n'est pas censée être une définition complète. Le modèle est défini de façon formelle dans le Tableau 30.

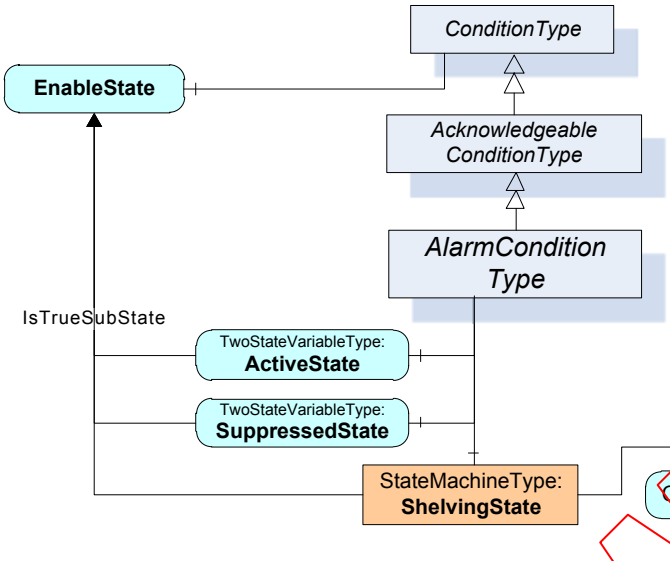


Figure 12 – Modèle Alarm

Tableau 30 – Définition d'AlarmConditionType

Attribut	**Valeur**				
BrowseName	AlarmConditionType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type du 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	Object	ShelvingState		ShelvedStateMachineType	Facultative
HasProperty	Variable	SuppressedOrShelved	Boolean	PropertyType	Obligatoire
HasProperty	Variable	MaxTimeShelved	Duration	PropertyType	Facultative

L'*AlarmConditionType* hérite de toutes les *Properties* de l'*AcknowledgeableConditionType*. Les deux états suivants sont des sous-états d'*EnabledState* TRUE.

ActiveState/Id lorsqu'il est mis à TRUE indique que la situation que *Condition* représente est actuellement présente. Lorsqu'une instance *Condition* est dans l'état inactif (*ActiveState/Id* mis à FALSE), cela représente une situation qui est retournée à un état normal. Les transitions de *Conditions* vers les états inactif et actif sont déclenchées par des actions spécifiques au *Server*. Les sous-types d'AlarmConditionType spécifiés après dans cette norme ont des modèles de sous-état 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 *Value* est utilisée comme donnée d'entrée primaire dans le calcul de l'état *Alarm*. Si cette *Variable* n'est pas dans l'*AddressSpace*, un *NodeId* Null doit être fourni. Dans certains systèmes, une *Alarm* peut être calculée sur la base de plusieurs *Valeurs* de *Variables*, il incombe au système de déterminer quel NodeId de *Variable* est utilisé.

SuppressState est utilisé en interne par un *Server* afin de supprimer automatiquement des *Alarms* pour des raisons spécifiques au système. Par exemple, un système peut être configuré pour supprimer des *Alarms* qui sont associées à une machinerie à l'arrêt, telles qu'une *Alarm* de niveau bas pour un réservoir qui n'est pas en cours d'utilisation. Les noms d'états recommandés sont décrits dans l'Annexe A.

ShelvingState suggère si, oui ou non, une *Alarm* doit (temporairement) ne pas être affichée à l'attention de l'utilisateur. Ceci est très souvent utilisé pour bloquer les *Alarms* liées à une gêne. Le *ShelvingState* est défini en 5.8.3.

Le *SuppressedState* et le *ShelvingState* ensemble se traduisent par l'état *SuppressedOrShelved* de la *Condition*. Lorsqu'une *Alarm* est dans l'un des états, la propriété *SuppressedOrShelved* sera réglée sur TRUE et cette *Alarm* n'est généralement pas affichée par le *Client*. Les transitions d'états associées avec l'*Alarm* ont effectivement lieu, mais elles ne sont généralement pas affichées par les *Client*s tant que l'*Alarm* reste dans l'état "*suppressed*" ou "*shelved*".

La *propriété* facultative *MaxTimeShelved* est utilisée pour attribuer le temps maximal pendant lequel une *Condition* d'*Alarm* peut être suspendue. La valeur est exprimée sous la forme d'une durée. Les systèmes peuvent utiliser cette *Property* pour empêcher une *Shelving* (suspension) permanente d'une *Alarm*. Si cette *Property* est présente, elle sera une limite supérieure de 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 retourné lors de l'appel. Si cette *Property* est présente, elle sera également en vigueur pour l'état OneshotShelved, ainsi une *Condition d'Alarme* (*Alarm Condition*) passera à l'état Unshelved à partir de l'état OneshotShelved si la durée spécifiée dans cette *Property* expire à la suite d'une opération OneShotShelve sans changement des autres éléments éventuels associés à la *Condition*.

Plus de détails concernant le modèle *Alarm* et les divers états peuvent être consultés en 4.8.

5.8.3 ShelvedStateMachineType

5.8.3.1 Vue d'ensemble

Le *ShelvedStateMachineType* définit un diagramme de sous-états qui représente un modèle avancé de filtrage d'Alarmes. Ce modèle est illustré à la Figure 14.

Le modèle d'état prend en charge deux types de *Shelving*: *OneShotShelving* et *TimedShelving*. Ils sont illustrés à la Figure 13. L'illustration comporte les transitions autorisées entre les divers sous-états. *Shelving* est une activité déclenchée par l'*Operator*.

En *OneShotShelving* (suspension en une seule fois), un utilisateur demande qu'une *Alarm* soit suspendue pendant son état *Active*. Ce type de *Shelving* est typiquement utilisé lorsqu'une *Alarm* se produit en continu sur une limite (à savoir, une *Condition* oscille entre l'*Alarm* High et l'*Alarm* HighHigh, toujours dans l'état *Active*). Le *Shelving* en une seule fois s'effacera automatiquement lorsqu'une *Alarm* retourne à un état inactif. Une autre utilisation pour ce type de *Shelving* concerne une zone d'installation qui est arrêtée, à savoir une *Alarm* fonctionnant longtemps telle qu'une *Alarm* de niveau bas pour un réservoir qui n'est pas en cours d'utilisation. Lorsque le réservoir recommence à fonctionner, l'état *Shelving* s'effacera automatiquement.

En *TimedShelving* (suspension temporisée), un utilisateur spécifie qu'une *Alarm* soit suspendue pendant une durée donnée. Ce type de *Shelving* est très souvent utilisé pour bloquer les *Alarms* liées à une gêne. Par exemple, une *Alarm* qui se produit plus de 10 fois en une minute peut être en suspension pendant quelques minutes.

Dans tous les états, la méthode *Unshelve* (libérer la suspension) peut être appelée pour entraîner une transition vers l'état "Unshelve"; cela inclut le *Un-shelving* d'une *Alarm* 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 *Method* comme illustré à la Figure 13. La transition "Time Expired" (temps expiré) est simplement une transition générée par le système qui se produit lorsque la valeur de temps définie comme partie de l'appel de suspension temporisée (Timed Shelved Call) a expiré. La

transition “Any Transition Occurs” (n'importe quelle transition se produit) est également une transition générée par le système; cette transition est générée lorsque la *Condition* passe à un état inactif.

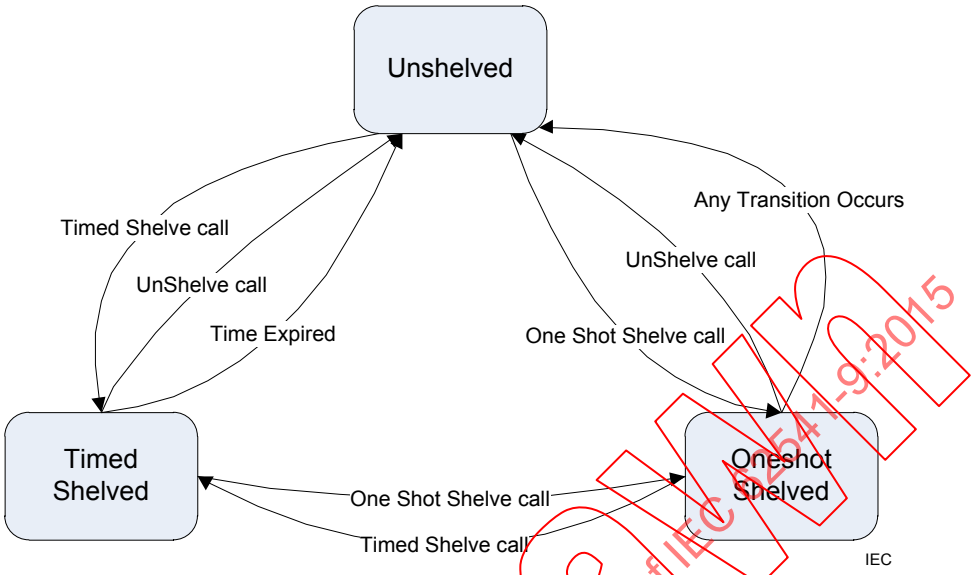


Figure 13 – Transitions d'états de suspension

Le *ShelvedStateMachine* (diagramme d'états suspendus) inclut une hiérarchie de sous-états. Il prend en charge toutes les transitions entre "Unshelved", "OneShotShelved" et "TimedShelved".

Le diagramme d’états est illustré à la Figure 14 et est défini de façon formelle dans le Tableau 31.

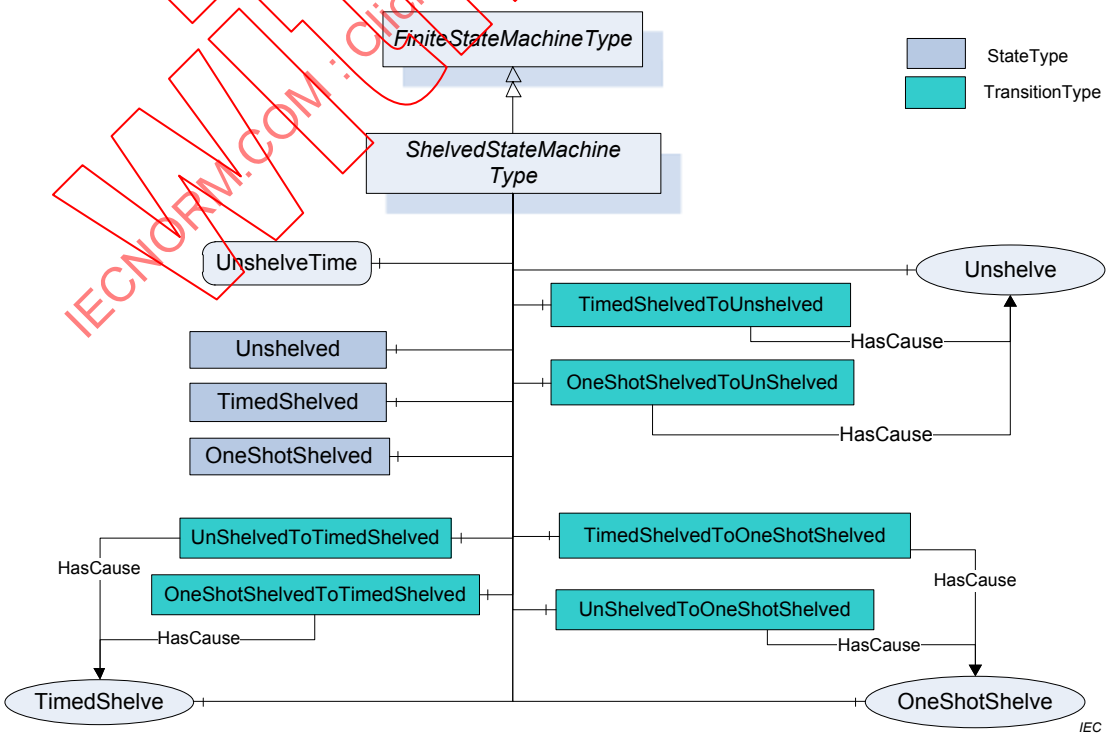


Figure 14 – Modèle de diagramme d’états Shelved (en suspension)

Tableau 31 – Définition de ShelvedStateMachine

Attribut	**Valeur**				
BrowseName	ShelvedStateMachineType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type du Finite*StateMachineType* défini dans l'IEC 62541-5					
HasProperty	Variable	UnshelveTime	Duration	PropertyType	Obligatoire
HasComponent	Object	Unshelved		StateType	Obligatoire
HasComponent	Object	TimedShelved		StateType	Obligatoire
HasComponent	Object	OneShotShelved		StateType	Obligatoire
HasComponent	Object	UnshelvedToTimedShelved		TransitionType	Obligatoire
HasComponent	Object	TimedShelvedToUnshelved		TransitionType	Obligatoire
HasComponent	Object	TimedShelvedToOneShotShelved		TransitionType	Obligatoire
HasComponent	Object	UnshelvedToOneShotShelved		TransitionType	Obligatoire
HasComponent	Object	OneShotShelvedToUnshelved		TransitionType	Obligatoire
HasComponent	Object	OneShotShelvedToTimedShelved		TransitionType	Obligatoire
HasComponent	Method	TimedShelve	Défini en 5.8.3.3		Obligatoire
HasComponent	Method	OneShotShelve	Défini en 5.8.3.4		Obligatoire
HasComponent	Method	Unshelve	Défini en 5.8.3.2		Obligatoire

UnshelveTime spécifie le temps restant en millisecondes jusqu'à ce que l'*Alarm* passe automatiquement à l'état *Un-shelved*. Pour l'état TimedShelved, ce temps est initialisé avec l'argument *ShelvingTime* de l'appel de la *Méthode TimedShelve*. Pour l'état OneShotShelved, le UnshelveTime est une constante réglée sur la durée maximale sauf si une propriété *MaxTimeShelved* est fournie.

Ce *FiniteStateMachine* prend en charge trois états *Active*: Unshelved, TimedShelved et OneShotShelved. Il prend également en charge six transitions. Les états et les transitions sont décrits au Tableau 32. Ce *FiniteStateMachine* prend également en charge trois *Methods*: TimedShelve, OneShotShelve et UnShelve.

Tableau 32 – Transitions de ShelvedStateMachine

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

5.8.3.2 Méthode Unshelve

Unshelve met *AlarmCondition* à l'état Unshelved.

Signature

```
Unshelve();
```

Codes de résultats de la méthode dans le Tableau 33 (définis dans le Service Call).

Tableau 33 – Codes de résultat de la méthode Unshelve

ResultCode	Description
Bad_ConditionNotShelved	Voir Tableau 70 pour la description de ce code de résultat.

Le Tableau 34 spécifie la représentation de l'*AddressSpace* pour la méthode *Unshelve*.

Tableau 34 – Définition de l'AddressSpace pour la méthode Unshelve

Attribut	Valeur				
BrowseName	Unshelve				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
AlwaysGeneratesEvent	Définie en 5.10.7			AuditConditionShelvingEventType	

5.8.3.3 Méthode TimedShelve

TimedShelve met l'*AlarmCondition* à l'état TimedShelved. (Les paramètres sont définis dans le Tableau 35 et les codes de résultats sont décrits dans le Tableau 36).

Signature

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

Tableau 35 – Paramètres TimedShelve

Argument	Description
ShelvingTime	Spécifie un temps fixe pendant lequel l'*Alarm* est à suspendre. Le *Server* peut refuser la durée fournie. Si une propriété *MaxTimeShelved* existe sur l'*Alarm*, le temps de suspension doit être inférieur ou égal à la valeur de cette propriété.

Codes de résultats de la méthode (définis dans le Service Call).

Tableau 36 – Codes de résultats de la méthode TimedShelve

ResultCode	Description
Bad_ConditionAlreadyShelved	Voir Tableau 70 pour la description de ce code de résultat. L'A*larm* est déjà dans l'état TimedShelved et le système n'autorise pas une réinitialisation du temporisateur de suspension.
Bad_ShelvingTimeOutOfRange	Voir Tableau 70 pour la description de ce code de résultat.

Commentaires

La *suspension* pendant une certaine durée est très souvent utilisée pour bloquer les *Alarms* liées à une gêne. Par exemple, une *Alarm* qui se produit plus de 10 fois en une minute peut être en suspension pendant quelques minutes.

Dans certains systèmes, la longueur de temps couverte par cette durée peut être limitée et le *Server* peut générer une erreur refusant la durée proposée. Cette limite peut être présentée comme étant la *propriété MaxTimeShelved*.

Le Tableau 37 spécifie la représentation de l'*AddressSpace* pour la méthode *TimedShelve*.

Tableau 37 – Définition de l'AddressSpace pour la méthode TimedShelve

Attribut	Valeur				
BrowseName	TimedShelve				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Obligatoire
AlwaysGenerates Event	Définie en 5.10.7			AuditConditionShelvingEventType	

5.8.3.4 Méthode OneShotShelve

OneShotShelve met *AlarmCondition* à l'état OneShotShelved.

Signature

```
OneShotShelve();
```

Codes de résultats de la méthode dans le Tableau 38 (définis dans le Service Call).

Tableau 38 – Codes de résultats de la méthode OneShotShelve

ResultCode	Description
Bad_AlreadyShelved	Voir Tableau 70 pour la description de ce code de résultat. L'*Alarm* est déjà dans l'état OneShotShelved.

Le Tableau 39 spécifie la représentation de l'*AddressSpace* pour la méthode *OneShotShelve*.

Tableau 39 – Définition de l'AddressSpace pour la méthode OneShotShelve

Attribut	Valeur				
BrowseName	OneShotShelve				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
AlwaysGeneratesEvent	Définie en 5.10.7			AuditConditionShelvingEventType	

5.8.4 LimitAlarmType

Les *Alarms* peuvent être modélisées avec plusieurs sous-états exclusifs et des limites attribuées ou peuvent être modélisées avec des limites non exclusives qui peuvent être utilisées pour regrouper plusieurs états.

Le *LimitAlarmType* est un type abstrait servant à fournir un *Type* de base pour les *Alarm Conditions* avec plusieurs limites. Le *LimitAlarmType* est illustré à la Figure 15.

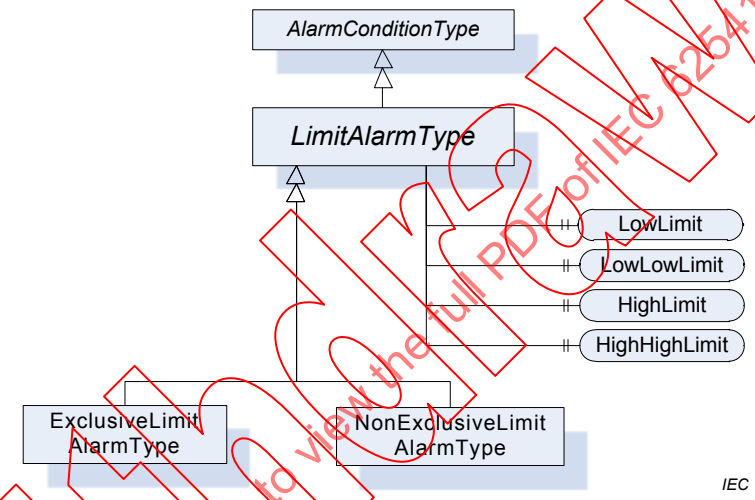


Figure 15 – LimitAlarmType

Le *LimitAlarmType* est défini de façon formelle dans le Tableau 40.

Tableau 40 – Définition de LimitAlarmType

Attribut	Valeur				
BrowseName	LimitAlarmType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type du AlarmConditionType défini en 5.8.2.					
HasSubtype	ObjectType	ExclusiveLimitAlarmType	Défini en 5.8.5.3		
HasSubtype	ObjectType	NonExclusiveLimitAlarmType	Défini en 5.8.6		
HasProperty	Variable	HighHighLimit	Double	PropertyType	Facultative
HasProperty	Variable	HighLimit	Double	PropertyType	Facultative
HasProperty	Variable	LowLimit	Double	PropertyType	Facultative
HasProperty	Variable	LowLowLimit	Double	PropertyType	Facultative

Il est défini quatre limites facultatives qui configurent les états des types d'*Alarms* limites dérivés. Ces propriétés doivent être établies pour toutes les éventuelles limites d'*Alarm* qui sont présentées par les types d'*Alarms* limites dérivés. Ces *propriétés* sont énumérées comme facultatives mais l'une au moins est requise. Pour les cas où un système sous-jacent ne peut pas fournir la valeur effective d'une limite, la *propriété* limite doit toujours être fournie, mais aura son *AccessLevel* réglé sur "not readable" (non lisible).

Les limites d'*Alarm* énumérées peuvent entraîner la génération d'une *Alarm* lorsqu'une valeur devient égale à la limite ou peut générer une *Alarm* lorsque la limite est dépassée (à savoir, la valeur est au-dessus de la limite pour HighLimit et en dessous de la limite pour LowLimit). Le comportement exact lorsque la valeur est égale à la limite est spécifique au *Server*.

5.8.5 ExclusiveLimit Types

5.8.5.1 Vue d'ensemble

Le présent paragraphe décrit le diagramme d'états et le comportement du Type d'*Alarm* de base pour les types d'*Alarme* comportant plusieurs limites mutuellement exclusives.

5.8.5.2 ExclusiveLimitStateMachineType

L'*ExclusiveLimitStateMachineType* (type de diagramme d'états aux limites exclusives) définit le diagramme d'états utilisé par des *AlarmTypes* qui gèrent plusieurs limites mutuellement exclusives. Il est illustré à la Figure 16.

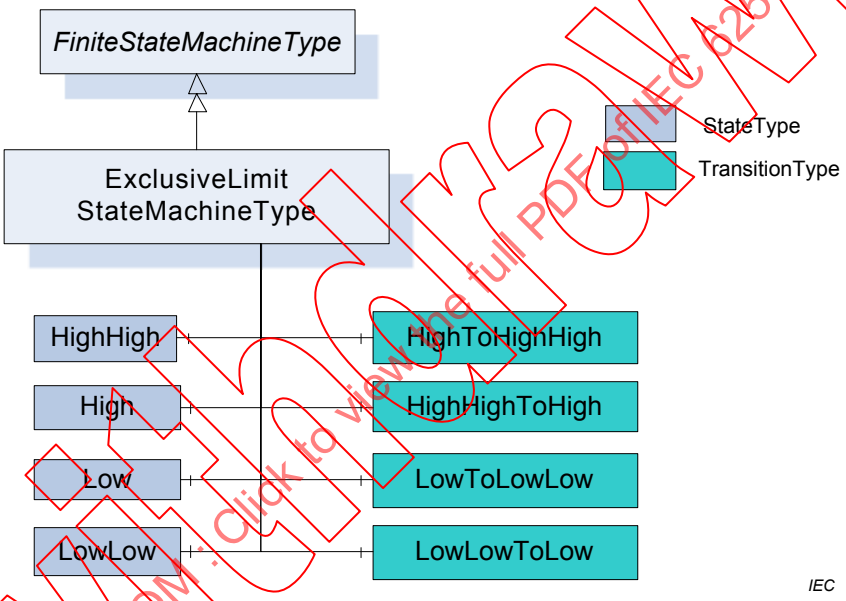


Figure 16 – ExclusiveLimitStateMachine

Il est créé en étendant le *FiniteStateMachineType*. Il est défini de façon formelle dans le Tableau 41 et les transitions d'états sont décrites au Tableau 42.

Tableau 41 – Définition d'ExclusiveLimitStateMachineType

Attribut	Valeur				
BrowseName	ExclusiveLimitStateMachineType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type du *FiniteStateMachineType*					
HasComponent	Object	HighHigh		StateType	Facultative
HasComponent	Object	High		StateType	Facultative
HasComponent	Object	Low		StateType	Facultative
HasComponent	Object	LowLow		StateType	Facultative
HasComponent	Object	LowToLowLow		TransitionType	Facultative
HasComponent	Object	LowLowToLow		TransitionType	Facultative
HasComponent	Object	HighToHighHigh		TransitionType	Facultative
HasComponent	Object	HighHighToHigh		TransitionType	Facultative

Tableau 42 – Transitions d'ExclusiveLimitStateMachineType

BrowseName	Références	BrowseName	TypeDefinition
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	

L'ExclusiveLimitStateMachine définit le diagramme de sous-états qui représente le niveau réel d'une *Alarm* à plusieurs niveaux lorsqu'elle est dans l'état *Active*. Le diagramme de sous-états défini ici inclut les états High, Low, HighHigh et LowLow. Ce modèle inclut également dans son état de transition une série de transitions vers et depuis un état parent, l'état inactif. Ce diagramme d'états tel que défini doit être utilisé comme un diagramme de sous-états pour un diagramme d'états qui a un état *Active*. Cet état *Active* pourrait être une *Alarm* "niveau" ou une *Alarm* "écart" ou n'importe quel autre diagramme d'états d'*Alarm*.

Les états LowLow, Low, High, HighHigh sont caractéristiques pour de nombreuses industries. Les fournisseurs peuvent introduire des modèles de sous-états qui incluent des limites supplémentaires; ils peuvent également omettre des limites dans une instance.

5.8.5.3 ExclusiveLimitAlarmType

L'*ExclusiveLimitAlarmType* est utilisé pour spécifier le comportement commun pour les *Alarm Types* ayant plusieurs limites mutuellement exclusives. L'*ExclusiveLimitAlarmType* est illustré à la Figure 17.

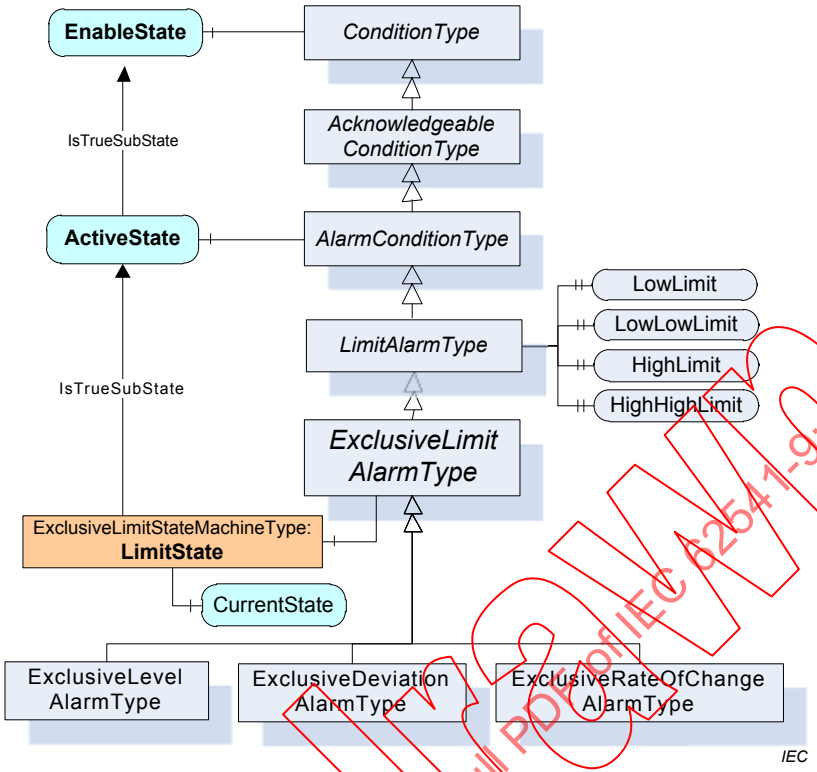


Figure 17 – ExclusiveLimitAlarmType

L'*ExclusiveLimiAlarmType* est défini de façon formelle dans le Tableau 43.

Tableau 43 – Définition d'ExclusiveLimitAlarmType

Attribut	**Valeur**				
BrowseName	ExclusiveLimitAlarmType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type du LimitAlarmType défini en 5.8.4.					
HasSubtype	ObjectType	ExclusiveLevelAlarmType	Défini en 5.8.7.3		
HasSubtype	ObjectType	ExclusiveDeviationAlarm Type	Défini en 5.8.8.3		
HasSubtype	ObjectType	ExclusiveRateOfChange AlarmType	Défini en 5.8.9.3		
HasComponent	Object	LimitState		*ExclusiveLimitStateMachine Type*	Obligatoire

Le *LimitState* est un sous-état de l'ActiveState et comporte une référence *IsTrueSubstate* à *ActiveState*. Le *LimitState* représente la violation effective de limite d'une *ExclusiveLimitAlarm*. Lorsque l'ActiveState du AlarmConditionType est inactif, le *LimitState* ne doit pas être disponible et doit renvoyer la valeur NULL en lecture. Les *Events* qui ont soumis un abonnement pour les champs issus du *LimitState* lorsqu'*ActiveState* est inactif, doivent renvoyer une valeur NULL pour ces champs non disponibles.

5.8.6 NonExclusiveLimitAlarmType

Le *NonExclusiveLimitAlarmType* est utilisé pour spécifier le comportement commun pour les *Alarm Types* ayant plusieurs limites non exclusives. Le *NonExclusiveLimitAlarmType* est illustré à la Figure 18.

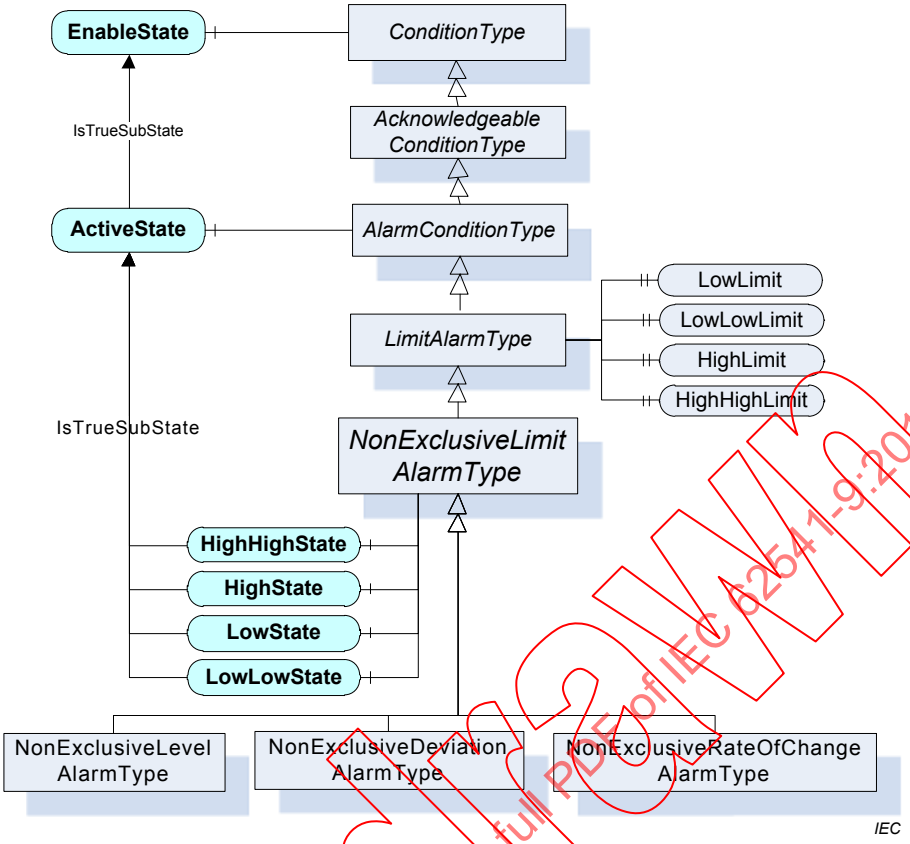


Figure 18 – NonExclusiveLimitAlarmType

Le *NonExclusiveLimitAlarmType* est défini de façon formelle dans le Tableau 44.

Tableau 44 – Définition de NonExclusiveLimitAlarmType

Attribut	**Valeur**				
BrowseName	NonExclusiveLimitAlarmType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type du LimitAlarmType défini en 5.8.4.					
HasSubtype	ObjectType	NonExclusiveLevelAlarmType	Défini en 5.8.7.2		
HasSubtype	ObjectType	NonExclusiveDeviationAlarmType	Défini en 5.8.8.2		
HasSubtype	ObjectType	NonExclusiveRateOfChangeAlarmType	Défini en 5.8.9.2		
HasComponent	Variable	HighHighState	LocalizedText	TwoStateVariableType	Facultative
HasComponent	Variable	HighState	LocalizedText	TwoStateVariableType	Facultative
HasComponent	Variable	LowState	LocalizedText	TwoStateVariableType	Facultative
HasComponent	Variable	LowLowState	LocalizedText	TwoStateVariableType	Facultative

HighHighState, *HighState*, *LowState* et *LowLowState* représentent les états non exclusifs. À titre d'exemple, il est possible que le HighState et le HighHighState soient tous deux dans leur état TRUE. Les fournisseurs peuvent choisir de prendre en charge n'importe quel sous-ensemble de ces états. Les noms d'états recommandés sont décrits dans l'Annexe A.

Il est défini quatre limites facultatives qui configurent ces états. Même si tous les états sont facultatifs, l'état HighState ou LowState doit au moins être fourni. Il découle de la définition d'un HighState et d'un LowState que ces regroupements sont mutuellement exclusifs. Une valeur ne peut pas dépasser simultanément à la fois une valeur HighState et une valeur LowState.

5.8.7 Alarm niveau

5.8.7.1 Vue d'ensemble

Une *Alarm* "niveau" est communément utilisée pour rapporter qu'une limite a été dépassée. Elle se rapporte typiquement à un instrument, par exemple un instrument de mesure de température. L'*Alarm* "niveau" devient active lorsque la valeur observée se situe au-dessus de la limite haute ou en dessous de la limite basse.

5.8.7.2 NonExclusiveLevelAlarmType

Le *NonExclusiveLevelAlarmType* est une *Alarm* de niveau spéciale avec un ou plusieurs états non exclusifs. Si, par exemple, il est nécessaire de maintenir les états High et HighHigh comme actifs en même temps, il convient d'utiliser cet *AlarmType*.

Le *NonExclusiveLevelAlarmType* est fondé sur le *NonExclusiveLimitAlarmType*. Il est défini de façon formelle dans le Tableau 45.

Tableau 45 – Définition de NonExclusiveLevelAlarmType

Attribut	Valeur				
BrowseName	NonExclusiveLevelAlarmType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type du NonExclusiveLimitAlarmType défini en 5.8.6.					

Il n'est pas défini de *Propriétés* supplémentaires du *NonExclusiveLimitAlarmType*.

5.8.7.3 ExclusiveLevelAlarmType

L'*ExclusiveLevelAlarmType* est une *Alarm* de niveau spéciale utilisée avec plusieurs limites mutuellement exclusives. Il est défini de façon formelle dans le Tableau 46.

Tableau 46 – Définition d'ExclusiveLevelAlarmType

Attribut	Valeur				
BrowseName	ExclusiveLevelAlarmType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Hérite des Propriétés de l'ExclusiveLimitAlarmType défini en 5.8.5.3.					

Il n'est pas défini de *Propriétés* supplémentaires de l'*ExclusiveLimitAlarmType*.

5.8.8 Alarm Ecart

5.8.8.1 Vue d'ensemble

Une *Alarm* "écart" est communément utilisée pour rapporter un écart excessif entre le niveau de consigne souhaité d'une valeur de processus et une mesure effective de la valeur en question. L'*Alarm* "écart" devient active lorsque l'écart passe au-dessus ou chute en dessous d'une limite définie.

Par exemple, si un point de consigne a une valeur de 10 et la limite haute de l'*Alarm* "écart" est réglée sur 2 et la limite basse de l'*Alarm* "écart" a une valeur de -1, il y a passage dans le sous-état bas si la valeur de processus chute en dessous de 9; il y a passage dans le sous-état haut si la valeur de processus devient supérieure à 12. Si le point de consigne passait à 11, les nouvelles valeurs de l'écart seraient respectivement 10 et 13.

5.8.8.2 NonExclusiveDeviationAlarmType

Le *NonExclusiveDeviationAlarmType* est une *Alarm* de niveau spéciale avec un ou plusieurs états non exclusifs. Si, par exemple, il est nécessaire de maintenir les états High et HighHigh comme actifs en même temps, il convient d'utiliser cet *AlarmType*.

Le *NonExclusiveDeviationAlarmType* est fondé sur le *NonExclusiveLimitAlarmType*. Il est défini de façon formelle dans le Tableau 47.

Tableau 47 – Définition de NonExclusiveDeviationAlarmType

Attribut	**Valeur**					
BrowseName	NonExclusiveDeviationAlarmType					
IsAbstract	False					
Références	**NodeClass**	**BrowseName**		**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type du NonExclusiveLimitAlarmType défini en 5.8.6.						
HasProperty	Variable	SetpointNode		NodeId	PropertyType	Obligatoire

La *propriété SetpointNode* (point de consigne du nœud) fournit le *NodeId* du point de consigne utilisé dans le calcul de l'écart. Si cette *Variable* n'est pas dans l'*AddressSpace*, un *NodeId* Null doit être fourni.

5.8.8.3 ExclusiveDeviationAlarmType

L'*ExclusiveDeviationAlarmType* est utilisé avec plusieurs limites mutuellement exclusives. Il est défini de façon formelle dans le Tableau 48.

Tableau 48 – Définition d'ExclusiveDeviationAlarmType

Attribut	**Valeur**					
BrowseName	ExclusiveDeviationAlarmType					
IsAbstract	False					
Références	**NodeClass**	**BrowseName**		**DataType**	**TypeDefinition**	**Règle de modélisation**
Hérite des *propriétés* de l'*ExclusiveLimitAlarmType* défini en 5.8.5.3.						
HasProperty	Variable	SetpointNode		NodeId	PropertyType	Obligatoire

La *propriété SetpointNode* (point de consigne du nœud) fournit le *NodeId* du point de consigne utilisé dans le calcul de l'*écart*. Si cette *Variable* n'est pas dans l'*AddressSpace*, un *NodeId* Null doit être fourni.

5.8.9 Rate of Change (Vitesse de variation)

5.8.9.1 Vue d'ensemble

Une *Alarm Rate* of *Change* (vitesse de variation) est communément utilisée pour rapporter une variation inhabituelle ou une absence de variation d'une valeur mesurée liée à la vitesse à laquelle la valeur a changé. L'*Alarm Rate* of *Change* devient active lorsque la vitesse de variation d'une valeur dépasse ou devient inférieure à une limite définie.

Une *Rate* of *Change* est mesurée en une certaine unité de temps (secondes ou minutes par exemple) et une certaine unité de mesure (pourcentage ou mètre par exemple). Par exemple, un réservoir peut avoir une limite High pour la *Rate* of *Change* de son niveau (mesuré en mètres) qui serait de 4 m/min. Si le niveau du réservoir varie à une vitesse supérieure à 4 m/min, il y a passage dans le sous-état High.

5.8.9.2 NonExclusiveRateOfChangeAlarmType

Le *NonExclusiveRateOfChangeAlarmType* est une *Alarm* de niveau spéciale utilisée avec un ou plusieurs états non exclusifs. Si, par exemple, il est nécessaire de maintenir les états High et HighHigh comme actifs en même temps, il convient d'utiliser cet *AlarmType*.

Le *NonExclusiveRateOfChangeAlarmType* est fondé sur le *NonExclusiveLimitAlarmType*. Il est défini de façon formelle dans le Tableau 49.

Tableau 49 – Définition de NonExclusiveRateOfChangeAlarmType

Attribut	**Valeur**				
BrowseName	NonExclusiveRateOfChangeAlarmType				
IsAbstract	False				
Références	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type du NonExclusiveLimitAlarmType défini en 5.8.6.					

Il n'est pas défini de *Propriétés* supplémentaires du *NonExclusiveLimitAlarmType*.

5.8.9.3 ExclusiveRateOfChangeAlarmType

ExclusiveRateOfChangeAlarmType est utilisé avec plusieurs limites mutuellement exclusives. Il est défini de façon formelle dans le Tableau 50.

Tableau 50 – Définition d'ExclusiveRateOfChangeAlarmType

Attribut	**Valeur**				
BrowseName	ExclusiveRateOfChangeAlarmType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Hérite des *propriétés* de l'*ExclusiveLimitAlarmType* défini en 5.8.5.3.					

Il n'est pas défini de *Propriétés* supplémentaires de l'*ExclusiveLimitAlarmType*.

5.8.10 Alarmes de type Discret

5.8.10.1 DiscreteAlarmType

Le *DiscreteAlarmType* est utilisé pour classer les *Types* en *Alarm Conditions* (conditions d'alarme) où l'entrée pour *Alarm* ne peut prendre qu'un certain nombre de valeurs possibles (par exemple vrai/faux, en marche/à l'arrêt/se terminant). Le *DiscreteAlarmType* avec des sous-types définis dans la présente norme est illustré à la Figure 19. Il est défini de façon formelle dans le Tableau 51.

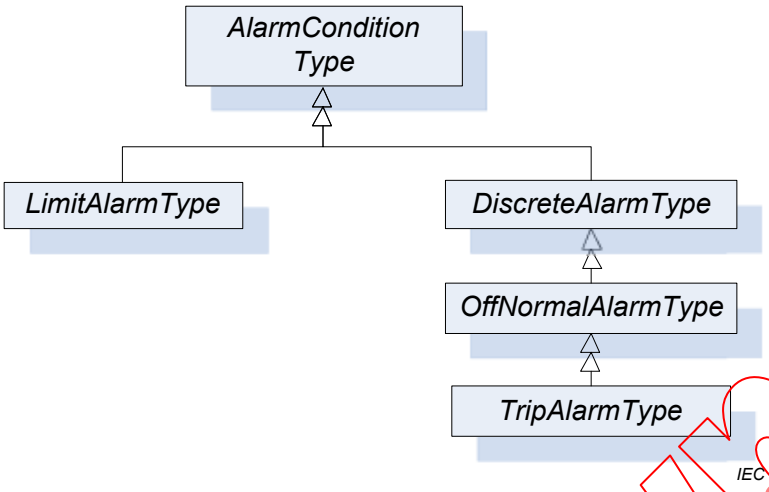


Figure 19 – Hiérarchie de DiscreteAlarmType

Tableau 51 – Définition de DiscreteAlarmType

Attribut	**Valeur**				
BrowseName	DiscreteAlarmType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**Data Type**	**TypeDefinition**	**Règle de modélisation**
Sous-type de l'AlarmConditionType défini en 5.8.2.					
HasSubtype	*ObjectType*	OffNormalAlarmType	Défini en 5.8.8		

5.8.10.2 OffNormalAlarmType

L'*OffNormalAlarmType* est une spécialisation du *DiscreteAlarmType* visant à représenter une *Condition* discrète qui est considérée comme anormale. Il est défini de façon formelle dans le Tableau 52. Ce sous-type est habituellement utilisé pour indiquer qu'une valeur discrète est un état d'*Alarm*; il est actif tant que la valeur anormale est présente.

Tableau 52 – Définition d'OffNormalAlarmType

Attribut	**Valeur**				
BrowseName	OffNormalAlarmType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**Data Type**	**TypeDefinition**	**Règle de modélisation**
Sous-type du DiscreteAlarmType défini en 5.8.10.1					
HasSubtype	*ObjectType*	TripAlarmType	Défini en 5.8.10.4		
HasProperty	*Variable*	NormalState	NodeId	PropertyType	Obligatoire

La *propriété NormalState* est une propriété qui désigne une *Variable* ayant une valeur qui correspond à l'une des valeurs possibles de la *Variable* désignée par la *propriété InputNode,* où la valeur de *Variable* de la *propriété NormalState* correspond à la valeur considérée comme étant l'état normal de la *Variable* désignée par la *propriété InputNode.* Lorsque la valeur de la *Variable* référencée par la *propriété InputNode* n'est pas égale à la valeur de la *propriété NormalState*, l'*Alarm* est *Active*. Si cette *Variable* n'est pas dans l'*AddressSpace*, un *NodeId* Null doit être fourni.

5.8.10.3 SystemOffNormalAlarmType

Cette *Condition* est utilisée par un *Server* pour indiquer qu'un système sous-jacent qui fournit des informations d'*Alarm* présente un problème de communication, et que le *Server* peut avoir un état de *Condition* non valide ou incomplet dans *Subscription*. Sa représentation dans l'*AddressSpace* est définie de façon formelle dans le Tableau 53.

Tableau 53 – Définition de SystemOffNormalAlarmType

Attribut	**Valeur**				
BrowseName	SystemOffNormalAlarmType				
IsAbstract	True				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type du *OffNormalAlarmType*, c'est-à-dire que ses *Références* HasProperty se rapportent aux mêmes *Nœuds*.					

5.8.10.4 TripAlarmType

Le *TripAlarmType* est une spécialisation de l'*OffNormalAlarmType* visant à représenter une *Condition* de déclenchement d'un équipement. L'*Alarm* devient active lorsque l'élément d'équipement surveillé accuse un défaut anormal tel que l'arrêt moteur en raison d'une *Condition* de surcharge. Il est défini de façon formelle dans le Tableau 54. Ce *Type* est principalement utilisé pour la catégorisation.

Tableau 54 – Définition de TripAlarmType

Attribut	**Valeur**				
BrowseName	TripAlarmType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**Data Type**	**TypeDefinition**	**Règle de modélisation**
Sous-type de l'*OffNormalAlarmType* défini en 5.8.10.2.					

5.9 ConditionClasses

5.9.1 Vue d'ensemble

Les *Conditions* sont utilisées dans des domaines d'application spécifiques tels que la maintenance, le système ou le processus. La hiérarchie de *ConditionClass* est utilisée pour spécifier les domaines et elle est orthogonale à la hiérarchie du *ConditionType*. La *propriété ConditionClassId* du *ConditionType* est utilisée pour affecter une *Condition* à une *ConditionClass*. Les *Clients* peuvent utiliser cette *propriété* pour filtrer les classes essentielles. OPC UA définit l'*ObjectType* de base pour toutes les *ConditionClasses* et un jeu de classes communes utilisées par de nombreuses industries. La Figure 20 décrit de façon informelle la hiérarchie des *types* de *ConditionClass* définis dans la présente norme.

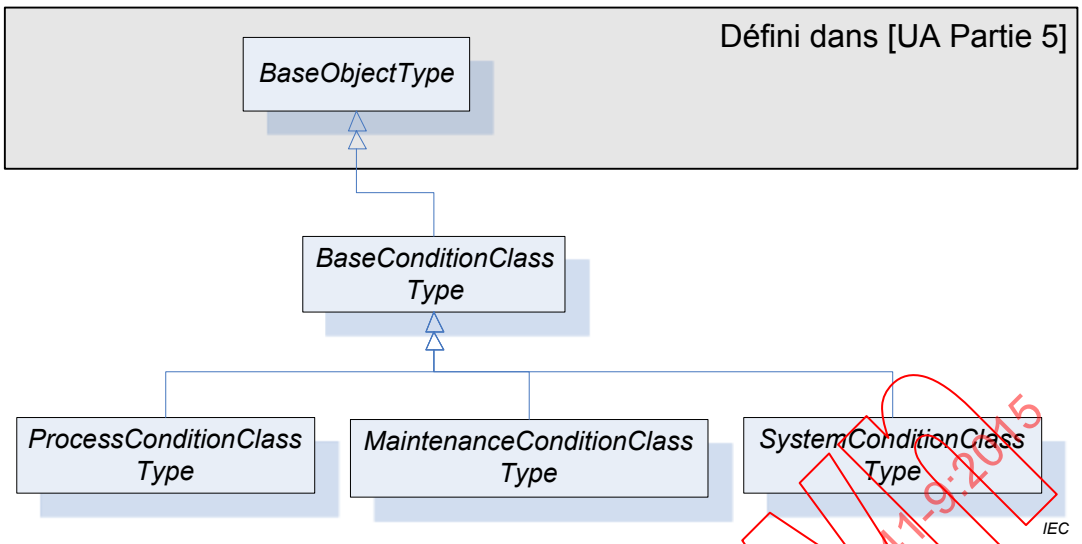


Figure 20 – Hiérarchie de types de ConditionClass

Les *ConditionClasses* ne sont pas des représentations d'*objets* dans le système sous-jacent et, donc, n'existent que comme *Type Nodes* dans l'*AddressSpace*.

5.9.2 ConditionClassType de base

BaseConditionClassType est utilisé comme une classe chaque fois qu'une *Condition* ne peut pas être affectée à une classe plus concrète. Il convient que les *Servers* utilisent une *ConditionClass* plus spécifique, si possible. Tous les *types* de *ConditionClass* dérivent de *BaseConditionClassType*. Il est défini de façon formelle dans le Tableau 55.

Tableau 55 – Définition de BaseConditionClassType

Attribut	**Valeur**				
BrowseName	BaseConditionClassType				
IsAbstract	True				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type du BaseObjectType défini dans l'IEC 62541-5.					

5.9.3 ProcessConditionClassType

Le *ProcessConditionClassType* est utilisé pour classer des *Conditions* relatives au processus lui-même. Les exemples de processus sont les suivants: système de commande d'une chaudière ou instrumentation associée à une usine chimique ou une machine à papier. Le *ProcessConditionClassType* est défini de façon formelle dans le Tableau 56.

Tableau 56 – Définition de ProcessConditionClassType

Attribut	**Valeur**				
BrowseName	ProcessConditionClassType				
IsAbstract	True				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type du BaseConditionClassType défini en 5.9.2					

5.9.4 MaintenanceConditionClassType

Le *MaintenanceConditionClassType* est utilisé pour classer des *Conditions* relatives à la maintenance. Les exemples de maintenance sont les suivants: systèmes de gestion de l'actif ou conditions, qui se produisent avec les systèmes de commande de processus, associés à l'étalonnage des équipements. Le *MaintenanceConditionClassType* est défini de façon formelle dans le Tableau 57. Aucune définition supplémentaire n'est fournie ici. Il est prévu que d'autres groupes de normes définiront les sous-types spécifiques au domaine.

Tableau 57 – Définition de MaintenanceConditionClassType

Attribut	**Valeur**				
BrowseName	MaintenanceConditionClassType				
IsAbstract	True				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type du BaseConditionClassType défini en 5.9.2					

5.9.5 SystemConditionClassType

Le *SystemConditionClassType* est utilisé pour classer des *Conditions* relatives au Système. Il est défini de façon formelle dans le Tableau 58. Les *Conditions* système se produisent dans le processus du système de commande ou de surveillance. Les exemples d'éléments liés au système peuvent inclure l'espace disque disponible sur un ordinateur, la disponibilité des supports d'archivage, les problèmes de charge de réseau ou une erreur de contrôleur. Aucune définition supplémentaire n'est fournie ici. Il est prévu que d'autres groupes de normes ou fournisseurs définiront les sous-types spécifiques au domaine.

Tableau 58 – Définition de SystemConditionClassType

Attribut	**Valeur**				
BrowseName	SystemConditionClassType				
IsAbstract	True				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type du *BaseConditionClassType* défini en 5.9.2					

5.10 Événements Audit

5.10.1 Vue d'ensemble

Les sous-types suivants d'*AuditUpdateMethodEventTypes* seront générés en réponse aux *Méthodes* définies dans la présente norme. Ils sont illustrés à la Figure 21.

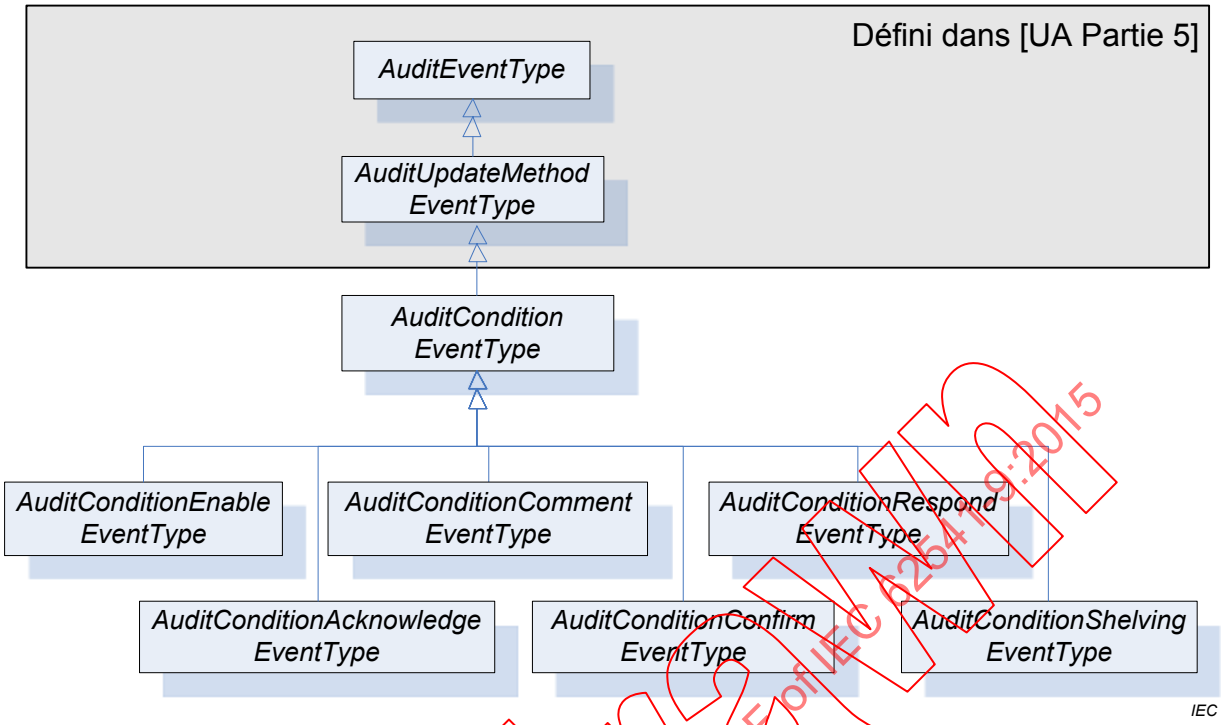


Figure 21 – Hiérarchie d'AuditEvent

Les *AuditConditionEventTypes* sont normalement utilisés en réponse à un appel de *Méthode*. Cependant, ces *Events* doivent aussi être notifiés si la fonctionnalité d'une telle *Méthode* est accomplie par d'autres moyens spécifiques au *Server*. Dans ce cas, la *propriété SourceName* doit contenir une description correcte de ces moyens internes et il convient de remplir les autres propriétés comme décrit pour le type d'*Event* donné.

5.10.2 AuditConditionEventType

Cet *EventType* est utilisé pour subsumer tous les *AuditConditionEventTypes*. Il est défini de façon formelle dans le Tableau 59.

Tableau 59 – Définition d'AuditConditionEventType

Attribut	**Valeur**				
BrowseName	AuditConditionEventType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type de l'*AuditUpdateMethodEventType* défini dans l'IEC 62541-5					

Les AuditConditionEventTypes héritent de toutes les *Properties* du *AuditUpdateMethodEventType* défini dans l'IEC 62541-5. À moins qu'un sous-type neutralise la définition, les propriétés héritées de la *Condition* sont utilisées comme défini.

- La *Propriété SourceNode* héritée doit être remplie avec le *ConditionId*.
- La *SourceName* doit être "Method/" et le nom du *Service* qui a généré l'*Event* (par exemple, *Disable*, Enable, *Acknowledge,* etc.*)*.

Cet *Event* Type peut être adapté davantage afin de refléter les actions particulières relatives à la *Condition*.

5.10.3 AuditConditionEnableEventType

Cet *EventType* est utilisé pour indiquer un changement de l'état activé d'une instance *Condition*. Il est défini de façon formelle dans le Tableau 60.

Tableau 60 – Définition d'AuditConditionEnableEventType

Attribut		Valeur			
BrowseName		AuditConditionEnableEventType			
IsAbstract		False			
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
Sous-type de l'*AuditConditionEventType* défini en 5.10.2, c'est-à-dire qu'il hérite des InstanceDeclarations de ce Nœud.					

Le SourceName doit indiquer Method/Enable ou Method/Disable. Si l'*Event* d'audit n'est pas le résultat d'un appel de *Method*, mais en raison d'une action interne du *Server,* le SourceName doit refléter Enable ou Disable, il peut être précédé d'une description appropriée telle que "Internal/Enable" ou "Remote/Enable"

5.10.4 AuditConditionCommentEventType

Cet *EventType* est utilisé pour rapporter une action *AddComment*. Il est défini de façon formelle dans le Tableau 61.

Tableau 61 – Définition d'AuditConditionCommentEventType

Attribut	Valeur				
BrowseName	AuditConditionCommentEventType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
HasProperty	Variable	EventId	ByteString	PropertyType	Obligatoire
HasProperty	Variable	Comment	LocalizedText	PropertyType	Obligatoire
Sous-type de l'*AuditConditionEventType* défini en 5.10.2 c'est-à-dire qu'il hérite des InstanceDeclarations de ce Nœud.					

Le champ *EventId* doit contenir l'identificateur de l'événement pour lequel le Comment a été ajouté.

Le Comment contient le Comment réel ayant été ajouté.

5.10.5 AuditConditionRespondEventType

Cet *EventType* est utilisé pour rapporter une action *Respond*. Il est défini de façon formelle dans le Tableau 62.

Tableau 62 – Définition d'AuditConditionRespondEventType

Attribut	Valeur				
BrowseName	AuditConditionRespondEventType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Règle de modélisation**
HasProperty	Variable	SelectedResponse	Uint32	PropertyType	Obligatoire
Sous-type de l'*AuditConditionEventType* défini en 5.10.2 c'est-à-dire qu'il hérite des InstanceDeclarations de ce Nœud.					

Le champ SelectedResponse doit contenir la réponse qui a été choisie.

5.10.6 AuditConditionAcknowledgeEventType

Cet *EventType* est utilisé pour indiquer l'acquittement ou la confirmation d'une ou de plusieurs *Conditions*. Il est défini de façon formelle dans le Tableau 63.