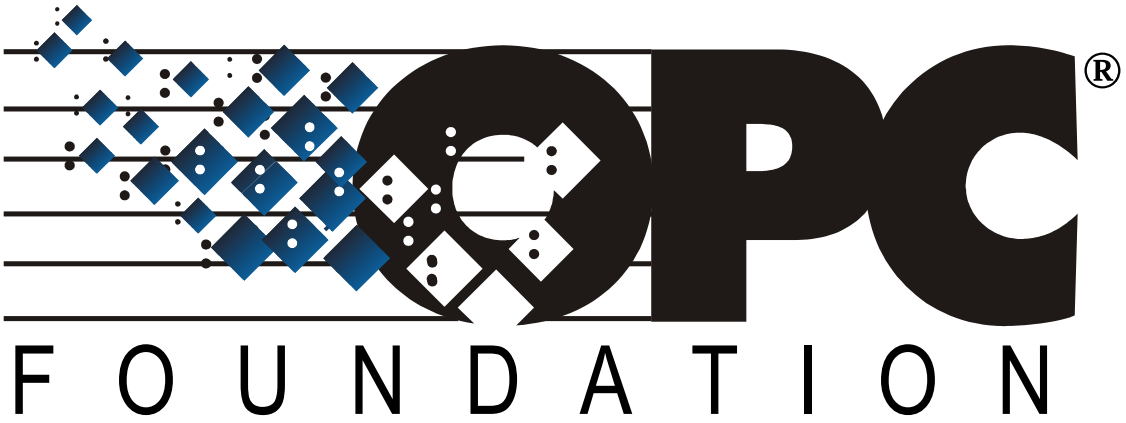

OPC Unified Architecture

Specification

Part 9: Alarms & Conditions

Release 1.00

March 9, 2010

Specification Type:	Industry Standard Specification	Comments:	Report or view errata: http://www.opcfoundation.org/errata
Title:	OPC Unified Architecture Part 9 :Alarms & Conditions	Date:	March 9, 2010
Version:	Release 1.00	Software:	MS-Word
		Source:	OPC UA Part 9 - Alarms and Conditions 1.00 Specification.doc
Author:	OPC Foundation	Status:	Release

CONTENTS

Page

FIGURES

TABLES

OPC FOUNDATION

UNIFIED ARCHITECTURE –

FOREWORD

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

AGREEMENT OF USE

COMPLIANCE

The OPC Foundation shall at all times be the sole entity that may authorize developers, suppliers and sellers of hardware and software to use certification marks, trademarks or other special designations to indicate compliance with these materials. Products developed using this specification may claim compliance or conformance with this specification if and only if the software satisfactorily meets the certification requirements set by the OPC Foundation. Products that do not meet these requirements may claim only that the product was based on this specification and must not claim compliance or conformance with this specification.

TRADEMARKS

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

GENERAL PROVISIONS

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, this specification.

ISSUE REPORTING

The OPC Foundation strives to maintain the highest quality standards for its published specifications, hence they undergo constant review and refinement. Readers are encouraged to report any issues and view any existing errata here: http://www.opcfoundation.org/errata

OPC Unified Architecture Specification

Part 9: Alarms & Conditions

1 Scope

This specification 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 Reference documents

Part 1 : OPC UA Specification: Part 1 – Concepts, Version 1.01 or later.
http://www.opcfoundation.org/UA/Part1/

Part 2 : **OPC UA Specification: Part 2** – Security Model, Version 1.01 or later
http://www.opcfoundation.org/UA/Part2/

Part 3 : **OPC UA Specification: Part 3** – Address Space Model, Version 1.01 or later
http://www.opcfoundation.org/UA/Part3/

Part 4 : **OPC UA Specification: Part 4** – Services, Version 1.01 or later
http://www.opcfoundation.org/UA/Part4/

Part 5 : **OPC UA Specification: Part 5** – Information Model, Version 1.01 or later
http://www.opcfoundation.org/UA/Part5/

Part 6 : **OPC UA Specification: Part 6** – Mappings, Version 1.0 or later
http://www.opcfoundation.org/UA/Part6/

Part 7 : **OPC UA Specification: Part 7** – Profiles, Version 1.0 or later
http://www.opcfoundation.org/UA/Part7/

Part 8 : **OPC UA Specification: Part 8** – Data Access, Version 1.01 or later
http://www.opcfoundation.org/UA/Part8/

Part 11 : **OPC UA Specification: Part 11** – Historical Access, Version 1.01 or later
http://www.opcfoundation.org/UA/Part11/

Additional external reference used to provide information model suggestions for this document:

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

3 Terms, definitions, and abbreviations

3.1 OPC UA Part 1 terms

The following terms defined in Part 1 of this multi-part specification apply.

AddressSpace

Alarm

Attribute

Client

Condition

Event

Information Model

Message

Method

MonitoredItem

Node

NodeClass

Notification

Object

ObjectType

Reference

ReferenceType

Server

Service

Subscription

Variable

3.2 OPC UA Part 3 terms

The following terms defined in Part 3 of this multi-part specification apply.

EventType

ModellingRule

Property

3.3 OPC UA Part 5 terms

The following term defined in Part 5 of this multi-part specification apply.

ClientUserId

In addition base concepts like state machines and *Event* types defined in Part 5 are used in this specification.

3.4 OPC UA Alarms and Condition terms

3.4.1 Acknowledge

“The *Operator* action that indicates recognition of a new *Alarm*” as defined in EEMUA. The term “Accept” is another common term used to describe *Acknowledge*. They can be used interchangeably. This document will use *Acknowledge.*

3.4.2 Active

An *Alarm* in the *Active* state indicates that the situation it is representing currently exists. 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*.

3.4.3 ConditionClass

ConditionClass allows specifying in which domain or for what purpose a certain *Condition* is used. Some top-level *ConditionClasses* are defined in this specification. Vendors or organisations may derive more concrete classes or define different top-level classes.

3.4.4 ConditionBranch

A *ConditionBranch* represents a specific state of a *Condition*. The *Server* can maintain *ConditionBranches* for the current state as well as for previous states.

3.4.5 ConditionSource

A *ConditionSource* is an element which a specific *Condition* is based upon or related to. 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*.

3.4.6 Confirm

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

3.4.7 Disable

“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” as defined in EEMUA.

3.4.8 Operator

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

3.4.9 Refresh

The concept of providing an update to an event *Subscription* that provides all *Alarms* which are considered to be *Retained*. This concept is further described in EEMUA.

3.4.10 Retain

An *Alarm* is to be retained when it is in a state that is interesting for a *Client* wishing to synchronize its state of *Conditions* with the *Server*’s state.

3.4.11 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” as defined in EEMUA.

3.4.12 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” as defined in EEMUA.

3.5 Abbreviations and symbols

DA	Data Access
UA	Unified Architecture
UML	Unified Modelling Language
XML	Extensible Mark-up Language

3.6 Used data types

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

Table 1 – Parameter Types defined in Part 3

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

Table 2 – Parameter Types defined in Part 4

Parameter Type
IntegerId
StatusCode

4 Concepts

4.1 General

This specification defines an *Information Model* for *Conditions*, Dialog *Conditions*, and *Alarms* including acknowledgement capabilities. It is built upon and extends base eventing which is defined in Part 3, Part 4 and Part 5. This *Information Model* can also be extended to support the additional needs of specific domains.

4.2 Conditions

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

- a temperature exceeding a configured limit
- a device needing maintenance
- a batch process that requires a user to confirm some step in the process before proceeding

Each *Condition* instance is of a specific *ConditionType*. The *ConditionType* and derived types are sub-types of the *BaseEventType* (see Part 3 and Part 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*. Section 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. Section 4.4 provides more information and use cases. Maintaining previous states and therefore also the support of multiple branches is optional for *Servers*.

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

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

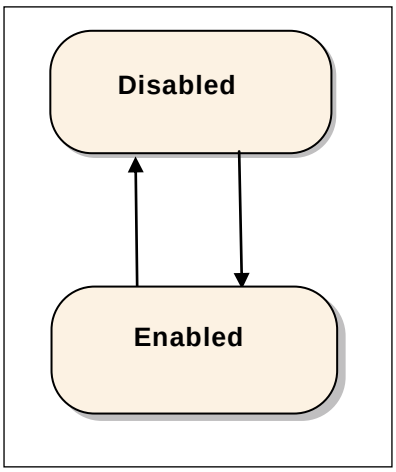

Figure 1 – Base Condition State Model

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

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

The *Server* will generate *AuditEvents* for enable and disable operations (either through a *Method* call or some server / vendor – specific means), rather than generating an *AuditEvent Notification* for each *Condition* instance being enabled or disabled. For more information, see the definition of *AuditConditionEnableEventType* in section 5.10.2.

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

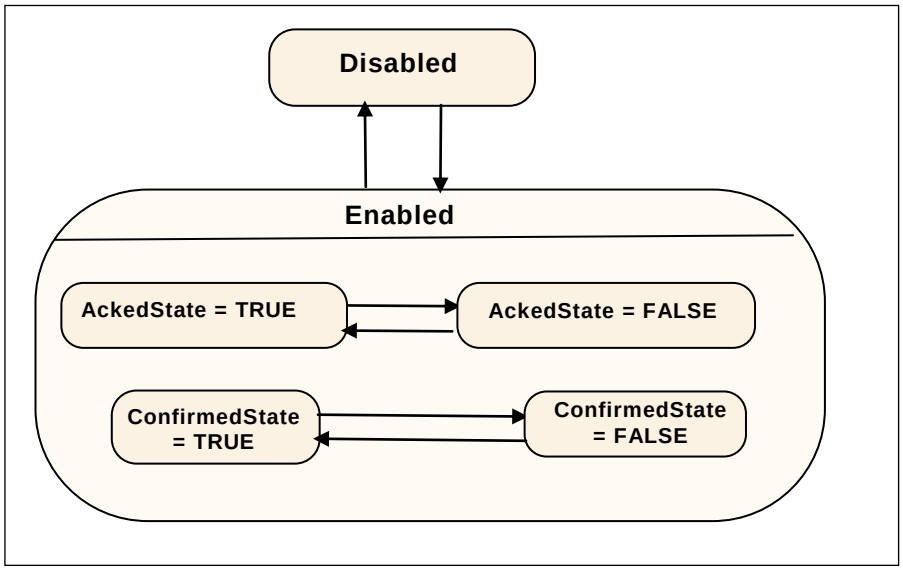

Figure 2 - AcknowledgeableConditions State Model

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

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

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

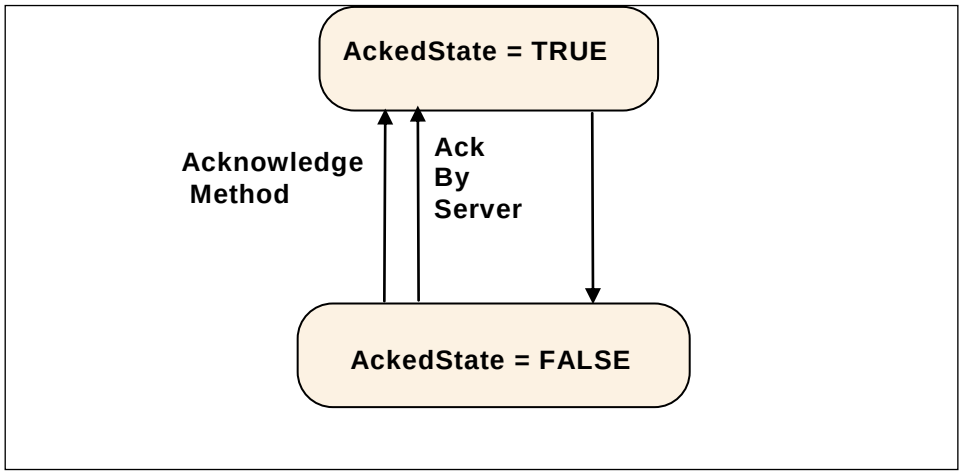

Figure 3 – Acknowledge State Model

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.

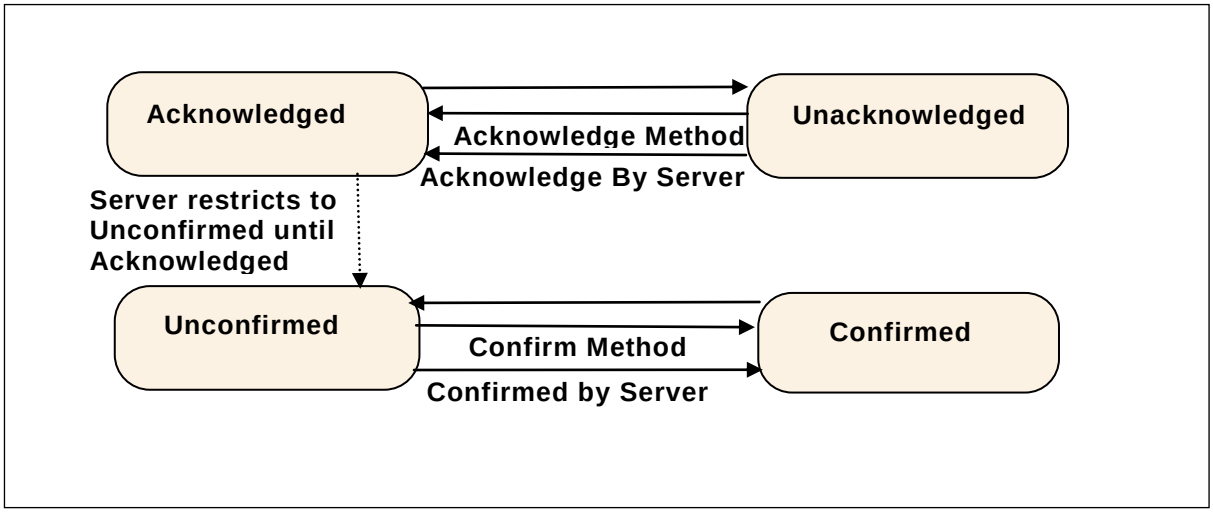

Figure 4 – Confirmed Acknowledge State Model

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 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* must check for multiple *Event Notifications* for a *ConditionBranch* to avoid overwriting a new state delivered together with an older state from the refresh process.

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

The following items should be noted with regard to ConditionRefresh:

- As described in section 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 [UA Part 5]. It indicates the urgency of the *Condition* and is also commonly called ‘priority’, especially in relation to *Alarms* in the *ProcessConditionClass*.

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

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

4.7 Dialogs

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

4.8 Alarms

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

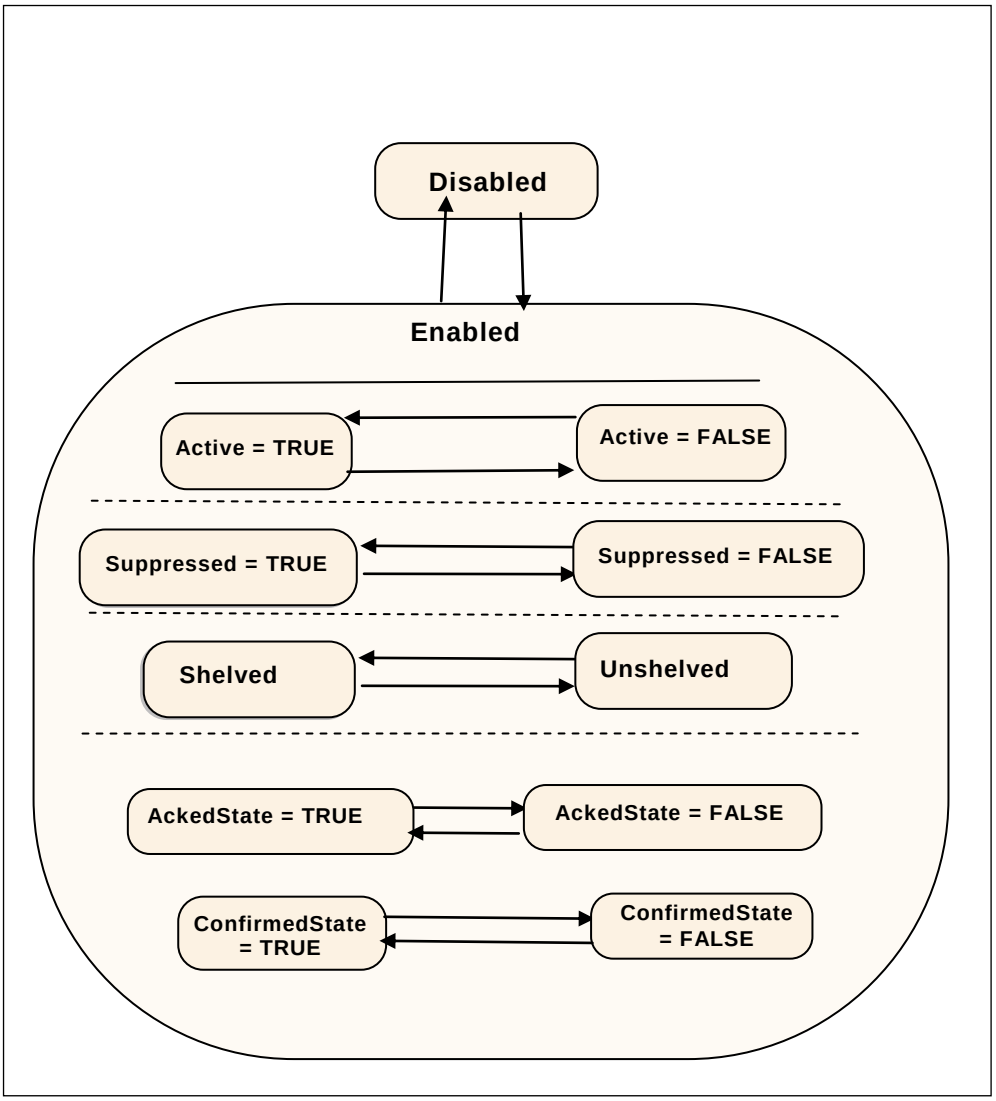

Figure 5 – Alarm State Machine Model

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

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

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 active may be in

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

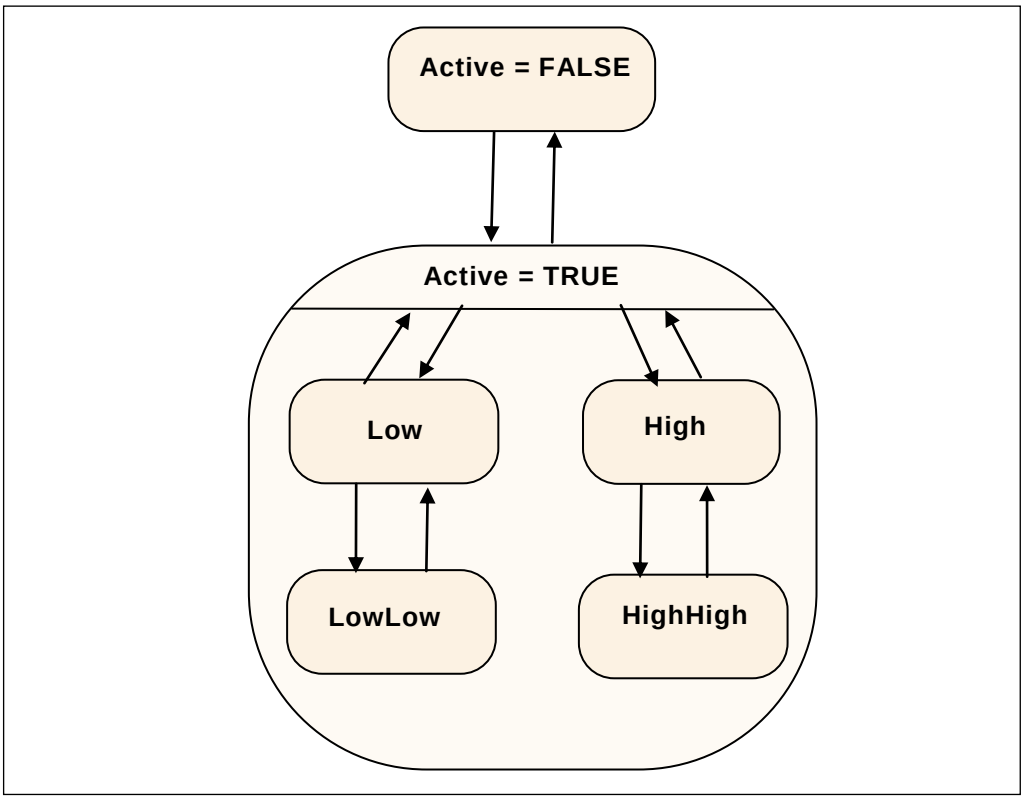

Figure 6 – Multiple Active States Example

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

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

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

4.10 Condition Instances in the Address Space

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

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

While exposing *Condition* instances in the *AddressSpace* is not always possible, doing so allows for direct interaction (read, write and *Method* invocation) with a specific *Condition* instance. For

example, if a *Condition* instance is not exposed, there is no way to invoke the *Enable* or *Disable Method* for the specific *Condition* instance.

4.11 Alarm and Condition Auditing

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

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

This specification describes the *AuditEventType* that each *Method* is required to generate. For example, the *Disable Method* has an *AlwaysGeneratesEvent Reference* to an 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 specification can be further extended to form domain or *Server* specific *Alarm* and *Condition Types*.

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

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

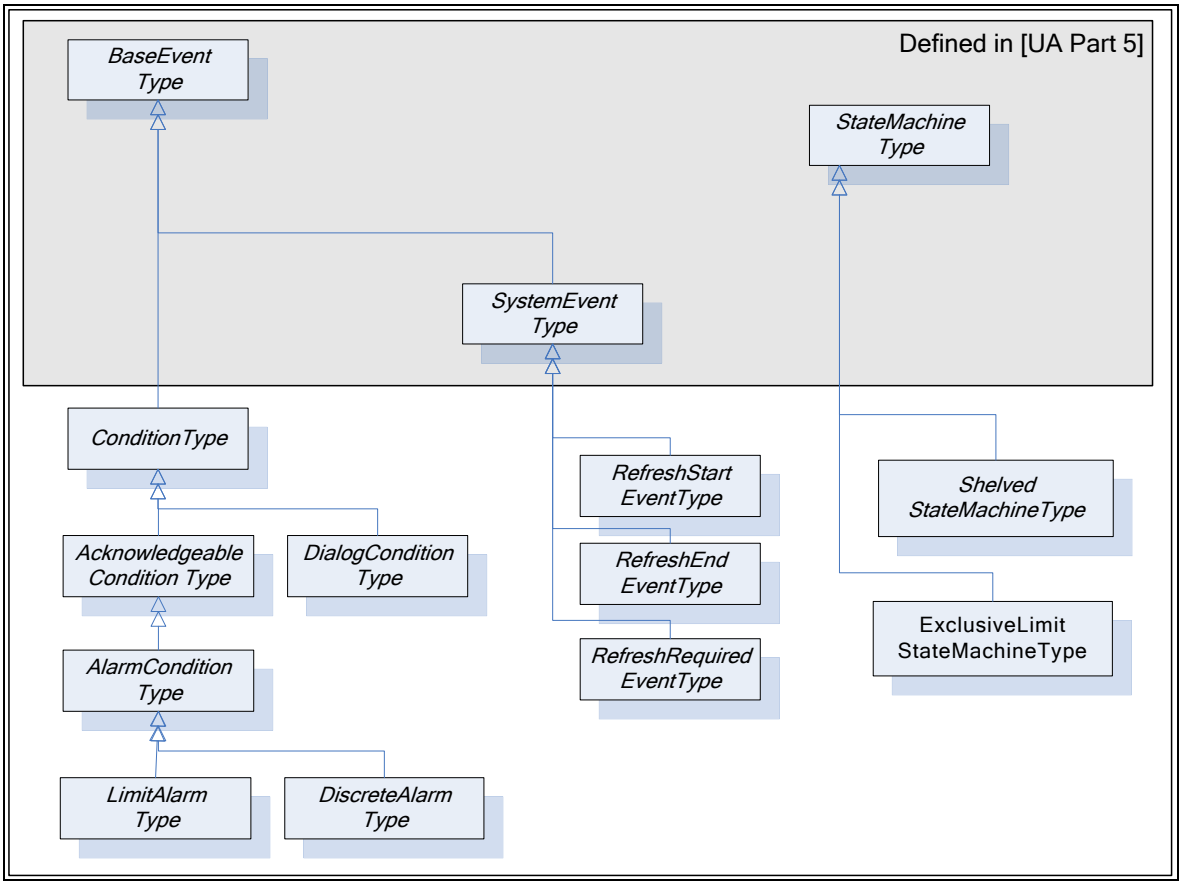

Figure 7 - ConditionType Hierarchy

5.2 Two-State State Machines

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

The *TwoStateVariableType* is derived from the *StateVariableType* defined in Part 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 Part 5. Note that a *Reference* to this subtype is not shown in the definition of the *StateVariableType*					
HasProperty	Variable	Id	Boolean	PropertyType	Mandatory
HasProperty	Variable	TransitionTime	UtcTime	PropertyType	Optional
HasProperty	Variable	EffectiveTransitionTime	UtcTime	PropertyType	Optional
HasProperty	Variable	TrueState	LocalizedText	PropertyType	Optional
HasProperty	Variable	FalseState	LocalizedText	PropertyType	Optional
HasTrueSubState	StateMachine or TwoStateVariableType	<StateIdentifier>	Defined in Clause 5.4.2		Optional
HasFalseSubState	StateMachine or TwoStateVariableType	<StateIdentifier>	Defined in Clause 5.4.3		Optional

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

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

TransitionTime specifies the time when the current state was entered.

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

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

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

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

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

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

5.3 Condition Variables

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

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

Table 4 – ConditionVariableType Definition

Attribute	Value				
BrowseName	ConditionVariableType				
DataType	BaseDataType				
ValueRank	-2 (-2 = Any)				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *BaseDataVariableType* defined in Part 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 section defines *ReferenceTypes* that are needed beyond those already specified as part of Part 3 and Part 5 to extend *TwoState* state machines with substates. These *References* will only exist when substates are available. With this approach *TwoStateVariables* can be extended with subordinate state machines in a similar fashion to the *StateMachineType* defined in Part 5.

5.4.2 HasTrueSubState ReferenceType

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

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

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

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

Table 5 – HasTrueSubState ReferenceType

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

5.4.3 HasFalseSubState ReferenceType

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

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

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

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

Table 6 – HasFalseSubState ReferenceType

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

5.5 Condition Model

5.5.1 General

The *Condition* model extends the *Event* model by defining the *ConditionType*. The *ConditionType* introduces the concept of states differentiating it from the base *Event* model. Unlike the *BaseEventTypes*, *Conditions* are not transient. The *ConditionType* is further extended into Dialog and *AcknowledgeableConditionTypes*, each of which have their own 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 document, is not intended to be complete. Rather, the figures only illustrate information provided by the formal definitions.

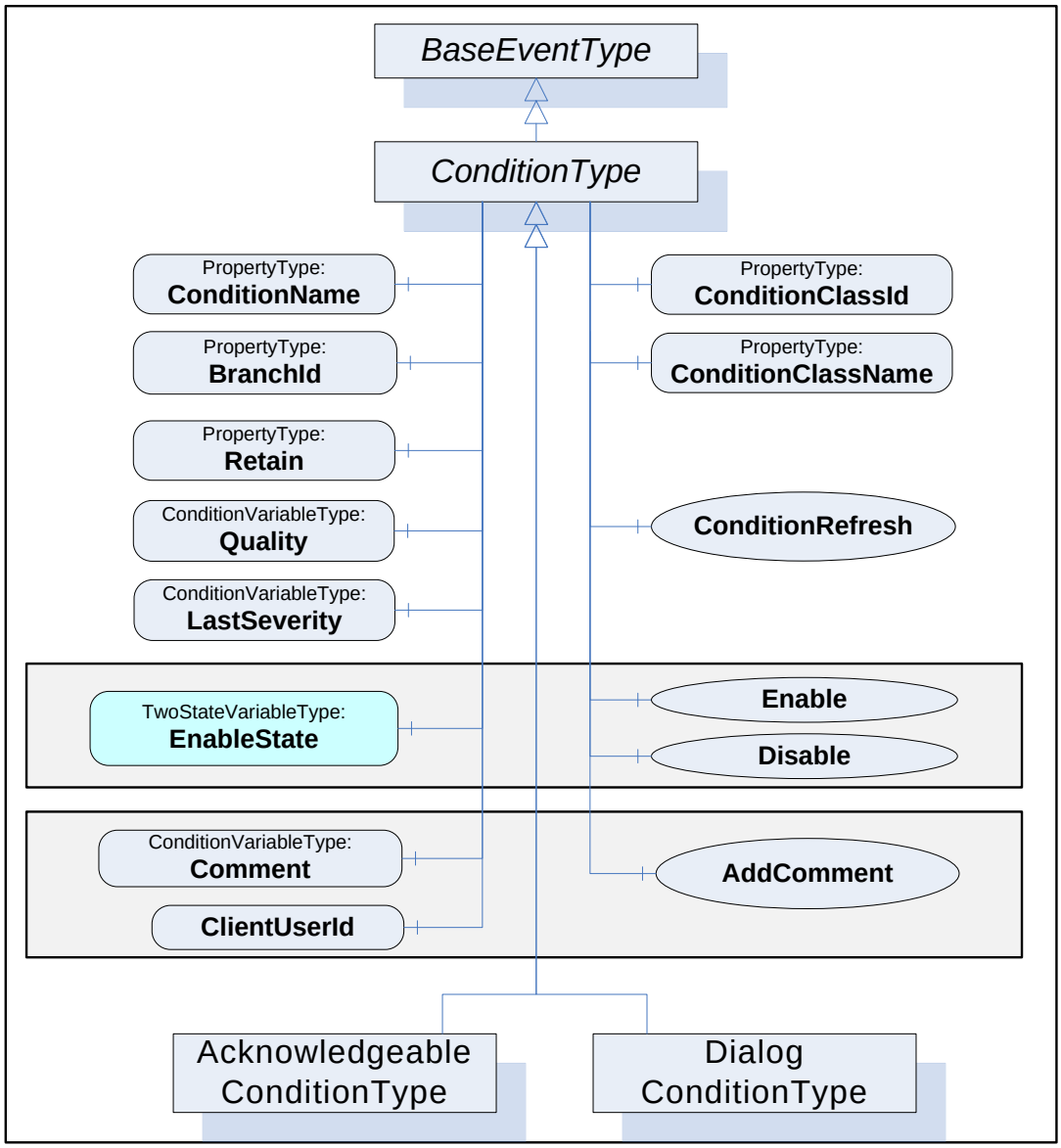

Figure 8 - Condition Model

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 substate machine.

Table 7 – ConditionType Definition

Attribute	Value				
BrowseName	ConditionType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseEventType* defined in Part 5					
HasSubtype	ObjectType	DialogConditionType	Defined in Clause 5.6.2		
HasSubtype	ObjectType	AcknowledgeableConditionType	Defined in Clause 5.7.2		
HasProperty	Variable	ConditionClassId	NodeId	PropertyType	Mandatory
HasProperty	Variable	ConditionClassName	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	ConditionName	String	PropertyType	Mandatory
HasProperty	Variable	BranchId	NodeId	PropertyType	Mandatory
HasProperty	Variable	Retain	Boolean	PropertyType	Mandatory
HasComponent	Variable	EnabledState	LocalizedText	TwoStateVariableType	Mandatory
HasComponent	Variable	Quality	StatusCode	ConditionVariableType	Mandatory
HasComponent	Variable	LastSeverity	UInt16	ConditionVariableType	Mandatory
HasComponent	Variable	Comment	LocalizedText	ConditionVariableType	Mandatory
HasProperty	Variable	ClientUserId	String	PropertyType	Mandatory
HasComponent	Method	Disable	Defined in Clause 5.5.4		Mandatory
HasComponent	Method	Enable	Defined in Clause 5.5.5		Mandatory
HasComponent	Method	AddComment	Defined in Clause 0		Mandatory
HasComponent	Method	ConditionRefresh	Defined in Clause 5.5.6		None

The *ConditionType* inherits all *Properties* of the *BaseEventType*. Their semantic is defined in Part 5. *SourceNode* identifies the *ConditionSource*. See section 5.12 for more details. If the *ConditionSource* is not a *Node* in the *AddressSpace* the *NodeId* is set to null.

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

ConditionClassName provides the display name of the *ConditionClassType*.

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

BranchId is Null for all *Event Notifications* that relate to the current state of the *Condition* instance. If *BranchId* is not Null it identifies a previous state of this *Condition* instance 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. See section 4.4 for more information about the need for creating and maintaining previous *ConditionBranches* and appendix B.1 for an example using branches. Note that a *NodeId DataType* is used as the identifier although the *Server* is not required to have *ConditionBranches* in the *Address Space*. The use of a *NodeId* allows the *Server* to use simple numeric identifiers, strings or blobs.

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

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

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

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

- When the *Condition* instance enters the disabled state, the *Retain Property* of this *Condition* shall be set to FALSE by the *Server* to indicate to the *Client* that the *Condition* instance is currently not of interest to *Client*s.
- When the *Condition* instance enters the enabled state, the *Condition* shall be evaluated and all of its *Properties* updated to reflect the current values. If this evaluation causes the *Retain Property* to transition to TRUE for any *ConditionBranch*, then an *Event Notification* shall be generated for that *ConditionBranch*.
- The *Server* may choose to continue to test for a *Condition* instance while it is disabled. However, no *Event Notifications* will be generated while the *Condition* instance is disabled.
- For any *Condition* that exists in the *AddressSpace* the *Attributes* and the following *Variables* will continue to have valid values even in the disabled state; EventId, 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

Note that 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 Part 8 (DataAccess) as well as *Good*, *Uncertain* and *Bad* as defined in Part 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 must return *Good*.

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* or some other *Method*. The initial value of this *Variable* is null. 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 Part 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* in the *SelectClause* of the *EventFilter*:

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 variables that belong to the *Condition* instance. If one or more variable values change the *Server* generates an *Event* with a unique *EventId*.

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

A *ConditionBranch* is a copy of the *Condition* instance state that can change independently of the current *Condition* instance state. Each *Branch* has an identifier called a *BranchId* which is unique among all active *Branches* for a *Condition* instance. *Branches* are typically not visible in the *Address Space* and this specification 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 (defined in Call Service)

ResultCode	Description
Bad_ConditionAlreadyDisabled	See Table 53 for the description of this result code.

Table 8 specifies the *AddressSpace* representation for the *Disable Method*.

Table 8 – 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.*

Signature

```
Enable();
```

Method Result Codes (defined in Call Service)

ResultCode	Description
Bad_ConditionAlreadyEnabled	See Table 53 for the description of this result code.

Table 9 specifies the *AddressSpace* representation for the *Enable Method*.

Table 9 – Enable Method AddressSpace Definition

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

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
   );
```

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 (defined in Call Service)

ResultCode	Description
Bad_MethodInvalid	See Part 4 for the description of this result code. The addressed Condition does not support adding comments.
Bad_EventIdUnknown	See Table 53 for the description of this result code.
Bad_NodeIdUnknown	See Part 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 10 specifies the *AddressSpace* representation for the *AddComment Method*.

Table 10 – 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	

5.5.6 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*.

Signature

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

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

Method Result Codes (defined in Call Service)

ResultCode	**Description**
Bad_SubscriptionIdInvalid	See Part 4 for the description of this result code
Bad_RefreshInProgress	See Table 53 for the description of this result code

Comments

Clause 4.3 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 section 5.11.2) marking the start of *Refresh.* A copy of the *RefreshStartEvent* is queued into the *Event* stream for every *Event 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 must 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* must 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 section 5.11.3) to signal the end of the *Refresh.* A copy of the *RefreshEndEvent* is queued into the Event stream for every

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 11 specifies the *AddressSpace* representation for the *ConditionRefresh Method*.

Table 11 – 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 12.

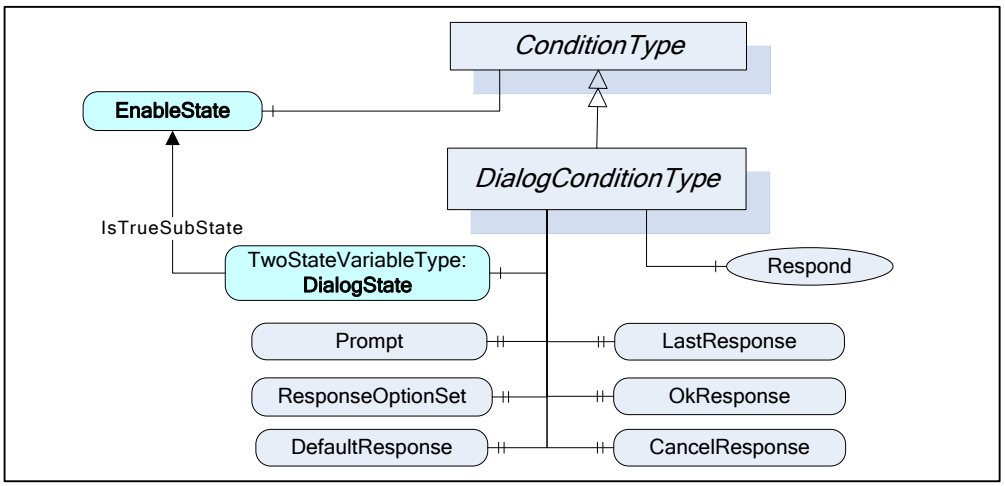

Figure 9 - DialogConditionType Overview

Table 12 – DialogConditionType Definition

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

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

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

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

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

Typical combinations of response options are

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

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

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

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

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

5.6.3 Respond Method

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

Signature

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

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

Method Result Codes (defined in Call Service)

ResultCode	Description
Bad_DialogNotActive	See Table 53 for the description of this result code.
Bad_DialogResponseInvalid	See Table 53 for the description of this result code.

Table 13 specifies the *AddressSpace* representation for the *Respond Method*.

Table 13 – 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 subsections.

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 14.

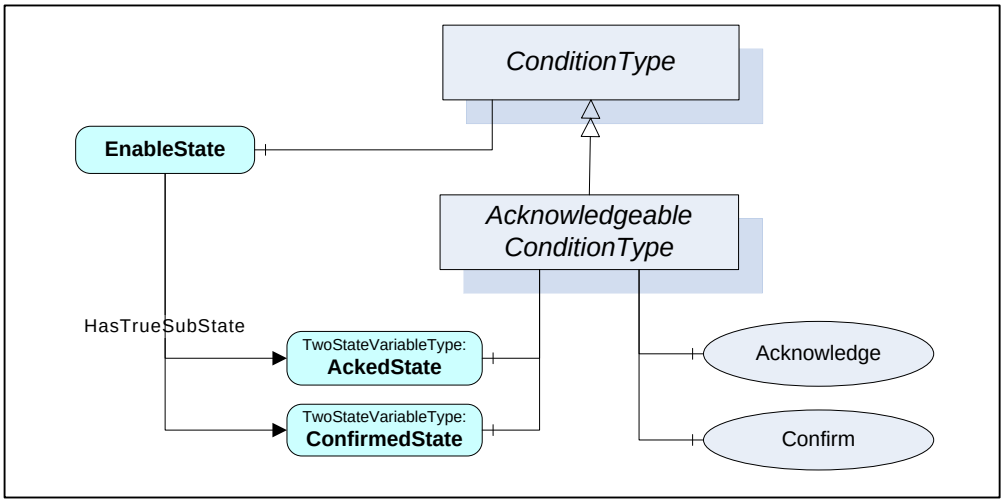

Figure 10 - AcknowledgeableConditionType Overview

Table 14 – AcknowledgeableConditionType Definition

Attribute	**Value**				
BrowseName	AcknowledgeableConditionType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *ConditionType* defined in clause 5.5.2.					
HasSubtype	ObjectType	AlarmConditionType	Defined in Clause 5.8.2		
HasComponent	Variable	AckedState	LocalizedText	TwoStateVariableType	Mandatory
HasComponent	Variable	ConfirmedState	LocalizedText	TwoStateVariableType	Optional
HasComponent	Method	Acknowledge	Defined in Clause 5.7.3		Mandatory
HasComponent	Method	Confirm	Defined in Clause 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 section 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 section 4.4. *Clients* are expected to keep track of all *ConditionBranches* where *AckedState*/Id is FALSE to allow acknowledgement of those. See also section 5.5.2 for more information about *ConditionBranches* and the examples in appendix 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
   );
```

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 (defined in Call Service)

ResultCode	Description
Bad_ConditionBranchAlreadyAcked	See Table 53 for the description of this result code.
Bad_EventIdUnknown	See Table 53 for the description of this result code.
Bad_NodeIdUnknown	See Part 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*.

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 15 specifies the *AddressSpace* representation for the *Acknowledge Method*.

Table 15 – 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
   );
```

Table 16 – 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 (defined in Call Service)

ResultCode	Description
Bad_ConditionBranchAlreadyConfirmed	See Table 53 for the description of this result code.
Bad_EventIdUnknown	See Table 53 for the description of this result code.
Bad_NodeIdUnknown	See Part 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*/Id is set to TRUE and the *Comment Property* contains the text of the optional comment argument. If a previous state is confirmed, the *BranchId* and all Condition values of this branch will be reported.

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

Table 17 – 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			AuditConditionConfirm EventType	

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*..

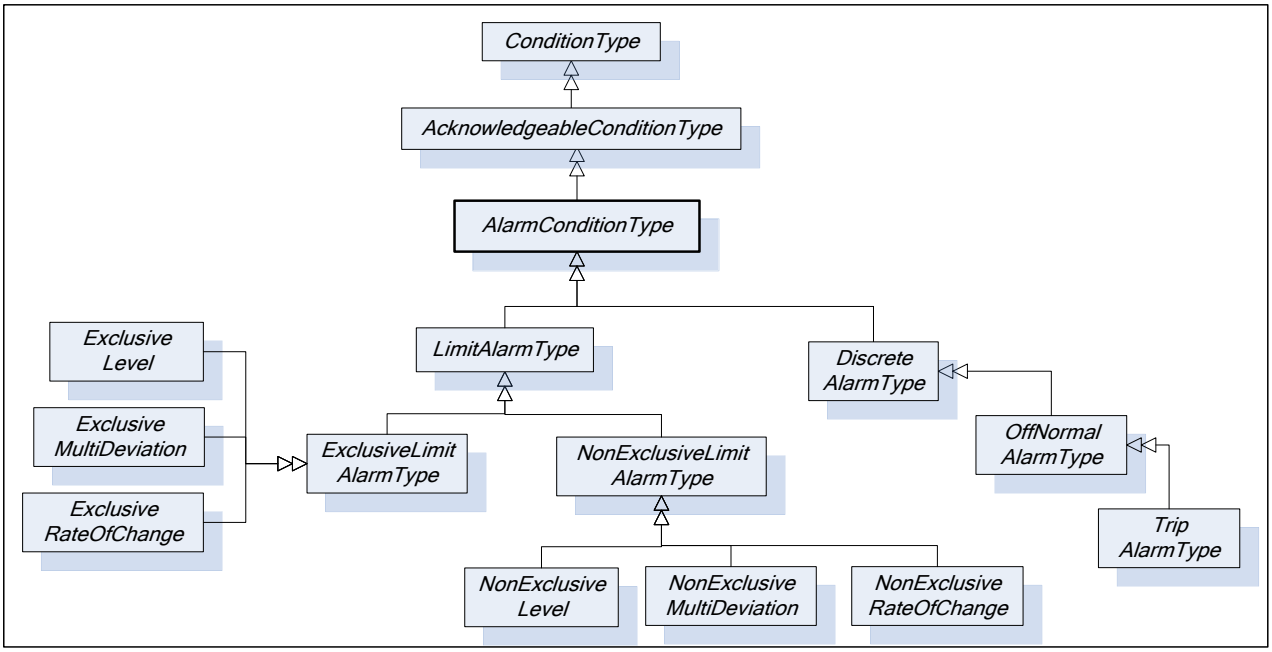

Figure 11 – AlarmConditionType Hierarchy Model

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 18.

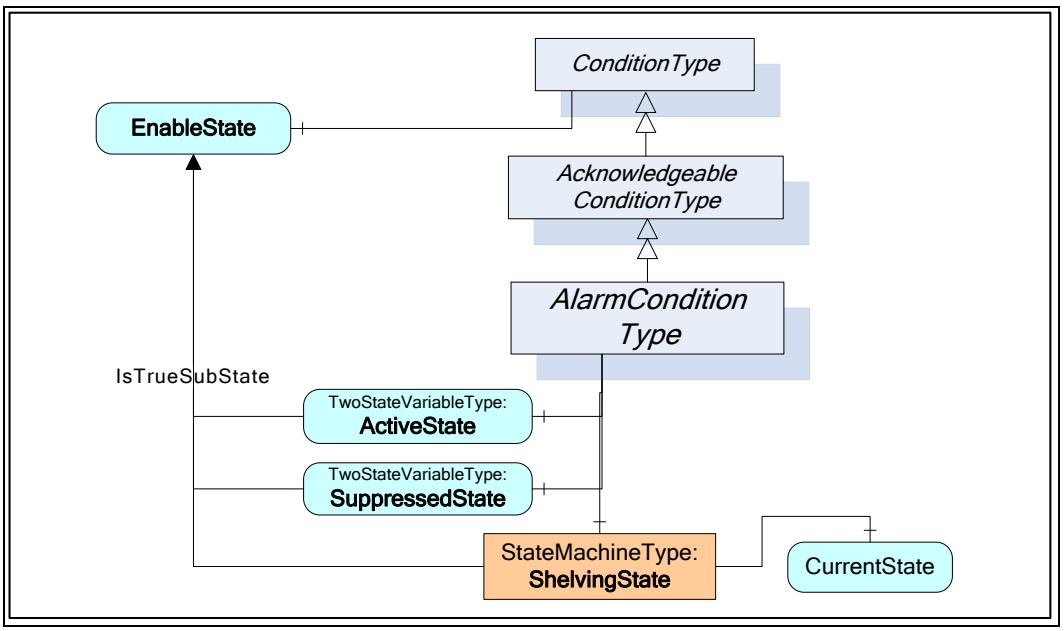

Figure 12 - Alarm Model

Table 18 – AlarmConditionType Definition

Attribute	Value				
BrowseName	AlarmConditionType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the AcknowledgeableConditionType defined in clause 5.7.2					
HasComponent	Variable	ActiveState	LocalizedText	TwoStateVariableType	Mandatory
HasProperty	Variable	InputNode	NodeId	PropertyType	Mandatory
HasComponent	Variable	SuppressedState	LocalizedText	TwoStateVariableType	Optional
HasComponent	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 document 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.

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 the following section.

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 this *Property* is present it will also be enforced for the OneshotShelved state, in that a *Alarm Condition* will transition to Unshelved state from the OneshotShelved state without a transition if the duration specificied in this *Property* expires following a OneShotShelve operation.

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

5.8.3 ShelvedStateMachineType

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 unshelving 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* becomes Inactive.

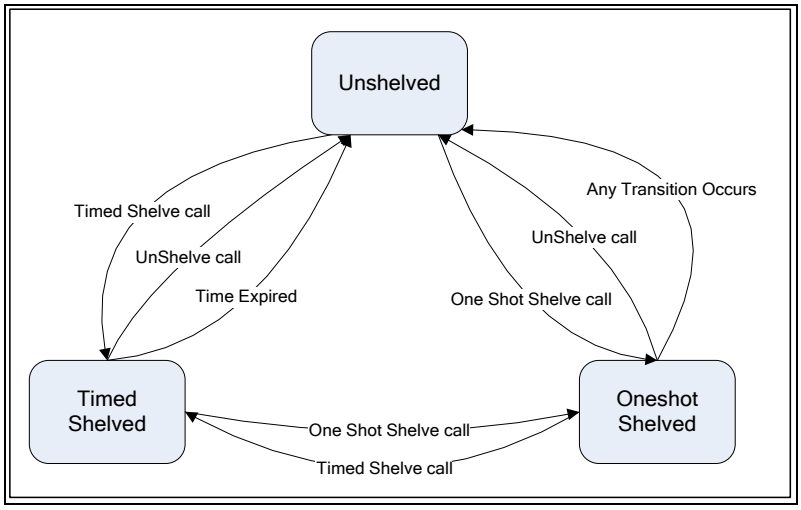

Figure 13 - Shelve state transitions

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 19.

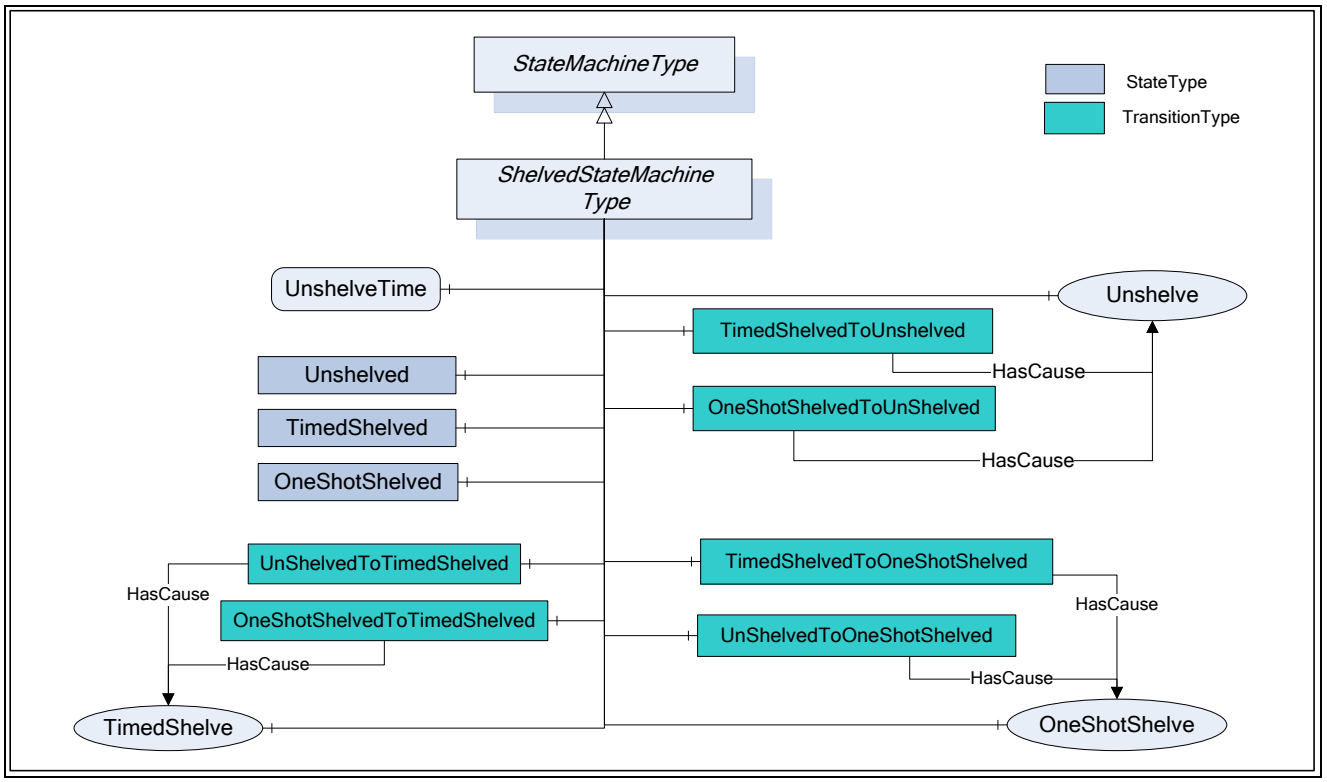

Figure 14 - Shelved State Machine Model

Table 19 –ShelvedStateMachine Definition

Attribute	**Value**				
BrowseName	ShelvedStateMachineType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the Finite*StateMachineType* defined in Part 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 Clause 5.8.5		Mandatory
HasComponent	Method	OneShotShelve	Defined in Clause 5.8.6		Mandatory
HasComponent	Method	Unshelve	Defined in Clause 5.8.4		Mandatory

UnshelveTime specifies the remaining time in Milliseconds until the TimedShelved state or for the cases where a *MaxTimeShelved* Property is provided, the OneshotShelved state will automatically transition into the UnShelved state

This *StateMachine* supports three active states; Unshelved, TimedShelved and OneShotShelved. It also supports six transitions. The states and transitions are described in Table 20. This *StateMachine* also supports three *Methods*; TimedShelve, OneShotShelve and UnShelve.

Table 20 – ShelvedStateMachine Transitions

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
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*	

5.8.4 Unshelve Method

Unshelve sets the *AlarmCondition* to the Unshelved state.

Signature

```
Unshelve( );
```

Method Result Codes (defined in Call Service)

ResultCode	Description
Bad_ConditionNotShelved	See Table 53 for the description of this result code.

Table 21 specifies the *AddressSpace* representation for the *Unshelve Method*.

Table 21 – Unshelve Method AddressSpace Definition

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

5.8.5 TimedShelve Method

TimedShelve sets the *AlarmCondition* to the TimedShelved state.

Signature

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

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 condition then the shelving time must be less then or equal to the value of this property.

Method Result Codes (defined in Call Service)

ResultCode	Description
Bad_ConditionAlreadyShelved	See Table 53 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 53 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. The limit may be exposed as the *MaxTimeShelved Property*.

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

Table 22 – 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	

5.8.6 OneShotShelve Method

OneShotShelve sets the *AlarmCondition* to the OneShotShelved state.

Signature

```
OneShotShelve( );
```

Method Result Codes (defined in Call Service)

ResultCode	Description
Bad_AlreadyShelved	See Table 53 for the description of this result code. The Alarm is already in OneShotShelved state.

Table 23 specifies the *AddressSpace* representation for the *OneShotShelve Method*.

Table 23 – OneShotShelve Method AddressSpace Definition

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

5.8.7 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 *Alarm Conditions* with multiple limits. The *LimitAlarmType* is illustrated in Figure 15.

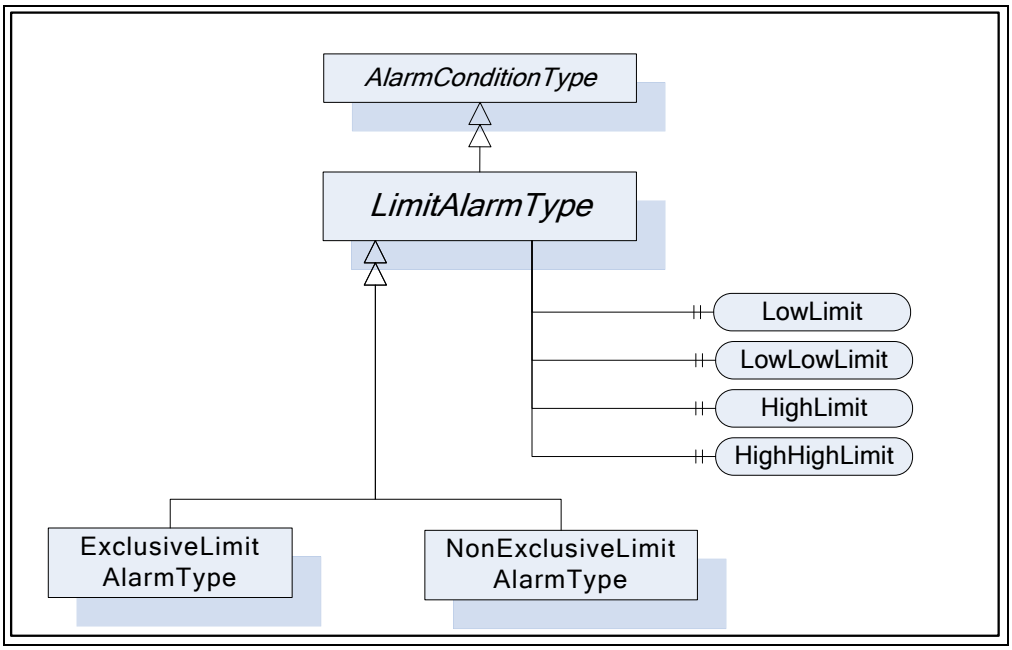

Figure 15 **- LimitAlarmType**

The *LimitAlarmType* is formally defined in Table 24.

Table 24 –LimitAlarmType Definition

Attribute	**Value**				
BrowseName	LimitAlarmType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the AlarmConditionType defined in clause 5.8.2.					
HasSubtype	ObjectType	ExclusiveLimitAlarmType	Defined in Clause 5.8.8.3		
HasSubtype	ObjectType	NonExclusiveLimitAlarmType	Defined in Clause 5.8.9		
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 can not 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 with regards to equals the limit is server specific.

5.8.8 ExclusiveLimit Types

5.8.8.1 Overview

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

5.8.8.2 ExclusiveLimitStateMachineType

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

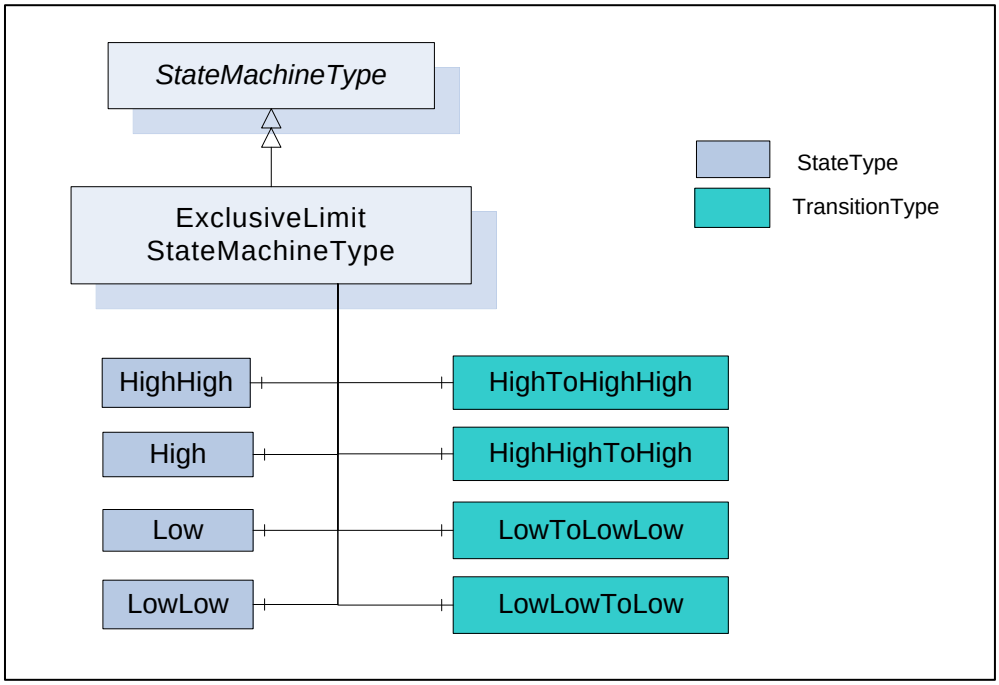

Figure 16 - ExclusiveLimitStateMachine

It is created by extending the *StateMachineType*. It is formally defined in Table 25 and the state transitions are described in Table 26.

Table 25 – ExclusiveLimitStateMachineType Definition

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

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
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 substate machine that represents the actual level of a multilevel *Alarm* when it is in the active state. The substate machine defined here includes high, low, highhigh and lowlow states. This model also includes in its transition state a series of transition to and from a parent state, the inactive state. This state machine as it is defined must be used as a substate machine for a state machine which has an active state. This active state could be part of a “level” Alarm or “deviation” alarm or any other alarm state machine.

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

5.8.8.3 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.

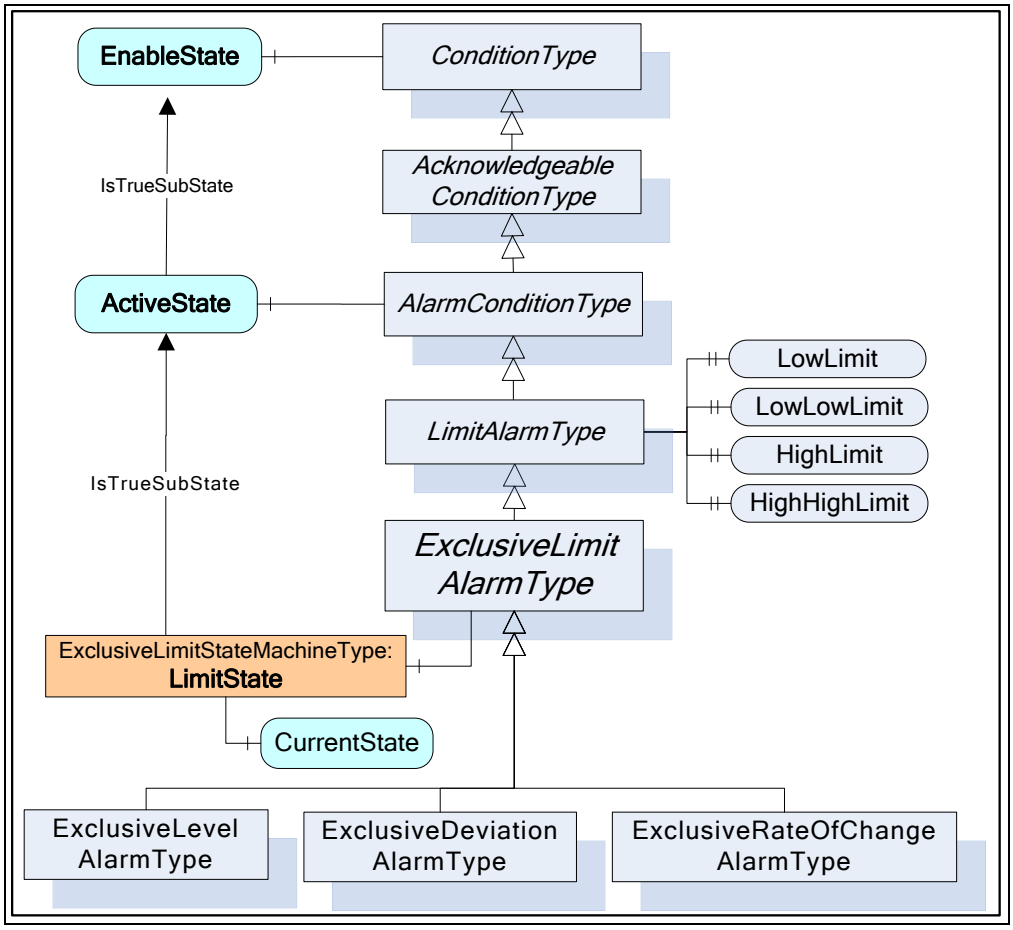

Figure 17 - ExclusiveLimitAlarmType

The *ExclusiveLimitAlarmType* is formally defined in Table 27.

Table 27 – ExclusiveLimitAlarmType Definition

Attribute	Value				
BrowseName	ExclusiveLimitAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the LimitAlarmType defined in clause 5.8.7.					
HasSubtype	ObjectType	ExclusiveLevelAlarmType	Defined in Clause 5.8.10.3		
HasSubtype	ObjectType	ExclusiveDeviationAlarm Type	Defined in Clause 5.8.11.3		
HasSubtype	ObjectType	ExclusiveRateOfChange AlarmType	Defined in Clause 5.8.12.3		
HasComponent	Object	LimitState		*ExclusiveLimitStateMachineType*	Mandatory

LimitState represents the actual limit violation of an *ExclusiveLimitAlarm* when it is in the active state.

5.8.9 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.

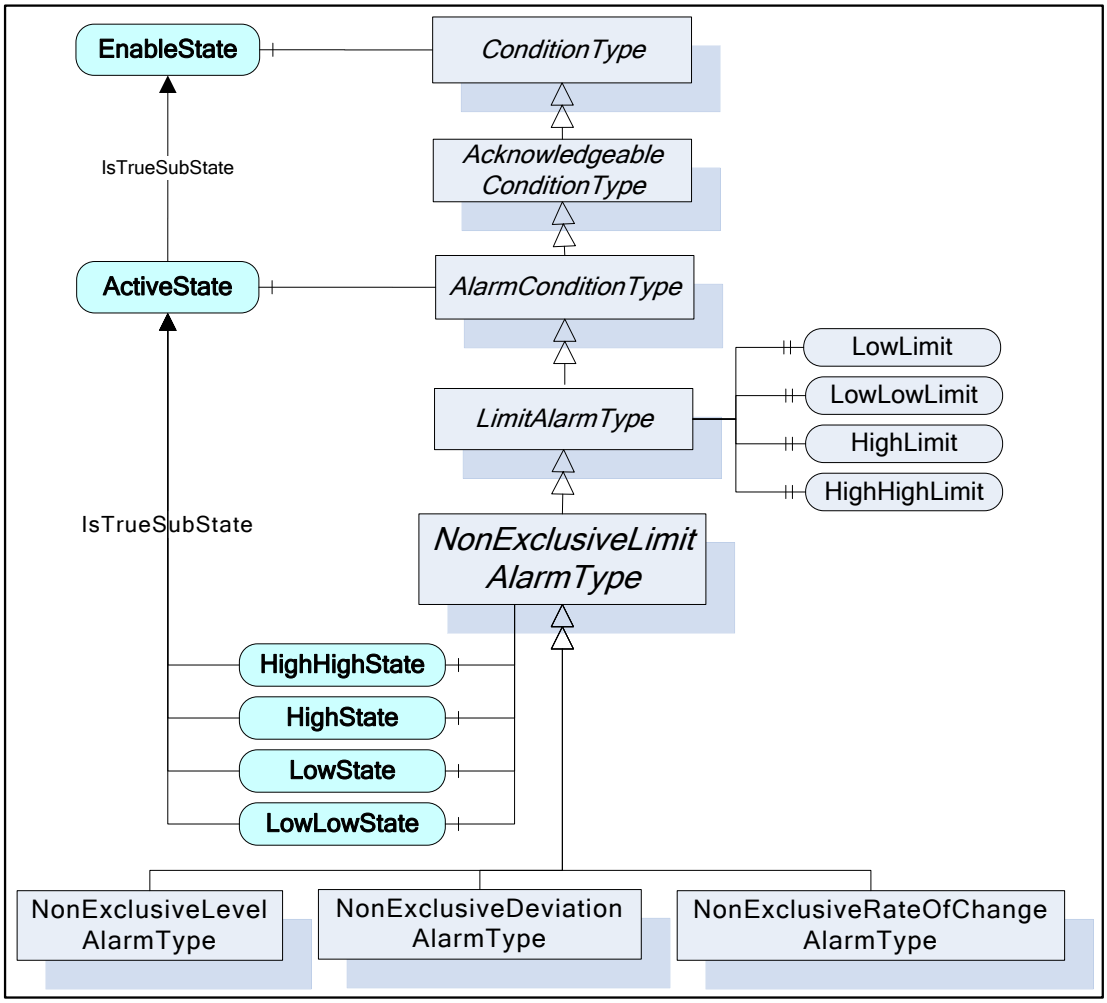

Figure 18 **- NonExclusiveLimitAlarmType**

The *NonExclusiveLimitAlarmType* is formally defined in Table 28.

Table 28 – NonExclusiveLimitAlarmType Definition

Attribute	**Value**				
BrowseName	NonExclusiveLimitAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the LimitAlarmType defined in clause 5.8.7.					
HasSubtype	ObjectType	NonExclusiveLevelAlarmType	Defined in Clause 5.8.10.2		
HasSubtype	ObjectType	NonExclusiveDeviationAlarmType	Defined in Clause 5.8.11.2		
HasSubtype	ObjectType	NonExclusiveRateOfChange AlarmType	Defined in Clause 5.8.12.2		
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 can not exceed both a HighState value and a LowState value simultaneously.

5.8.10 Level Alarm

5.8.10.1 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.

5.8.10.2 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 29.

Table 29 – NonExclusiveLevelAlarmType Definition

Attribute	Value				
BrowseName	NonExclusiveLevelAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the NonExclusiveLimitAlarmType defined in clause 5.8.9.					

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

5.8.10.3 ExclusiveLevelAlarmType

The *ExclusiveLevelAlarmType* is a special level *Alarm* utilized with multiple mutually exclusive limits. It is formally defined in Table 30.

Table 30 – ExclusiveLevelAlarmType Definition

Attribute	Value				
BrowseName	ExclusiveLevelAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Inherits the Properties of the ExclusiveLimitAlarmType defined in clause 5.8.8.3.					

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

5.8.11 Deviation Alarm

5.8.11.1 Overview

A deviation *Alarm* is commonly used to report an excess deviation between a desired setpoint 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 setpoint 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 substate is entered if the process value dropped to below 9; the high substate is entered if the process value became larger than 12. If the setpoint were changed to 11 then the new deviation values would be 10 and 13 respectively.

5.8.11.2 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 31.

Table 31 – NonExclusiveDeviationAlarmType Definition

Attribute	**Value**				
BrowseName	NonExclusiveDeviationAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the NonExclusiveLimitAlarmType defined in clause 5.8.9.					
HasProperty	Variable	SetpointNode	NodeId	PropertyType	Mandatory

The *SetpointNode Property* provides the *NodeId* of the setpoint used in the deviation calculation. If this *Variable* is not in the *AddressSpace*, a Null *NodeId* shall be provided.

5.8.11.3 ExclusiveDeviationAlarmType

The *ExclusiveDeviationAlarmType* is utilized with multiple mutually exclusive limits. It is formally defined in Table 32.

Table 32 – ExclusiveDeviationAlarmType Definition

Attribute	**Value**				
BrowseName	ExclusiveDeviationAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Inherits the *Properties* of the *ExclusiveLimitAlarmType* defined in clause 5.8.8.3.					
HasProperty	Variable	SetpointNode	NodeId	PropertyType	Mandatory

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

5.8.12 Rate of Change

5.8.12.1 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 meters per minute. If the tank level changes at a rate that is greater than 4 meters per minute then the High substate is entered.

5.8.12.2 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 33.

Table 33 – NonExclusiveRateOfChangeAlarmType Definition

Attribute	Value				
BrowseName	NonExclusiveRateOfChangeAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the NonExclusiveLimitAlarmType defined in clause 5.8.9.					

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

5.8.12.3 ExclusiveRateOfChangeAlarmType

ExclusiveRateOfChangeAlarmType is utilized with multiple mutually exclusive limits. It is formally defined in Table 34.

Table 34 – ExclusiveRateOfChangeAlarmType Definition

Attribute	Value				
BrowseName	ExclusiveRateOfChangeAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherits the *Properties* of the *ExclusiveLimitAlarmType* defined in clause 5.8.8.3.					

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

5.8.13 Discrete Alarms

5.8.13.1 DiscreteAlarmType

The *DiscreteAlarmType* is an abstract type 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 specification is illustrated in Figure 19. It is formally defined in Table 35.

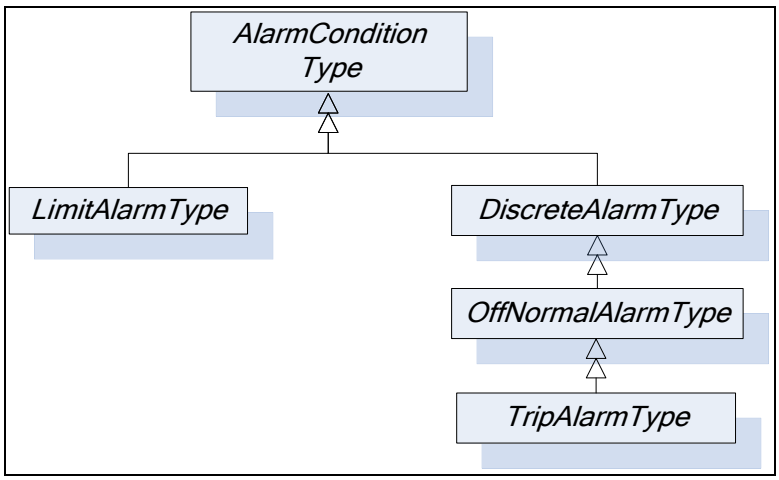

Figure 19 **- DiscreteAlarmType Hierarchy**

Table 35 – DiscreteAlarmType Definition

Attribute	**Value**				
BrowseName	DiscreteAlarmType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the AlarmConditionType defined in clause 5.8.2.					
HasSubtype	*ObjectType*	OffNormalAlarmType	Defined in Clause 5.8.11		

5.8.13.2 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 36. 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. This *Type* is mainly used for categorization. No additional properties are defined.

Table 36 – OffNormalAlarmType Definition

Attribute	**Value**				
BrowseName	OffNormalAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the DiscreteAlarmType defined in clause 5.8.13.1					
HasSubtype	*ObjectType*	TripAlarmType	Defined in Clause 5.8.13.3		
HasProperty	*Variable*	NormalState	NodeId	PropertyType	Mandatory

The *NormalState Property* indicates which one of the possible input values indicates the normal state. 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.

5.8.13.3 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 37. This *Type* is mainly used for categorization.

Table 37 – TripAlarmType Definition

Attribute	Value				
BrowseName	TripAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the OffNormalAlarmType defined in clause 5.8.13.2.					

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 7 informally describes the hierarchy of *ConditionClass Types* defined in this specification.

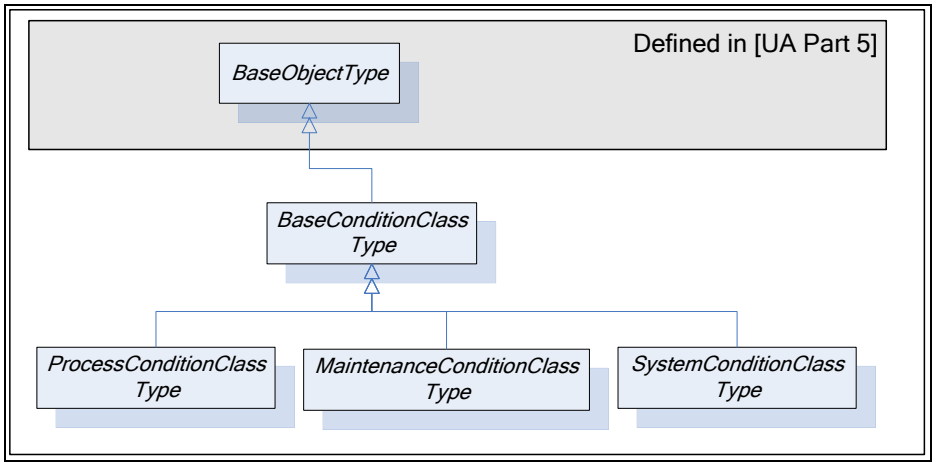

Figure 20 - ConditionClass Type Hierarchy

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

5.9.1.1 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 38.

Table 38 – BaseConditionClassType Definition

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

5.9.1.2 ProcessConditionClassType

The *ProcessConditionClassType* is used to classify *Conditions* related to the process itself. It is formally defined in Table 39.

Table 39 – ProcessConditionClassType Definition

Attribute	Value				
BrowseName	ProcessConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the BaseConditionClassType defined in clause 5.9.1.1.					

5.9.1.3 MaintenanceConditionClassType

The *MaintenanceConditionClassType* is used to classify *Conditions* related to maintenance. It is formally defined in Table 40. No further definition is provided here. It is expected that other standards groups will define domain-specific sub-types.

Table 40 – MaintenanceConditionClassType Definition

Attribute	Value				
BrowseName	MaintenanceConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the BaseConditionClassType defined in clause 5.9.1.1.					

5.9.1.4 SystemConditionClassType

The *SystemConditionClassType* is used to classify *Conditions* related to the System. It is formally defined in Table 41. System *Conditions* occur in the controlling or monitoring system process. No further definition is provided here. It is expected that other standards groups or vendors will define domain-specific sub-types.

Table 41 – SystemConditionClassType Definition

Attribute	Value				
BrowseName	SystemConditionClassType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the BaseConditionClassType defined in clause 5.9.1.1.					

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 document. They are illustrated in Figure 21.

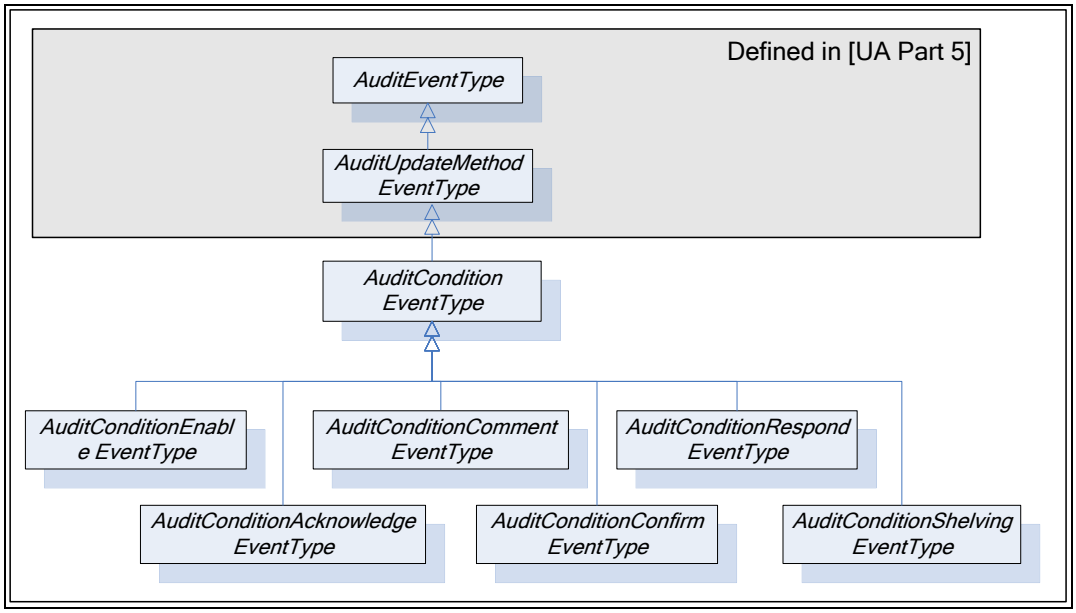

Figure 21 – AuditEvent Hierarchy

Audit Condition EventTypes inherit all *Properties* of the *AuditUpdateMethodEventType* defined in Part 5. The inherited *Property SourceNode* must be filled with the *ConditionId*. The *SourceName* shall be "Method/" and the name of the *Service* that generated the *Event* (e.g. *Disable* or *Acknowledge*).

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.

5.10.2 AuditConditionEventType

This *EventType* is used to subsume all *Audit Condition EventTypes*. It is formally defined in Table 43.

Table 42 – AuditConditionEnableEventType Definition

Attribute	Value
BrowseName	AuditConditionEventType
IsAbstract	True

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 43.

Table 43 – AuditConditionEnableEventType Definition

Attribute	Value
BrowseName	AuditConditionEnableEventType
IsAbstract	True

5.10.4 AuditConditionCommentEventType

This *EventType* is used to report an *AddComment* action. It is formally defined in Table 44.

Table 44 – AuditConditionCommentEventType Definition

Attribute	Value
BrowseName	AuditConditionCommentEventType
IsAbstract	True

5.10.5 AuditConditionRespondEventType

This *EventType* is used to report a *Respond* action. It is formally defined in Table 45.

Table 45 – AuditConditionRespondEventType Definition

Attribute	Value
BrowseName	AuditConditionRespondEventType
IsAbstract	True

5.10.6 AuditConditionAcknowledgeEventType

This *EventType* is used to indicate acknowledgement or confirmation of one or more *Conditions*. It is formally defined in Table 46.

Table 46 – AuditConditionAcknowledgeEventType Definition

Attribute	Value
BrowseName	AuditConditionAcknowledgeEventType
IsAbstract	True

5.10.7 AuditConditionConfirmEventType

This *EventType* is used to report a *Confirm* action. It is formally defined in Table 47.

Table 47 – AuditConditionConfirmEventType Definition

Attribute	Value
BrowseName	AuditConditionConfirmEventType
IsAbstract	True

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 48.

Table 48 – AuditConditionShelvingEventType Definition

Attribute	Value
BrowseName	AuditConditionShelvingEventType
IsAbstract	True

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.

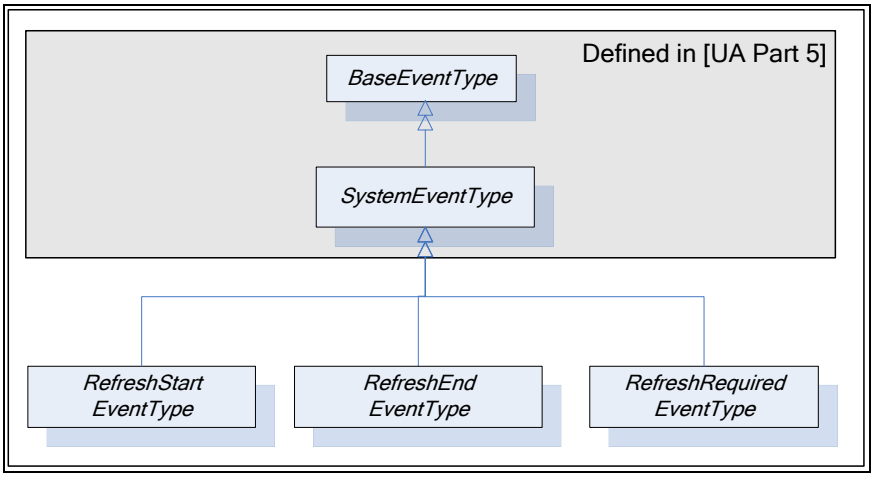

Figure 22 – Refresh Related Event Hierarchy

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 49.

Table 49 – RefreshStartEventType Definition

Attribute	Value				
BrowseName	RefreshStartEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherit the *Properties* of the *SystemEventType* defined in Part 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 50.

Table 50 – RefreshEndEventType Definition

Attribute	Value				
BrowseName	RefreshEndEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherit the *Properties* of the *SystemEventType* defined in Part 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 51.

Table 51 – RefreshRequiredEventType Definition

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

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 52.

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 appendix 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 52 – HasCondition ReferenceType

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

5.13 Alarm & Condition Status Codes

Table 53 defines the *StatusCodes* defined for Alarm & Conditions.

Table 53 – Alarm & Condition Result Codes

Symbolic Id	Description
Bad_ConditionAlreadyEnabled	The addressed Condition is already enabled.
Bad_ConditionAlreadyDisabled	The addressed Condition is already disabled.
Bad_ConditionAlreadyShelved	The Alarm is already in a shelved state.
Bad_ConditionBranchAlreadyAcked	The EventId does not refer to a state that needs acknowledgement.
Bad_ConditionBranchAlreadyConfirmed	The EventId does not refer to a state that needs confirmation.
Bad_ConditionNotShelved	The Alarm is not in the requested shelved state.
Bad_DialogNotActive	The *DialogCondition* 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

6 AddressSpace Organisation

6.1 General

The *AddressSpace* organisation described in this section allows *Clients* to detect *Conditions* and *ConditionSources*. An additional hierarchy of *Object Nodes* that are notifiers may be established to define one or more areas; the *Client* can subscribe to specific areas to limit the *Event Notifications* sent by the *Server*. Additional examples can be found in appendix 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 specification. 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 Part 3 for the formal definition of these *Reference Types*.

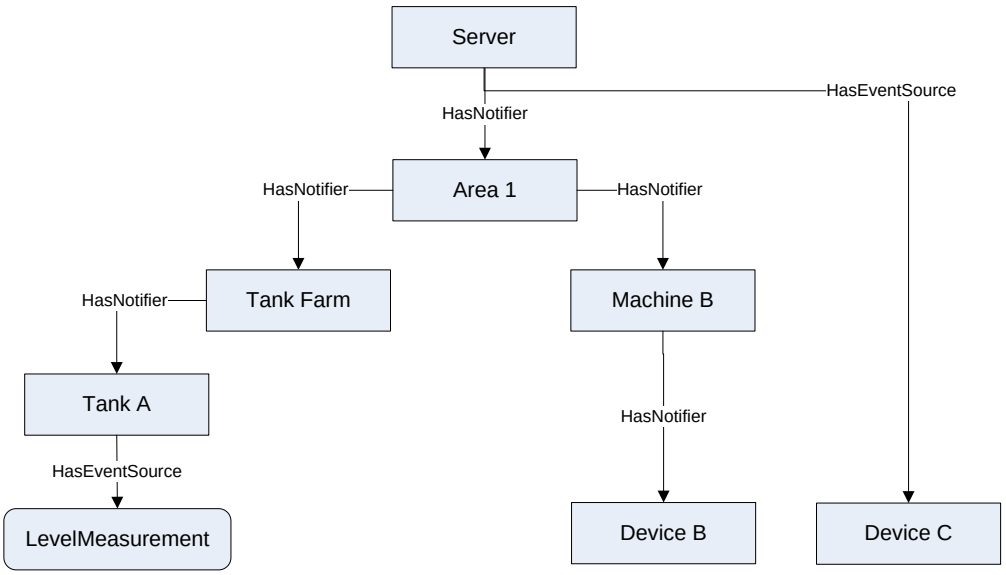

Figure 23 – Typical Event Hierarchy

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*.

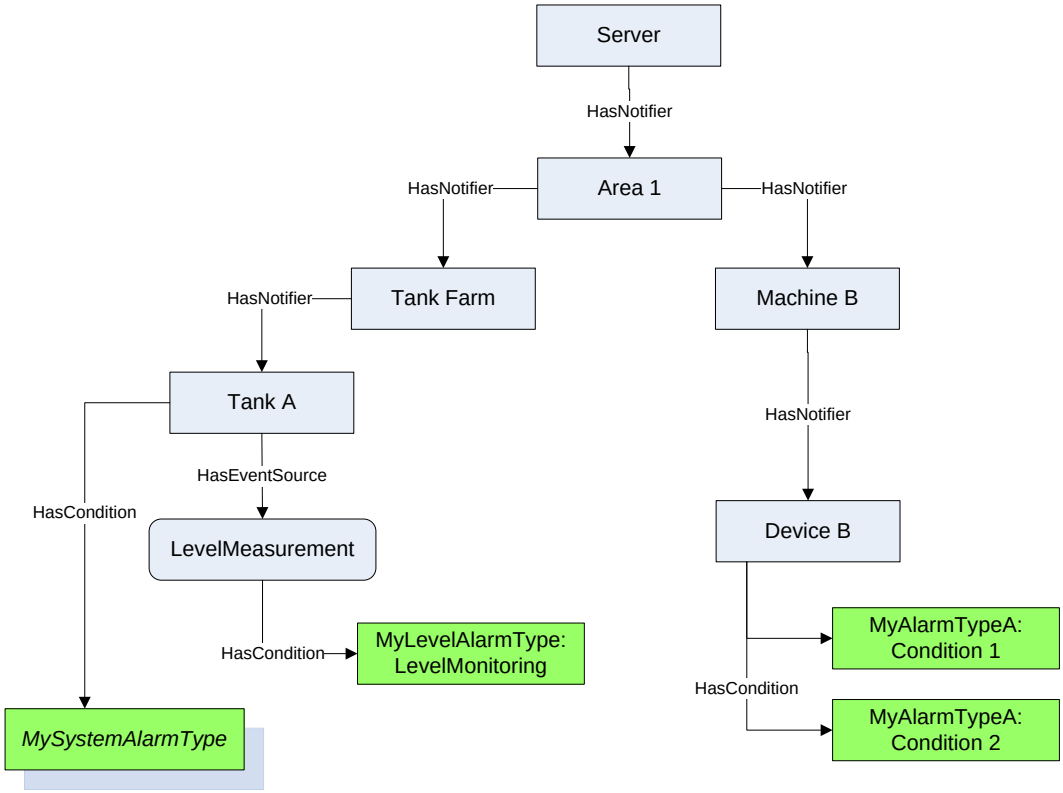

Figure 24 – Use of HasCondition in an Event Hierarchy

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.

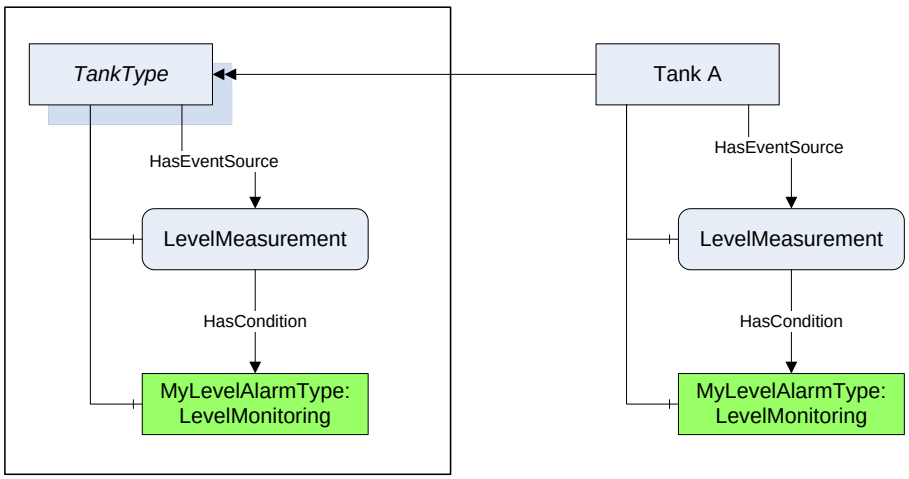

Figure 25 – Use of HasCondition in an InstanceDeclaration

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*.

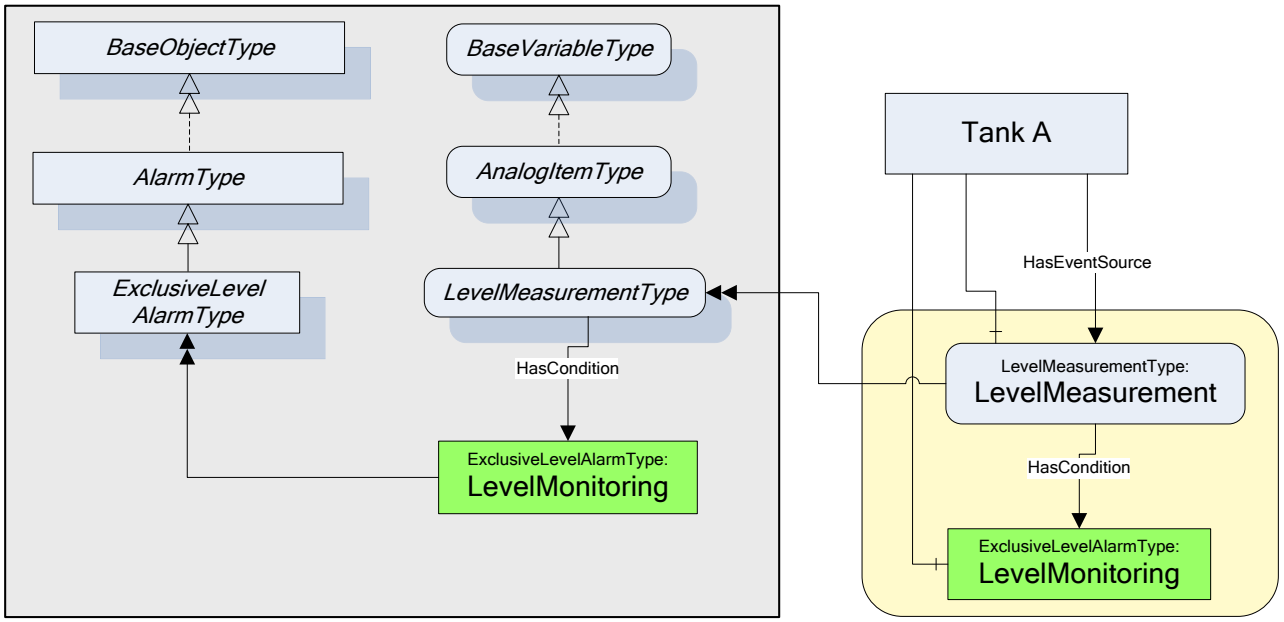

Figure 26 – Use of HasCondition in a VariableType

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 54 and Table 55

Table 54 – 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 55 – 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 56 and Table 57.

Table 56 – 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 57 – 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.2 Recommended Dialog Response Options

The recommended *Dialog* response option names in different locales are listed in Table 58.

Table 58 – Recommended Dialog Response Options

Locale “en”	Locale “de”	
Ok	OK	
Cancel	Abbrechen	
Yes	Ja	
No	Nein	
Abort	Abbrechen	
Retry	Wiederholen	
Ignore	Ignorieren	
Next	Nächster	
Previous	Vorheriger	

Annex B (informative): Examples

B.1 Examples for Event sequences from Condition instances

B.1.1 Introduction

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 27 shows an Alarm as it becomes active and then inactive and also the acknowledgement and confirmation cycles. Table 59 lists the values of the state *Variables*. All *Events* are coming from the same *Condition* instance and therefore have the same *ConditionId*.

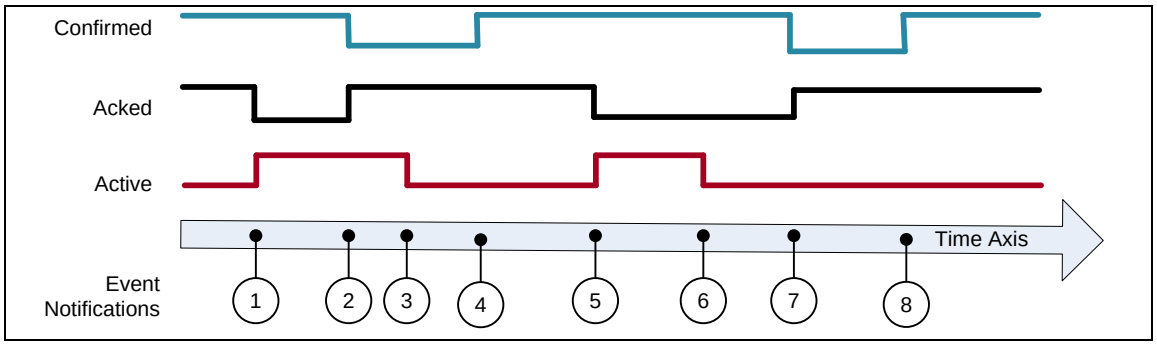

Figure 27 – Single State Example

Table 59 – 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 28 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 inactive state. Table 60 lists the values of the state *Variables*. All *Events* are coming from the same *Condition* instance and have therefore the same *ConditionId*.

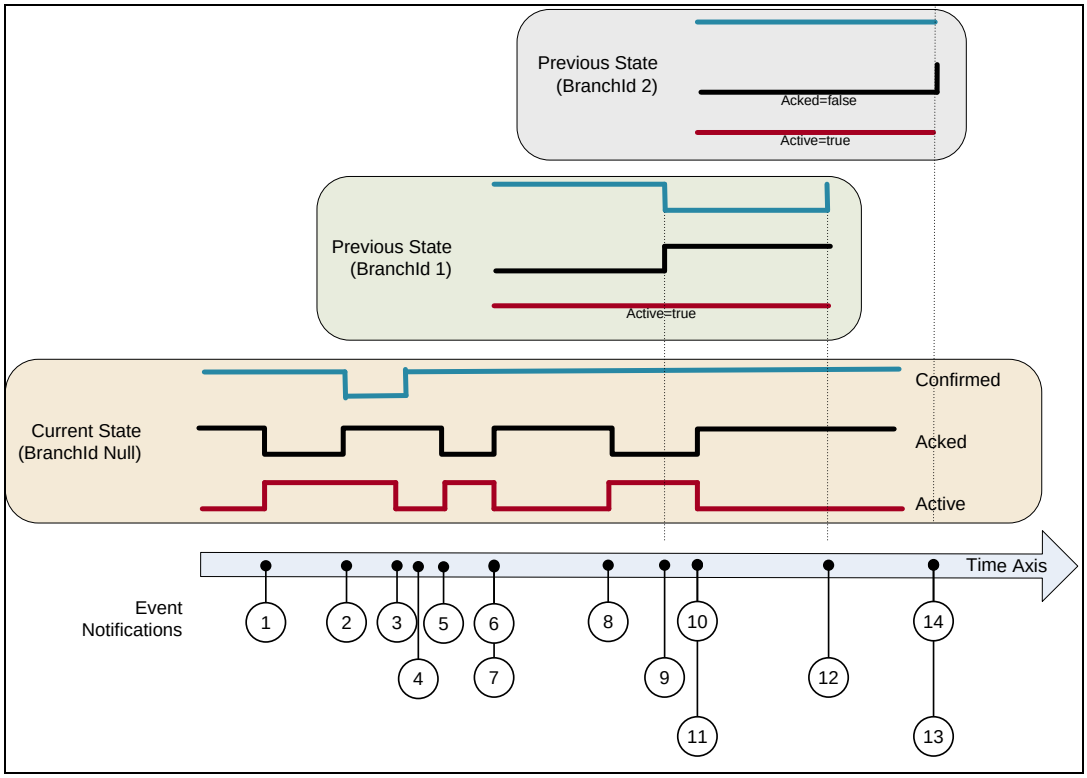

Figure 28 – Previous State Example

Table 60 – Example of a *Condition* that maintains previous states via branches

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	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 **)	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 **)	Alarm goes inactive again.
11	2	True	False	True	True	Prior state needs acknowledgment. Branch #2 created.
12	1	True	True	True	False	Prior state confirmed. Branch #1 deleted.
13	2	True	True	True	False	Prior state acknowledged, Auto Confirmed by system Branch #2 deleted.
14	Null	False	True	True	False	No longer of interest.

*) 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 Events #6 and #7 is identical.

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

A single *Condition* can have many branch states active (Events #11)

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

B.2 Address Space Examples

This section 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 29 illustrates the use of the *HasCondition Reference* with *Condition* instances.

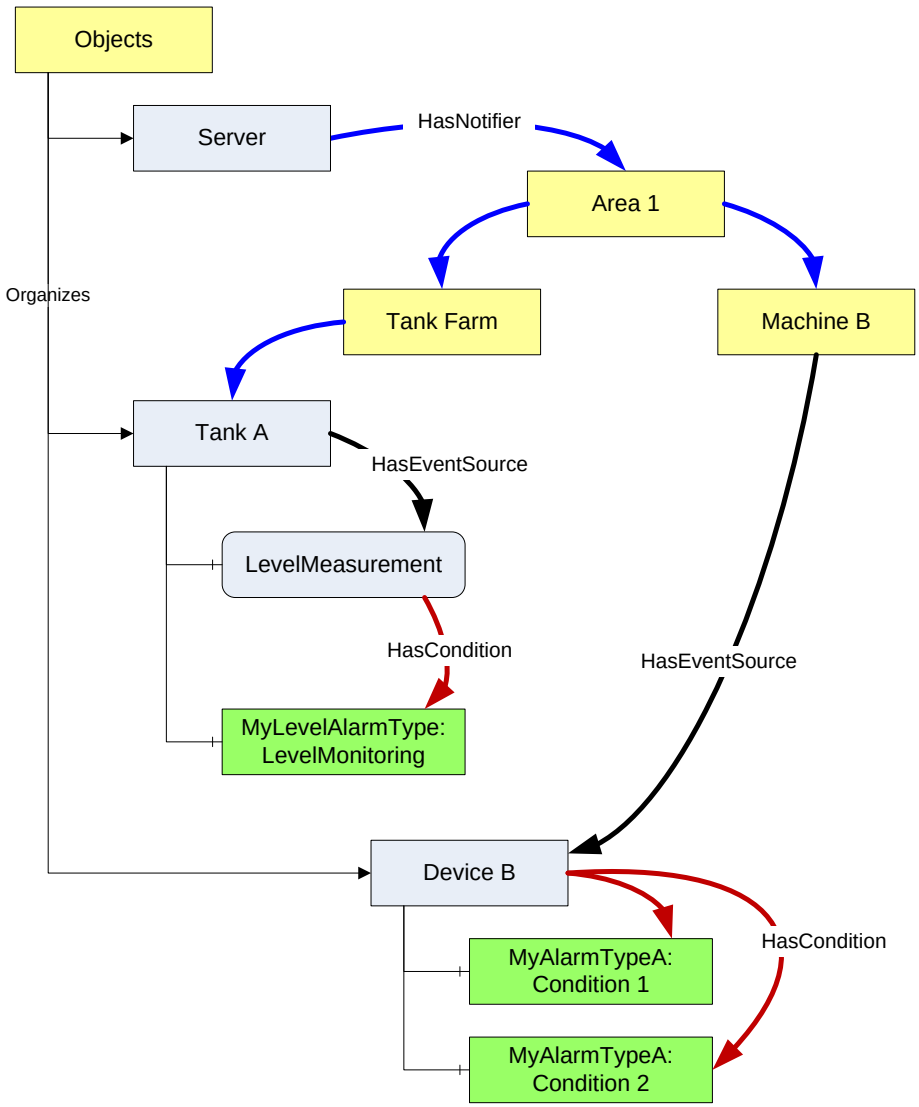

Figure 29 – HasCondition used with Condition instances

In systems where Conditions are not available as instances, the ConditionSource can reference the ConditionTypes instead. This is illustrated with the example in Figure 30.

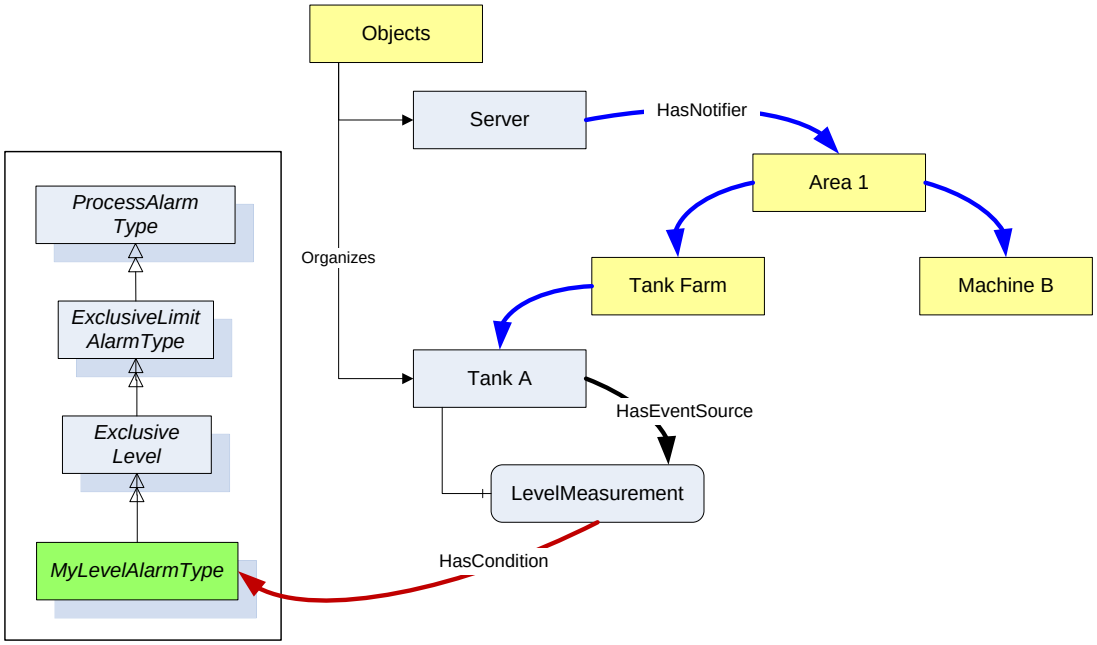

Figure 30 – HasCondition reference to a Condition Type

Figure 31 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

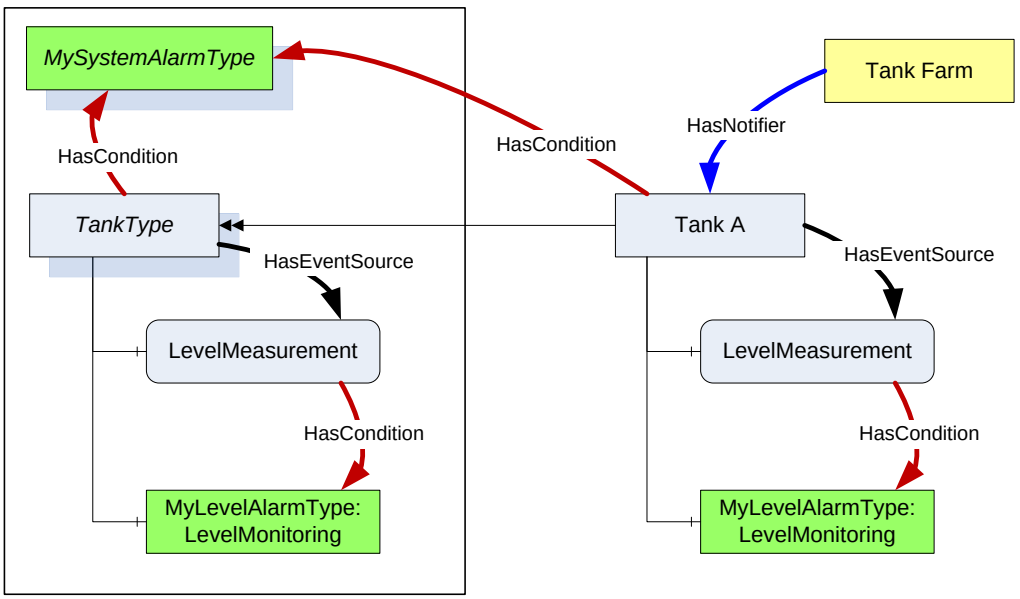

Figure 31 – HasCondition used with an instance declaration

Annex C (informative): Mapping to EEMUA

Table 61 lists EEMUA terms and how OPC UA terms maps to them.

Table 61 –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 Part 5.)	Test information presented to the operator that describes the *Alarm* condition.
Alarm Priority	Severity Property (defined in Part 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 Events 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 Events 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 Events 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 Events 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.1.1 Alarms and Events COM UA Wrapper

D.1.1.1 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 Servers to be wrapped within a single UA Server.

The Area hierarchy is discovered with the BrowseOPCAreas and the GetQualifiedAreaName methods. 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 references 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 Nodes 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.1.1.2 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 methods. The Source name returned by BrowseOPCAreas is used as the BrowseName and DisplayName. The QualifiedSourceName is used to construct the NodeId. Event Source Nodes are always targets of a HasEventSource reference from an Area.

D.1.1.3 Event Categories

Event Categories in the A&E COM Server are represented in the UA Server as ObjectTypes which are subtypes of BaseEventType. The description of the Event Category is used as the BrowseName and DisplayName of the ObjectType Node.

Simple EventTypes are direct subtypes of BaseEventType. Tracking EventTypes are direct subtypes of AuditEventType. Condition EventTypes are subtypes of the AlarmType. The NodeId is constructed from the EventType and the EventCategoryID assigned by the COM AE Server.

Each Condition EventType Category is represented by an ObjectType with the IsAbstract attribute set to True. Each ConditionName belonging to the Event Category is represented by an ObjectType (ConditionName-ObjectType) which is a subtype of the abstract ObjectType used for the Event Category. The BrowseName and DisplayName of the subtype is the ConditionName.

The NodeId for each ConditionName-ObjectType is constructed from the EventType, the EventCategoryID and the ConditionName assigned by the COM AE Server.

ConditionName-ObjectTypes that have Subconditions will have a special Variable with a TypeDefinition of StateVariableType. The BrowseName of this Variable is "SubConditionState" and the DataType of the Id property will be a String. The Id is set to the SubConditionName specified in an Event Notification. The list of Subconditions is not exposed because it is not part of the A&E Event Notification.

Figure 32 illustrates the ObjectType Nodes created from the Event Categories supported by a COM AE Server.

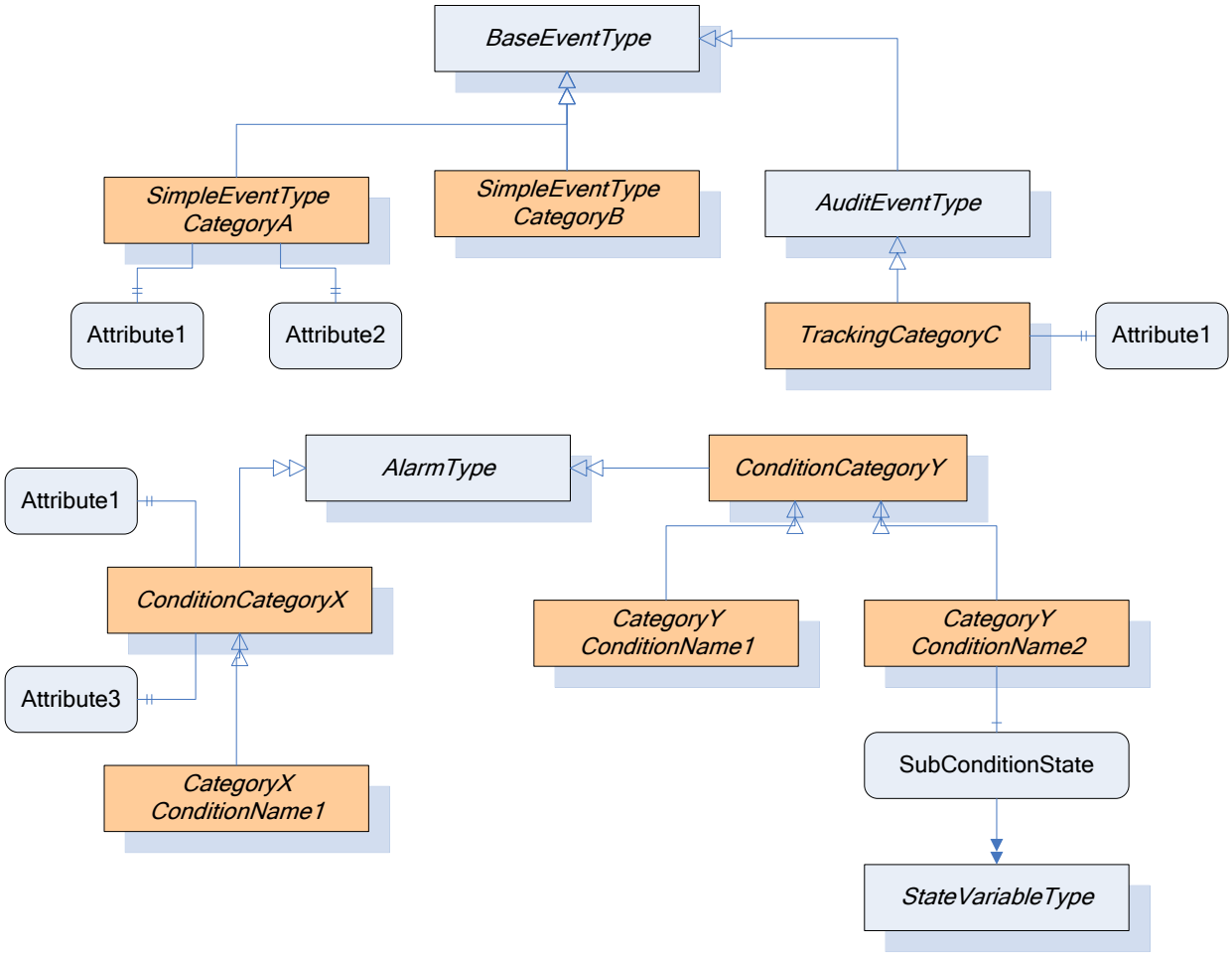

Figure 32 – The Type Model of a Wrapped COM AE Server

D.1.1.4 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 of the Variable is constructed from the AttributeID. The DisplayName is 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 and the AttributeID.

D.1.1.5 Event Subscriptions

The 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 Event Attributes are selected by finding all of the AttributeOperands used in the EventFilters which reference one of the Attributes supported by the Server (the BrowseName of the properties representing the Attributes contain the AttributeID).

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

Table 62 lists how the fields in the ONEVENTSTRUCT that are used by the A&E COM UA Wrapper are mapped to UA BaseEventType Variables.

Table 62 – 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 63 lists how the fields in the ONEVENTSTRUCT that are used by the A&E COM UA Wrapper are mapped to UA AuditEventType Variables.

Table 63 – Mapping from ONEVENTSTRUCT fields to UA AuditEventType Variables

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

Table 64 lists how the fields in the ONEVENTSTRUCT that are used by the A&E COM UA Wrapper are mapped to UA AlarmType Variables.

Table 64 – Mapping from ONEVENTSTRUCT fields to UA AlarmType Variables

UA Event Variable	ONEVENTSTRUCT Field	Notes
ConditionClassId	dwEventType	Always set to BaseConditionClassType
ConditionClassName	dwEventType	Always set to "BaseConditionClass"
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 contructed 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 recieved 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	-
SubConditionState	szSubconditionName	Locale is the default locale for the COM AE Server.
SubConditionState.Id	szSubconditionName	-

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

D.1.1.6 Condition Instances

Condition Instances do not appear in the UA Server address space. Conditions can be acknowledged by passing the EventId to the Acknowledge method defined on the AcknowledgeableConditionType.

Conditions cannot be enabled or disabled via the wrapper.

D.1.1.7 Condition Refresh

The wrapper does not store the state of conditions. When ConditionRefresh is called the Refresh method is called on all COM AE Subcriptions associated with the ConditionRefresh call. The wrapper needs to wait until it receives the callback with the bLastRefresh flag set to True in the OnEvent call before it can tell the UA client that the refresh has completed.

D.1.2 Alarms and Events COM UA Proxy

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.

Sections D.1.2.1 through D.1.2.4 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 interoperabilisty, 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 utilmately 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.1.2.1 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.1.2.1.1 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 33 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.

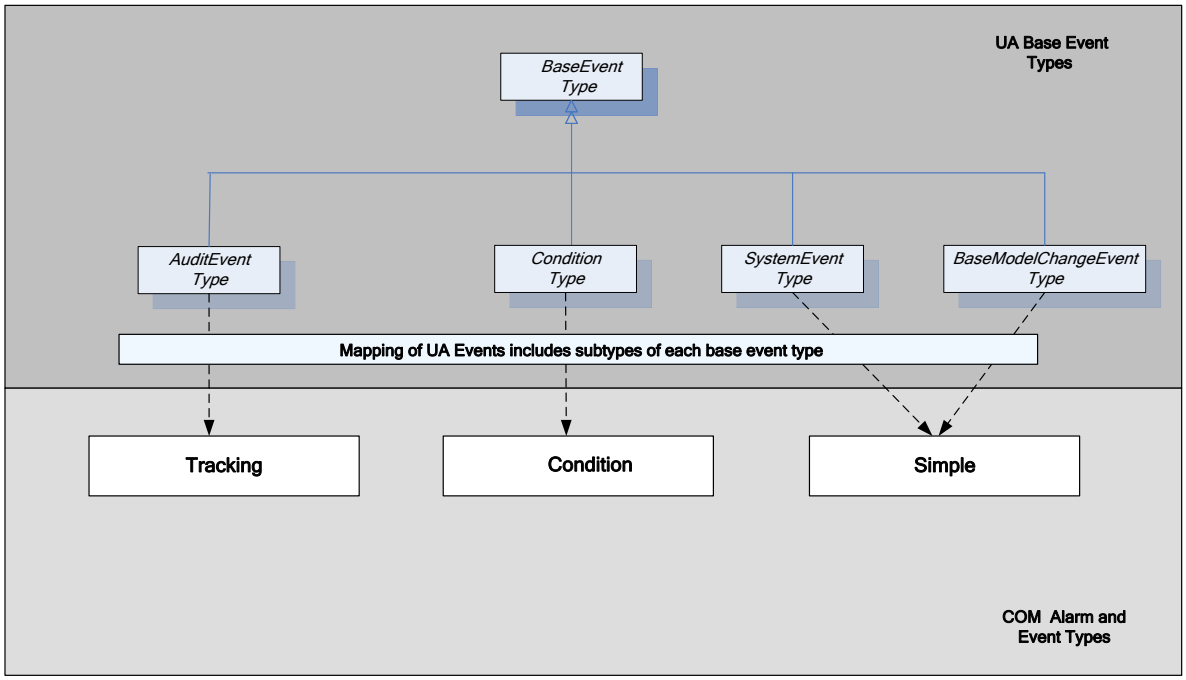

Figure 33 – Mapping UA Event Types to COM A&E Event Types

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

D.1.2.1.2 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 events. 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 34 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.

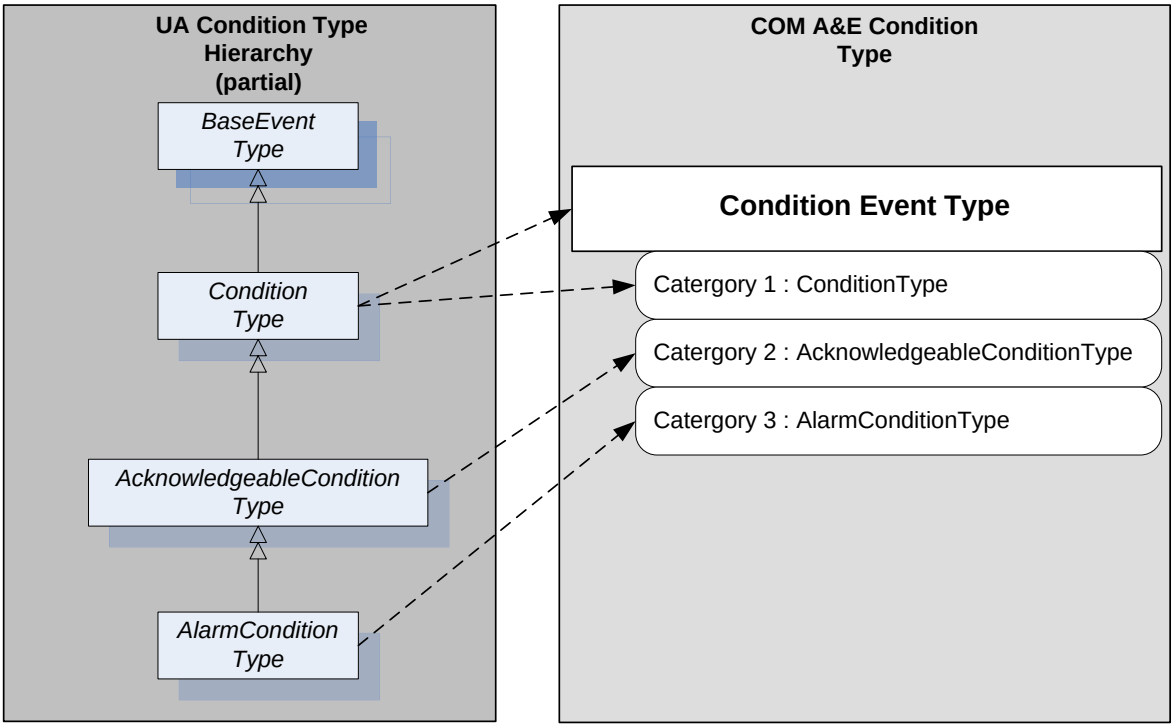

Figure 34 – Example Mapping of UA Event Types to COM A&E Categories

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.1.2.1.3 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 65 lists the attributes currently defined in the ONEVENTSTRUCT in the leftmost column. The rightmost column of Table 65 indicates how the A&E COM UA proxy defines that attribute.

Table 65 – 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 section D.1.2.1.1
dwEventCategory	See section D.1.2.1.2
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	

A&E ONEVENTSTRUCT "attribute"	**A&E COM UA Proxy Mapping**
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 events and those that do not derive from *AlarmConditionType* are set to ACTIVE by default.
wNewState : OPC_CONDITION_ENABLED	A&C *ConditionType* property: EnabledState Note, events mapped as non-condition events are set to ENABLED (state bit mask = 0x1) by default.
wNewState: OPC_CONDITION_ACKED	A&C *AcknowledgeableConditionType* property : AckedState Note that A&C events mapped as non-conditon events or which do not derive from *AcknowledgeableConditionType* are set to UNACKNOWLEDGED and AckRequired = false by default.
wQuality	A&C ConditionType property : Quality Note that events mapped as non-conditon events 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> BadConfigurationError => OPC_QUALITY_CONFIG_ERROR BadNotConnected => OPC_QUALITY_NOT_CONNECTED BadDeviceFailure => OPC_QUALITY_DEVICE_FAILURE BadSensorFailure => OPC_QUALITY_SENSOR_FAILURE BadNoCommunication => OPC_QUALITY_COMM_FAILURE BadOutOfService => OPC_QUALITY_OUT_OF_SERVICE <u>Uncertain status codes</u> UncertainNoCommunicationLastUsableValue => OPC_QUALITY_LAST_USABLE UncertainLastUsableValue => OPC_QUALITY_LAST_USABLE UncertainSensorNotAccurate => OPC_QUALITY_SENSOR_CAL UncertainEngineeringUnitsExceeded => OPC_QUALITY_EGU_EXCEEDED UncertainSubNormal => OPC_QUALITY_SUB_NORMAL <u>Good status codes</u> GoodLocalOverride => 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.
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	

A&E ONEVENTSTRUCT "attribute"	**A&E COM UA Proxy Mapping**
ACK Comment	
AREAS	All A&E events 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. Events 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). Events 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 attributes – 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 clients. 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 clients.

The A&E COM UA Proxy maintains the mapping of attributes on an event category basis. An event category inherits its attributes from the properties defined on all supertypes in the UA Event Type hierarchy. New attributes 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 attributes to each category: AckComment and Areas. Figure 35 shows an example of this mapping.

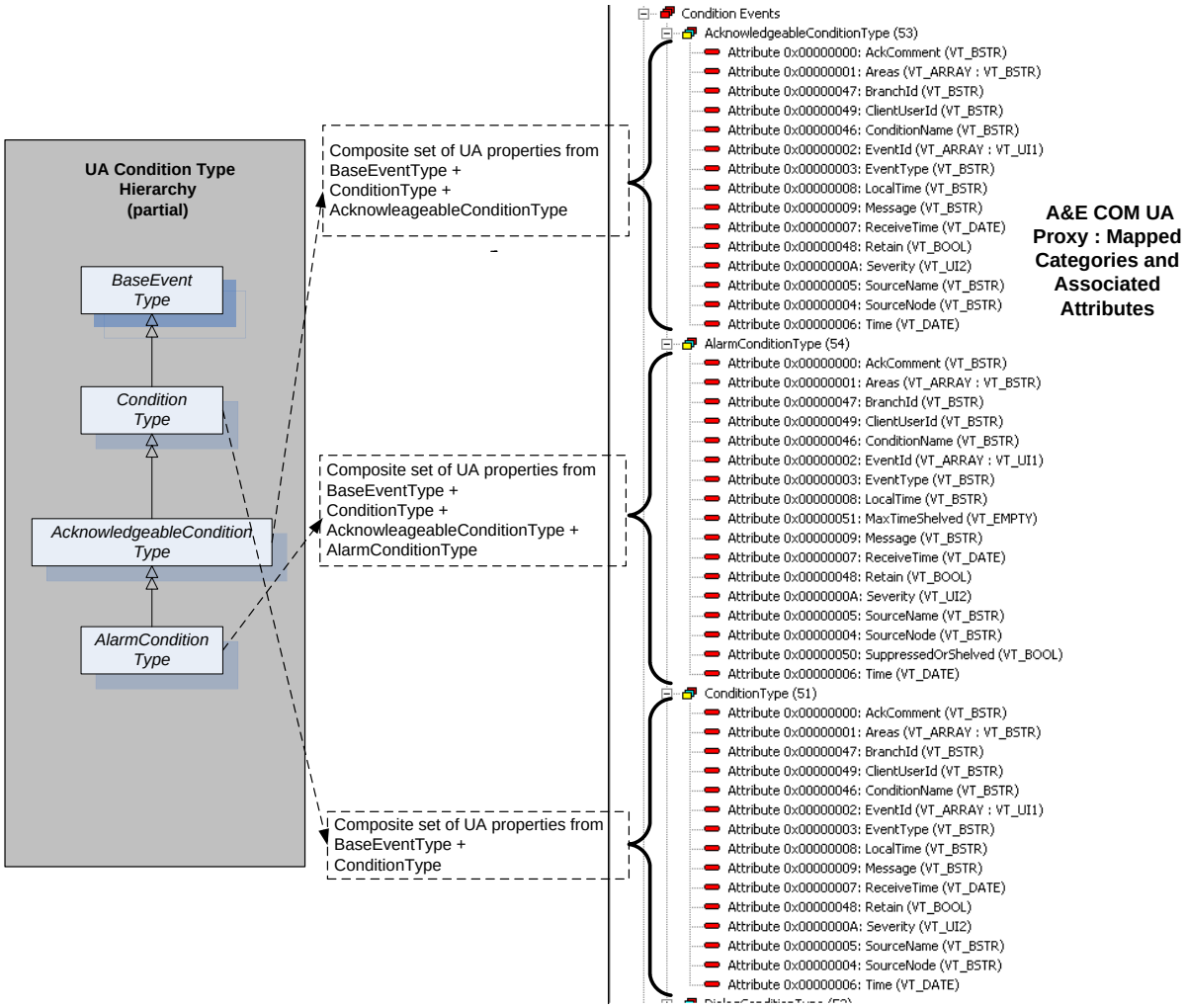

Figure 35 – Example Mapping of UA Event Types to A&E Categories with Attributes

D.1.2.1.4 Event Condition Mapping

Events of any subtype of *ConditionType* are designated COM condition events and are subject to additional processing due to the stateful nature of condition events. COM condition events 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 subtyping, may carry additional information which can be mapped to ACTIVE (INACTIVE) or ACKNOWLEDGED (UNACKNOWLEGED). Condition event processing proceeds as described in Table 65 (see A&E ONEVENTSTRUCT “attribute” rows : OPC_CONDITION_ACTIVE, OPC_CONDITION_ENABLED and OPC_CONDITION_ACKED).

D.1.2.2 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 references are followed and each HasNotifier target whose EventNotifier attribute is set to SubscribeToEvents becomes a nested COM area within the hierarchy.

Note that the HasNotifier target can 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 a A&E COM “source” or leaf in the A&E COM browse tree.

In general, nodes which are the source nodes of the HasEventSource reference and/or are the source nodes of the HasNotifier reference are always A&ECOM Areas. Nodes which are the target nodes of the HasEventSource reference are always A&E COM Sources. Note however that targets of HasEventSource which cannot be found by following the HasNotifier references from the Server object are ignored.

Given the above logic, the A&ECOM UA Proxy browsing will have the following limitations: Only those nodes 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 objects folder) are considered for area designation. Only those nodes 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 objects folder) are considered for area or source designation. To be an area, a node must contain a HasNotifier reference and its EventNotifier attribute must be set to SubscribeToEvents. To be a source, a node must be the target node of a HasEventSource reference and must have been found by following HasNotifier references from the Server object.

D.1.2.3 Qualified Names

D.1.2.3.1 Qualified Name Syntax

From the root of any subtree 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 nodes 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 OPC UA Part 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 OPC UA Part 4 (e.g., “/6:Boiler1/6:Pipe100X/1:Input/2:Measurement“). Relative path indices refer to the namespace table described below.

D.1.2.3.2 Namespace Table

UA server Namespace table indices may vary over time. This represents a problem for those A&E COM clients 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 unwieldly and impractical for use by A&E COM clients. 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 nodes 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.1.2.4 Subscription Filters

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

D.1.2.4.1 Filter by Event, Category or Severity

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

D.1.2.4.2 Filter by Source

In the case of souce filters, the UA A&C Server is free to provide any appropriate, server-specific value for SourceName. There is no expectation that source nodes 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 events from sources which are not discoverable by following only HasNotifier and/or HasEventSource references. 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.1.2.4.3 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 events 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 subscriptions 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 subtree of interest. From this subtree root node, the A&E COM UA Proxy will collect the list of

notifier nodes below this point. The relative path associated with each of the collected notifier nodes in the subtree will be matched against the client supplied relative path containing the wildcard character. A monitored item is created for each notifier node in the subtree 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.