OMAC

OPC 30050

OPC UA for PackML

Release 1.00

2018-04-14

OPC UA Companion-Specification

Standard Type:	OPC UA Information Model For PackML	Comments:	
Title:	OPC Unified Architecture for PackML	Date:	2018-04-14
Version:	Release 1.00	Software:	MS-Word
Editors:	Brandl, Dennis Hunkar, Paul	Source:	OPC 30050 - UA Companion Specification for Packml 1.00.docx
Owner:	OPC UA PackML WG	Status:	Release

Document History

Version	Date	Reason	COMMENTS	Clause/ Subclause	Proposed change
1.00.00	04/14/2018	Initial	Initial Release		
1.00.01	11/15/2018	Format fixes	Minor formatting fixes		

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.

1 Scope

This standard is an extension of the overall OPC Unified Architecture standards and defines an information model that conforms to the PackML object model defined in ISA-TR88.00.02-2015 Machine and Unit States: An Implementation Example of ISA-88, and in the OMAC (Organization for Machine and Automation Control) PackML Unit/Machine Implementation Guide, referred to collectively as PackML. The PackML object model describes a standard way to monitor and control a wide variety of production equipment, following the models defined in the ANSI/ISA 88 standard. ISA-TR88.00.02 has been implemented by manufacturers and machine builders worldwide on various control platforms to increase speed to production, ease line integration and improve reliability. PackML defines standard state models for external control of a piece of equipment, a standard mode model for determining which state model to follow, standard tag names and tag values to command the equipment, determine the status of the equipment, and perform administration of the equipment. The PackML goal is to provide an easy to use and easy to test method of integration of equipment control, into a production line.

The modelling targets of this standard shall exist in an OPC UA *AddressSpace*. This standard does not consider the modelling targets that are identified in other standards or vendor specifications.

2 Normative References

The following referenced documents are indispensable for the application of this standard. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

OPC 10000-1, *OPC Unified Architecture - Part 2: Overview and Concepts*

http://www.opcfoundation.org/UA/Part1/

OPC 10000-2, *OPC Unified Architecture - Part 2: Security Model*

http://www.opcfoundation.org/UA/Part2/

OPC 10000-3, *OPC Unified Architecture - Part 3: Address Space Model*

http://www.opcfoundation.org/UA/Part3/

OPC 10000-4, *OPC Unified Architecture - Part 4: Services*

http://www.opcfoundation.org/UA/Part4/

OPC 10000-5, *OPC Unified Architecture - Part 5: Information Model*

http://www.opcfoundation.org/UA/Part5/

OPC 10000-6, *OPC Unified Architecture - Part 6: Mappings*
http://www.opcfoundation.org/UA/Part6/

OPC 10000-7, *OPC Unified Architecture - Part 7: Profiles*
http://www.opcfoundation.org/UA/Part7/

OPC 10000-8, *OPC Unified Architecture - Part 8: Data Access*
http://www.opcfoundation.org/UA/Part8/

OPC 10000-9, *OPC UA Specification: Part 9 – Alarms & Conditions*
http://www.opcfoundation.org/UA/Part9/

OPC 10000-10, *OPC UA Specification: Part 10 - Programs*
http://www.opcfoundation.org/UA/Part10/

OPC 10000-11, *OPC Unified Architecture - Part 11: Historical Access*
http://www.opcfoundation.org/UA/Part11/

OPC 10000-12, *OPC Unified Architecture - Part 12: Discovery and Global Services*
http://www.opcfoundation.org/UA/Part12/

OPC 10000-13, *OPC Unified Architecture - Part 13: Aggregates*
http://www.opcfoundation.org/UA/Part13/

ISA-TR88.00.02-2015: Machine and Unit States: An implementation example of ANSI/ISA-88.00.01
https://www.isa.org/store/ansi/isa-tr880002-2015,-machine-and-unit-states-an-implementation-example-of-ansi/isa-880001-/43761120

ISA-TR88.00.01-2010: Batch Control Part 1: Models and Terminology
https://www.isa.org/store/products/product-detail/?productId=116649

OPC 30000, UA companion specification for IEC61131-3
https://opcfoundation.org/developer-tools/specifications-unified-architecture/opc-unified-architecture-plcopen-information-model/

3 Terms, definitions and conventions

3.1 Overview

It is assumed that basic concepts of OPC UA information modelling are understood in this specification. This specification will use these concepts to describe the PackML object models. The concepts and terms used to describe the OPC UA information models are defined in other parts and listed in the following sections. Note that OPC UA terms and terms defined in this standard are *italicized* in the specification.

3.2 OPC 10000-1 terms

The following terms defined in OPC 10000-1 apply.

- AddressSpace
- Attribute
- Client
- Message
- Node
- NodeClass
- Object
- ObjectType
- Profile
- Reference
- ReferenceType
- Server
- Variable
- View

3.3 OPC 10000-3 terms

The following terms defined in OPC 10000-3 apply.

- Hierarchical Reference
- InstanceDeclaration
- ModellingRule
- OptionalPlaceholder
- MandatoryPlaceholder
- DataVariable
- Property
- SourceNode
- TargetNode
- TypeDefinitionNode
- VariableType

3.4 OPC 10000-8

The following terms defined in OPC 10000-8 apply.

- AnalogItem
- EngineeringUnits

3.5 OPC 10000-9

The following term defined in OPC 10000-9 apply.

- Operator

3.6 OPC UA PackML Terms

3.6.1 PackML<Term>

3.6.1.1 General

This standard adopts adding a PackML prefix to all PackML defined terms that are used in this standard that are also defined by OPC. The <Term> is the terminology defined by PackML.

This allows the terminology used in PackML to be easily distinguished from terminology defined in OPC UA.

3.6.1.2 PackMLUnit

a concept that defines a collection of associated modules that can carry out one or more major processing activities.

Note: in PackML Unit and Machine are used to define the same thing and can at times be interchanged.

3.6.1.3 PackMLTag

a concept that represents named data element that is used to command, read status, or provide administration of a unit.

3.6.1.4 PackMLStateModel

a concept that represents a state model of the operational state of a unit.

3.6.1.5 PackMLMode

a concept that represents the mode of operation of a unit, indicating which state model is currently active.

3.7 Abbreviations and symbols

DA	Data Access
AC	Alarm and Condition
HA	Historical access
PLC	Programmable Logic Controllers
DCS	Distributed Control Systems
OCS	Open Control Systems

3.8 OPC UA Notation

This standard uses the *ModellingRules OptionalPlaceholder* and *MandatoryPlaceholder* to define instance declarations, and defines a rule that the BrowseName of instance declarations having an *OptionalPlaceholder* or *MandatoryPlaceholder ModellingRule* be enclosed in angle brackets (<>). Originally, this rule is defined in OPC 10000-3 as a recommendation. This naming rule is also used in the description of a table. The *BrowseName* of a *Node* that has *OptionalPlaceholder* or *MandatoryPlaceholder ModellingRule* are described with angle brackets, which denotes that the name is not fixed. For example, *BrowseName* of *Property* is described as <PropertyName> in graphical notation and tables results in a *Property* that can have any name.

4 Concept

4.1 Overview

When the ISA-TR88.00.01-2010 standard was applied to applications across a plant, there was a need to align the terminologies, models and key definitions between different process types: continuous, batch, and discrete processes. Discrete processes involve machines found in the packaging, converting, and material handling applications. The operation of these machines is typically defined by the OEM, system integrator, end user, or is industry specific.

OMAC (Organization for Machine Automation and Control) created a task group with members from technology providers, OEMs, system integrators, and end users to generate the PackML guidelines as a method to show how the ISA-TR88.00.01-2010 concepts could be extended into packaging machinery and other types of machines typically used in assembly lines, filling lines, and other production lines.

The purpose of PackML is to:

- Define a standard state-based model for automated machines.
- Identify definitions for common terminology.
- Explain to practitioners how to use state programming for automated machines.
- Provide references to actual implementation examples and templates from automation and control vendors.
- Identify a common tag structure for automated machines in order to:
  - Provide for “connect & pack” functionality
  - Provide functional interoperability and a consistent look and feel across the plant floor.
  - Provide consistent tag structure for connection to plant MES and enterprise systems.

If automated machinery is modelled in an ISA-TR88.00.01-2010 physical hierarchy, the example mapping shown in Figure 1 is possible, from ISA-TR88.00.02-2015. The example in this document will assume that a machine can represent the unit level in the ISA88 hierarchy.

Figure 1 – Automated Machines using the ISA 88 Models

In the figure, the OPC UA interface and the PackML model might exist at the machine level. The communication and interactions below this level are machine specific. Some machines might have multiple Units which communicate using OPC UA / PackML, but they might also only expose the Machine using the OPC UA PackML interface to other Machines in packaging line or to the Packaging line controller.

4.2 PackML Summary

4.2.1 Introduction

For an OPC UA user that may not be familiar with PackML, the following section provides a brief overview of key features that PackML provides along with a little background related to PackML and the concepts behind it.

4.2.2 Why PackML?

The Organization for Machine Automation and Control (OMAC) was formed to help manufacturers work together to find new and innovative ways to be successful in their production operations. OMAC brought together leading manufacturers representing End-User Manufacturers, OEM Machine Builders, System Integrators, Technology Providers, and Non-Profit / Government Agency organizations to address issues that confront global manufacturing today. OMAC aims to collectively derive common solutions for both technical and non-technical issues in the development, implementation, and commercialization of open, modular architecture control technologies.

Manufacturing systems are made of collections of equipment, often from multiple suppliers, usually each with its own specific and custom interface. In order to make this collection of equipment operate together as a complete system, there is an integration effort required, and it is often time consuming and custom for each supplier.

PackML stands for Packaging Machine Language and is an interface standard originally used in batch manufacturing in the packaging industry but which is now used in multiple different types of production and assembly lines. The primary objective of PackML is to bring a common “look and feel” and operational consistency to all machines that make up a production line. PackML provides:

- Standard defined machine states and operational flow
- Overall Equipment Effectiveness (OEE) data
- Root Cause Analysis (RCA) data
- Flexible recipe schemes and common SCADA or MES inputs

PackML has been implemented in multiple formats for different industrial networks, with a proven benefit of reducing the integration time for adding new equipment to existing lines, or installing new lines.

4.2.3 PackML Elements

In order to provide a standard interface PackML defines three elements:

1) PackML Unit Modes - A standard model that is used to control which state is being used (Producing, Maintenance, Manual, ….).
2) PackML StateMachine - Standard state machine models that are used to represent the internal operational state of the machine/unit. [note: StateMachine may change for Units and for the Mode of the unit]
3) PackTags -A standard set of tag names and extension used to control the mode and state, send commands to the machine/unit, and monitor the status of the machine/unit.

These three aspects will be translated to OPC UA models

4.2.4 Standard Modes

A Unit can be in different modes, for example Producing, Maintenance, Manual, Clean, Calibration, etc. A Unit control mode is an ordered subset of states and commands that determines the strategy carried out by the Unit process, as shown in Figure 2. For example, the producing mode is used when the unit is producing, a manual mode may be used when the unit is being manually controlled for troubleshooting.

The states that a unit can be in depends on the mode. In the producing mode there is a state called SUSPENDED, where the equipment is not running due to an external event, but this state is not available in maintenance mode, neither is the COMPLETE state.

Figure 2 – Mode Management of States

PackML includes a standard manner of changing modes as well as displaying the current mode. For additional information please see the ISA-TR88.00.02-2015.

4.2.5 Standard States

PackML Interface State Model is used to visualize and control the state of a unit/machine. The PackML Interface State Model is a state model that represents the Unit/Machine State in a standardized manner. The interface description is based on a state model, a state description and related control commands. For additional information see ISA-TR88.00.02-2015.

4.2.6 Standard Tag Names

At its core, PackML is the definition of standard tag names and standard values for the tags. These are used to control the state model of the unit (command tags), determine the state and status of the equipment (status tags), and administer the equipment (admin tags).

4.2.7 PackML Object Model

4.2.7.1 Overview

The PackML object model is composed of a series of tags. These tags can be one of three general type of tags Command Tags, Status Tags and Administrative Tags.

4.2.7.2 Command Tags

Command tags allow interaction with the state machine and general functionality of the server. Command tags include changing units, changing state machines. For additional detail see ISA-TR88.00.02-2015. In OPC UA command tags are generally mapped to *Methods*.

4.2.7.3 Status Tags

Status tags provide information about the state of the machine or device. This includes feedback from the commands issued and the general status. For additional detail see ISA-TR88.00.02-2015.

4.2.7.4 Admin Tags

Admin tags provide information about alarming in the machines or device. This include Alarm history and some summary statistics about the machine or device. For additional detail see ISA-TR88.00.02-2015.

4.2.8 Standard Tag Values

4.2.8.1 Overview

Several PackTags have specific values defined.

4.2.8.2 Machine Speed

This describes the set point for the current speed of the unit/machine in primary packages per minute. Keeping speed in a primary package unit of measure (UOM) allows for easier control integration. The primary package UOM is the normalized rate for the machine, normalized to a value chosen on the line.

The following example is for a bottle line running at balance line speed of 1000 packages/minute. The UOM chosen is equivalent to be the actual count of the Filler, or Labeler.

	Actual Pack Counts	Primary Packages (UOM)
Bulk Depalletizer	41.667 (24 pack => 1000/24 = 41.667)	1,000
Filler	1,000	1,000
Labeler	1,000	1,000
Packer	66.667 (15 pack => 1000/15 = 66.667)	1,000
Bulk Depalletizer	41.667 (24 pack => 1000/24 = 41.667)	1,000

4.2.8.3 Material Interlock

Indicates materials are ready for processing. It is comprised of a series of bits with 1 equalling ready or not low, 0 equalling not ready, or low. Each bit represents a different user material. The word contains bits that indicate when a critical material or process parameter is ready for use. It can also be used for production, and/or indication of low condition. This information may be sent to the unit machine at any time as the interlock information changes.

The format and meaning of the material interlock bits are determined by the machine/unit supplier, as shown in the example below:

	Material Interlock Bit #	Material Description
Filler	0	500 ml Bag
Filler	1	Flacked Cereal
Labeler	0	Small Box
Labeler	1	500 ml Bag
Labeler	2	Small Box Label

4.2.8.4 Remote Interface Structure

An array of structure elements used for coordinating upstream or downstream machines in a cell with multiple unit machines.

The array is a length that is equal to the number of machines that will be sending commands. This could be expanded if a machine is capable of receiving material from multiple upstream and/or downstream machines, thereby receiving multiple commands and parameters.

This can be used for machine to machine coordination without supervisory control, or for tightly controlled units under supervisory control. These tags are typically used for consumption within the unit machine procedure. Specifically, if a remote controller was issuing commands, the commands would be read by this tag and used in the unit machine.

4.3 OPC UA Summary

4.3.1 Introduction

For PackML users that may not be familiar with OPC UA the following section provides a brief overview of key features that OPC UA provides.

4.3.2 What is OPC UA?

OPC UA is an open and royalty free set of standards designed as a universal communications protocol. While there are numerous communication solutions available, OPC UA has several advantages:

- A state of art security model (see OPC 10000-2).
- A fault tolerant communication protocol.

- An information modelling framework that allows application developers to represent their data in a way that makes sense to them.

OPC UA has a broad scope which delivers for economies of scale for application developers. This means that a larger number of high quality applications at a reasonable cost are available. When combined with powerful semantic models such as PackML, OPC UA makes it easier for end users to access data via generic commercial application.

The OPC UA model is scalable from small devices to ERP systems. OPC UA devices process information locally and then provide that data in a consistent format to any application requesting data - ERP, MES, PMS, Maintenance Systems, HMI, Smartphone or a standard Browser, for examples. For a more complete overview see OPC 10000-1.

4.3.3 Basics of OPC UA

As an Open Standard, OPC UA is based on standard Internet technologies – TCP/IP, HTTPS, Ethernet, and XML.

As an Extensible Standard, OPC UA provides a set of services (see OPC 10000-4) and a basic information model framework. This framework provides an easy manner for creating and exposing vendor defined information in a standard way. More importantly all OPC UA *Clients* are expected to be able to discover and use vendor defined information. This means OPC UA users can benefit from the economies of scale that come with generic visualization and historian applications. This specification is an example of an OPC UA Information Model designed to meet the needs of developers and users.

OPC UA *Clients* can be any consumer of data from another device on the network to browser based thin clients and ERP systems. The full scope of OPC UA applications are shown in Figure 3.

Figure 3 – The Scope of OPC UA within an Enterprise

OPC UA provides a robust and reliable communication infrastructure having mechanisms for handling lost messages, failover, heartbeat, etc. With its binary encoded data, it offers a high-performing data exchange solution. Security is built into OPC UA as security requirements become more and more important especially since environments are connected to the office network or the internet and attackers are starting to focus on automation systems

4.3.4 Information Modelling in OPC UA

4.3.4.1 Concepts

OPC UA provides a framework that can be used to represent complex information as *Objects* in an address space which can be accessed with standard web services. These *Objects* consist of *Nodes* connected by *References*. Different classes of *Nodes* convey different semantics. For example, a *Variable Node* represents a value that can be read or written. The *Variable Node* has an associated *DataType* that can define the actual value, such as a string, float, structure etc. It can also describe the variable value as a variant. A *Method Node* represents a function that can be called. Every *Node* has a number of *Attributes* including a unique identifier called a *NodeId* and non-localized name called as *BrowseName*. An *Object* representing a 'Reservation' is shown in Figure 4.

Figure 4 – A Basic Object in an OPC UA Address Space

Object and *Variable Nodes* are called *Instance Nodes* and they always reference a *TypeDefinition* (*ObjectType* or *VariableType*) Node which describes their semantics and structure. Figure 5 illustrates the relationship between an Instance and its Type Definition.

The *Type Nodes* are templates that define all of the children that can be present in an *Instance* of the *Type*. In the example in Figure 5 the PersonType *ObjectType* defines two children: First Name and Last Name. All instances of PersonType are expected to have the same children with the same *BrowseNames*. Within a Type the *BrowseNames* uniquely identify the child. This means *Client* applications can be designed to search for children based on the BrowseNames from the Type instead of *NodeIds*. This eliminates the need for manual reconfiguration of systems if a *Client* uses *Types* that multiple devices implement.

OPC UA also supports the concept of sub typing. This allows a modeller to take an existing *Type* and extend it. There are rules regarding sub typing defined in OPC 10000-3, but in general they allow the extension of a given type or the restriction of a *DataType*. For example, the modeller may decide that the existing *ObjectType* in some cases needs an additional variable. The modeller can create a Subtype of the object and add the variable. A *Client* that is expecting the parent type can treat the new *Type* as if it was of the parent Type. With regard to DataTypes, if a *Variable* is defined to have a numeric value, a sub type could restrict the value to a float.

Figure 5 – The Relationship between Type Definitions and Instances

References allow Nodes to be connected together in ways that describe their relationships. All *References* have a *ReferenceType* that specifies the semantics of the relationship. *References* can be hierarchical or non-hierarchical. *Hierarchical References* are used to create the structure of *Objects* and *Variables. Non-Hierarchical* are used to create arbitrary associations. Applications can define their own *ReferenceType* by creating Subtypes of the existing *ReferenceType*. Subtypes inherit the semantics of the parent but may add additional restrictions. Figure 6 depicts several references connecting different *Objects*.

Figure 6 – Examples of References between Objects

The figures above use a notation that was developed for the OPC UA specification. The notation is summarized in Figure 7. UML representations can also be used; however, the OPC UA notation is less ambiguous because there is a direct mapping from the elements in the figures to Nodes in the address space of an OPC UA server.

Figure 7 – The OPC UA Information Model Notation

A complete description of the different types of *Nodes* and *References* can be found in OPC 10000-3 and the base OPC UA Address space is described in OPC 10000-5.

OPC UA specification defines a very wide range of functionality in its basic information model. It is not expected that all *Clients* or *Servers* support all functionality in the OPC UA specifications. OPC UA includes the concept of profiles, which segment the functionality into testable certifiable units. This allows the development of companion specification (such as OPC UA for ISA-95) that can describe the subset of functionality that is expected to be implemented. The profiles do not restrict functionality, but generate requirements for a minimum set of functionality (see OPC 10000-7).

The OPC Foundation also defines a set of information models that provide a basic set of functionality. The Data Access specification (see OPC 10000-8) provides a basic information model for typical data. The Alarm and Condition specification (see OPC 10000-9) defines a standard information model for Alarms and Conditions. The Programs specification (see OPC 10000-10) defines a standard information model for extending the functionality available via method calls and state machines. The Historical Access specification (see OPC 10000-11) defines the information model associated with Historical Data and Historical Events. The aggregates specification (see OPC 10000-13) defines a series of standard aggregate functions that allow a *Client* to request summary data. Examples of aggregates include averages, minimums, time in state, Standard deviation, etc.

4.3.4.2 Namespaces

OPC UA allows information from many different sources to be combined into a single coherent address space. Namespaces are used to make this possible by eliminating naming and id conflicts between information from different sources. Namespaces in OPC UA have a globally unique string called a *NamespaceUri* and a locally unique integer called a *NamespaceIndex*. The *NamespaceIndex* is only unique within the context of a *Session* between an OPC UA *Client* and an OPC UA *Server*. All of the web services defined for OPC UA use the *NamespaceIndex* to specify the *Namespace* for qualified values.

There are two types of values in OPC UA that are qualified with *Namespaces*: *NodeIds* and *QualifiedNames. NodeIds* are globally unique identifiers for *Nodes*. This means the same *Node* with the same *NodeId* can appear in many *Servers.* This, in turn, means *Clients* can have built in knowledge of some Nodes. OPC UA information models generally define globally unique *NodeIds* for the *TypeDefinitions* defined by the information model.

QualifiedNames are non-localized names qualified with a *Namespace*. They are used for the *BrowseNames* of Nodes and allow the same names to be used by different information models without conflict. The *BrowseName* is used to identify the children within a *TypeDefinitions*. Instances of a *TypeDefinition* are expected to have children with the same *BrowseNames. TypeDefinitions* are not allowed to have children with duplicate *BrowseNames*; however, instances do not have that restriction.

4.3.4.3 Companion Specifications

An OPC UA companion specification for an industry specific vertical market describes an information model by defining *ObjectTypes*, *VariableTypes*, *DataTypes* and *ReferenceTypes* that represent the concepts used in the vertical market. Table 1 contains an example of an *ObjectType* definition.

Table 1 – Example *ObjectType* Definition

Attribute	**Value**				
BrowseName	WidgetType				
IsAbstract	False				
Reference	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseObjectType* from OPC 10000-5.					
HasProperty	Variable	Color	String	PropertyType	Mandatory
HasProperty	Variable	Flavor	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	Rank	Int32	PropertyType	Mandatory

The *BrowseName* is a non-localized name for an *ObjectType.*

IsAbstract is a flag indicating whether instances of the *ObjectType* can be created.

The bottom of the table lists the child nodes for the type. The *Reference* is the type of reference between the *Object* instance and the child Node. The *NodeClass* is the class of Node. The *BrowseName* is the non-localized name for the child. The *DataType* is the structure of the Value accessible via the *Node* (only used for *Variable NodeClass Nodes*) and the *TypeDefinition* is the *ObjectType* or *VariableType* for the child.

The *ModellingRule* indicates whether a child is *Mandatory* or *Optional*. It can also indicate cardinality. Note that the *BrowseName* is not defined if the cardinality is greater than 1. Figure 8 visually depicts the *ObjectType* defined in Table 1 along with two instances of the *ObjectType*.

Figure 8 – A Visual Representation of the Sample ObjectType

5 Modelling Approach of PackML

The modelling approach for generating an UA model from the PackML specification follows the following general concepts / suggestions.

In PackML a number of standard tag names and standard values are defined, OPC UA defines standard types from which any number of instances can be created. Each instance will contain the same items as defined in the type, allowing easy access for *Clients*.

When possible OPC UA constructs will be used to represent parallel PackML concepts including:

- *StateMachines* to reflect the state of the system
- *Methods* to issue commands to the *Server*
- *DataTypes*

6 PackML Data Representation Model

6.1 General

The OPC UA PackML information model is a representation of the PackML data model in OPC *ObjectTypes*, *VariableTypes*, *DataTypes* and *ReferenceTypes*.

This model generates standard types. All PackML types will be defined in their own *Namespace* and will begin with “PackML” A key point is a standard *ObjectType* representation of the *StateMachines* defined in PackML. The model also defines some standard instances that are expected as a starting point for this model.

The following conventions apply to *ObjectType*, *VariableType* and *DataType* naming:

- All ObjectTypes include “*ObjectType*” as part of the name
- All StateMachines will end in “StateMachine”, all States will end in “State”, All Transitions will end in “Transition”
- All *DataTypes* that are structures include “DataType” as part of the name, this is to be able to differentiate them from any *VariableTypes* that will just end in Type.
- All enumerations will end in “Enum”, to clearly identify that it is an enumeration.
- All base *DataTypes* (int32, float, …) used in the OPC UA server will be those defined in OPC UA, see OPC 10000-6 for more detail on the representation of the datatypes. This specification is typically implemented in a PLC, Annex C provides a non-normative copy of the DataType mapping described in PLC Open

Figure 9 - System Overview

Figure 9 illustrates the scope of PackML in a typical environment, with units acting as OPC UA *Servers* and a line controller as an OPC UA *Client* application. It defines a standard set of interfaces to and from a unit/machine, so that it can be controlled as an element of an overall production line. It maps the internal states of the unit into a standard state model, and internal commands into a standard set of commands, hiding the details of the actual implementation of the unit’s code.

Figure 9 also illustrates another typical example in which units perform peer-to-peer communication to coordinate the states and modes of an entire line. In this situation, each unit could act as both an OPC UA *Server* (of their own local state) and an OPC UA *Client* to communicate to upstream and downstream units.

6.2 Instance AddressSpace

Figure 10 provides an overview of the instance object model for PackML

Figure 10 - PackML Object Instance Overview

The OPC UA Server shall have a PackMLObjects folder under the OPC defined *Objects* folder on a UA *Server.* This folder shall contain one or more instances of *PackMLBaseObjectType* (see 6.3.2 for definition of *PackMLBaseObjectType*). A single OPC UA *Server* might contain a single instance of a PackML system or it might contain multiple PackML systems.

The *PackMLObjects* node is formally defined in **Table 2**.

Table 2 – PackMLObjects definition

Attribute	**Value**		
BrowseName	PackMLObjects		
References	**NodeClass**	**BrowseName**	**TypeDefinition**
OrganizedBy by the Objects Folder defined in OPC 10000-5			
HasTypeDefinition	ObjectType	FolderType	

6.3 Objects and ObjectTypes

6.3.1 Overview

The PackML model when adapted to OPC UA results in a number of *StateMachines* (see section 6.3.5 for a definition). Instances of these *StateMachines* may not expose all states and transitions at all times. The actual list of *AvailableStates* and *AvailableTransitions* are configured and each instance would be defined by either the end user or the machine builder. The PackML model also includes other meta data such as available mode, current mode, see section 6.3.2 for a complete list

6.3.2 PackMLBaseObjectType

The *PackMLBaseObjectType* defines a base type that can be used with any machine or object. This base type provides all required information for a working PackML system.

Figure 11 - PackMLBaseObjectType Overview

Table 3 formally defines the *PackMLBaseObjectType*.

Table 3 – PackMLBaseObjectType Definition

Attribute	**Value**					
BrowseName	PackMLBaseObjectType					
IsAbstract	False					
Reference	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**	**RW**
Subtype of the *BaseObjectType* from OPC 10000-5.						
HasProperty	Variable	TagID	String	PropertyType	Optional	
HasProperty	Variable	PackMLVersion	String	PropertyType	Optional	
HasComponent	Object	Admin		PackMLAdminObjectType	Mandatory	
HasComponent	Object	Status		PackMLStatusObjectType	Mandatory	
HasComponent	Object	BaseStateMachine		PackMLBaseStateMachineType	Mandatory	
HasComponent	Method	SetUnitMode	Defined in section 6.7.2		Mandatory	
HasComponent	Method	SetMachSpeed	Defined in section 6.7.3		Mandatory	
HasComponent	Method	SetProduct	Defined in section 6.7.4		Mandatory	
HasComponent	Method	SetParameter	Defined in section 6.7.17		Mandatory	
HasComponent	Method	RemoteCommand	Defined in section 6.7.15		Optional	
HasComponent	Method	SetInterlock	Defined in section 6.7.16		Optional	

TagID – provide an additional field in which an associated name (third party cross reference or other string) can be stored. It can also be an additional name used to identify this PackML System.

PackMLVersion – provides information about the version of PackML.

Admin provides administrative functionality required for the PackML OPC UA server. It is defined in section 6.3.4. The administrative functionality exposed by this *Object* should be restricted to only users with administrative rights.

Status provides the status information required for a PackML OPC UA *Server*. It is defined in section 6.3.3.

UnitMode method allows an OPC UA *Client* to change the mode of the machine. The available modes are part of the supported Modes and a *Client* can pass any of the values listed. The *Method* may return an error if the requested mode is not allowed based on either the current mode of the machine or the state of the machine. For additional details see the definition of the *UnitMode Method* in 6.7.2

SetMachSpeed Method allows a *Client* to change the machine speed.

SetProduct Method allows a *Client* to change the product(s) and the *ProcessVariables* and Ingredients. For additional details see the definition of *SetProduct Method* in 6.7.4 .

SetParameter Method allows a *Client* to set the parameters for the machine. For additional details see the definition of SetParameter *Method* in 6.7.17.

RemoteCommand Method allows a *Client* to send a command to the UA *Server* that is to be passed to the PackML *Server* and or upstream or downstream *Servers.* Parameters sent to the Remote system are typically used in the EXECUTE and STARTING states for a production task. With the restriction that *RemoteCommand* Parameter Values are limited to REAL values. For additional details see the definition of the *RemoteCommand Method* in 6.7.15

SetInterlock method allows a *Client* to set one of the interlocks associated with the system. For additional details see the definition of the SetInterlock *Method* in 0

6.3.3 PackMLStatusObjectType

The *PackMLStatusObjectType* defines an *ObjectType* that is used to group all of the status information that is part of the PackML information model. It is illustrated in Figure 12

Figure 12 - PackMLStatusObjectType Overview

Table 4 formally defines the *PackMLStatusObjectType*.

Table 4 – PackMLStatusObjectType Definition

Attribute	**Value**
BrowseName	PackMLStatusObjectType

IsAbstract	False					
Reference	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**	**RW**
Subtype of the *BaseObjectType* from OPC 10000-5.						
HasComponent	Variable	UnitModeRequested	Boolean	BaseDataVariableType	Optional	R
HasProperty	Variable	UnitSupportedModes	NodeId	PropertyType	Mandatory	R
HasComponent	Variable	UnitCurrentMode	Enumeration	PropertyType	Mandatory	R
HasComponent	Variable	UnitModeChangeInProgress	Boolean	BaseDataVariableType	Optional	R
HasComponent	Variable	StateRequested	Int32	BaseDataVariableType	Optional	R
HasComponent	Variable	StateChangeInProgress	Boolean	BaseDataVariableType	Optional	R
HasComponent	Variable	MachSpeed	Float	AnalogItemType	Mandatory	R
HasComponent	Variable	CurMachSpeed	Float	AnalogItemType	Mandatory	R
HasComponent	Variable	EquipmentBlocked	Boolean	BaseDataVariableType	Mandatory	R
HasComponent	Variable	EquipmentStarved	Boolean	BaseDataVariableType	Mandatory	R
HasComponent	Variable	MaterialInterlocked	Boolean	BaseDataVariableType	Optional	R
HasInterlock	Variable	<MaterialInterlock>	Boolean	InterlockVariableType	OptionalPlaceholder	R
HasComponent	Variable	Parameters	PackMLDescriptorDataType[].	BasedDataVariableType	Optional	R
HasComponent	Variable	RemoteParameters	PackMLDescriptorDataType[].	BasedDataVariableType	Optional	R
HasComponent	Variable	Products	PackMLProductDataType[]	BaseDataVariableType	Optional	R

In OPC UA defined *StateMachines*, a mandatory *Variable CurrentState* provides the current state of the *StateMachine*, which is the current state of the PackML device. *CurrentState* is defined in OPC 10000-5.

UnitModeRequested - If TRUE, indicates that a unit mode change was requested, reflects the status of the Command UnitModeRequested.

UnitSupportedModes – provides the *NodeId* of the enumeration *DataType* that describes the available modes for this PackML instance. A *Server* might have more than one of these instances; each instance might expose a different set of available modes and thus have a different enumeration.

UnitCurrentMode - is used to display the current mode of the instance of this type. The *DataType* is *Enumeration* which is abstract, but an instance shall be assigned a concrete enumeration, which corresponds to the enumeration listed in *SupportedModes*.

UnitModeChangeInProgress – a flag that indicates a unit change has been requested and is in progress

StateRequested - This value is used for state transition checking, to ensure that transitions to a target state can be achieved. The target state, *StateRequested*, is a numerical value corresponding to a state in the base state model (shown above).

StateChangeInProgress – a flag that indicates that a state change has been requested and is in progress. The StateMachine will report the current state.

MachSpeed - Setpoint speed of the unit.

CurMachSpeed - Current speed of the unit.

EquipmentInterlock.Blocked - If TRUE, then processing is suspended because downstream equipment is unable to receive material (e.g. downstream buffer is full)

EquipmentInterlock.Starved - If TRUE, then processing is suspended because upstream equipment is unable to send material.

MaterialInterlocked – a flag that indicates this machine is interlocked. It is a summary of the interlock variable that is associated with this machine.

<*MaterialInterlock*> - this is a placeholder that indicates a machine may have one or more interlock variables defined. The interlock *Variable* provides information about the actual interlock including if it is active. See 6.4.1 for additional details and examples.

Parameters – Current parameters used in the production job. This reflects the last parameter sent via the *SetParameter Method*.

RemoteParameters. – the last remote parameter that were sent to the machine. This is optional variable is provided only if sending remote parameters are supported, the *RemoteCommand Method* is provided this variable shall also be provided. For additional details see the *RemoteCommand Method* definition.

Products – provides a list of the products supported by this machine. The array is typically needed for machines that run multiple products. It defines the IDs of the products and process & process variables associated with the product. The product data can come from either a local HMI or remote systems and are used to process the product on the unit machine.

6.3.4 PackMLAdminObjectType

The *PackMLAdminObjectType* defines an *ObjectType* that is used to group all of the Admin information that is part of the PackML information model. It is illustrated in Figure 13.

Figure 13 - PackMLAdminObjectType Overview

Table 5 formally defines the PackMLAdminObjectType.

Table 5 – PackMLAdminObjectType Definition

Attribute	**Value**

BrowseName	PackMLAdminObjectType					
IsAbstract	False					
Reference	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**ModelingRule**	**RW**
Subtype of the *BaseObjectType* from OPC 10000-5.						
HasComponent	Variable	Parameters	PackMLDescriptorDataType[].	BaseDataVariableType	Optional	R
HasAlarm	Variable	<Alarm>	PackMLAlarmDataType	PackMLAlarmVariableType	Optional Placeholder	R
HasComponent	Variable	AlarmExtent	Int32	BaseDataVariableType	Optional	R
HasAlarmHistory	Variable	<AlarmHistory>	PackMLAlarmDataType	PackMLAlarmVariableType	Optional Placeholder	R
HasComponent	Variable	AlarmHistoryExtent	Int32	BaseDataVariableType	Optional	R
HasWarning	Variable	<Warning>	PackMLAlarmDataType	PackMLAlarmVariableType	Optional Placeholder	R
HasComponent	Variable	WarningExtent	Int32	BaseDataVariableType	Optional	R
HasStopReason	Variable	<StopReason>	PackMLAlarmDataType	PackMLAlarmVariableType	Optional Placeholder	R
HasComponent	Variable	StopReasonExtent	Int32	BaseDataVariableType	Optional	R
HasComponent	Variable	ModeCurrentTime	Int32[]	BaseDataVariableType	Optional	R
HasComponent	Variable	ModeCumulativeTime	Int32[]	BaseDataVariableType	Optional	R
HasComponent	Variable	StateCurrentTime	Int32[][]	BaseDataVariableType	Optional	R
HasComponent	Variable	StateCumulativeTime	Int32[][]	BaseDataVariableType	Optional	R
HasComponent	Variable	ProdConsumedCount	PackMLCountDataType []	BaseDataVariableType	Optional	R
HasComponent	Variable	ProdProcessedCount	PackMLCountDataType []	BaseDataVariableType	Optional	R
HasComponent	Variable	ProdDefectiveCount	PackMLCountDataType []	BaseDataVariableType	Optional	R
HasComponent	Variable	AccTimeSinceReset	Int32	BaseDataVariableType	Optional	R
HasComponent	Variable	MachDesignSpeed	Float	BaseDataVariableType	Optional	R

Parameters - The parameter tags associated with the local interface are typically used as parameters that are displayed or used on the unit locally, for example from an HMI. These parameters can be used to display any quality, alarm, or machine downtime parameter. The *Parameters* are typically limited to parameters related the unit. The length of the array is the maximum number of parameters needed.

<Alarm> - Alarm Events (trigger, value, message, category,…). The alarm tags associated to the local interface are typically used as parameters that are displayed or used on the unit locally, for example from an HMI. These alarm parameters can be used to display any alarm, or machine downtime cause that is currently occurring in the system. The alarms are typically limited to the machine unit. Each machine can define as many alarms as are required for the machine. The alarms variables can have any name, but they are of *PackMLAlarmVariableType*. .

AlarmExtent - Defines the maximum number of alarms available, for the machine annunciation or reporting

AlarmHistory - Defines the maximum number of alarm history elements available.

Warning - Array of warning information Events. Warnings are general events that do not cause the machine to stop, but may require operator action because a stoppage may be imminent. Warning elements have the same structure as Stop Reason elements.

WarningExtent - Defines the maximum number of warning elements available.

StopReason - An array of stop reason *Events* (similar to Alarms) which define the possible stop reasons (trigger, value, message, category).

StopReasonExtent - Defines the maximum number of stop reason elements or available.

ModeCurrentTime - The current amount of time, in seconds, that the machine has been in each mode. The array index for a mode is the Unit mode value. The values roll over to 0 at 2,147,483,647.

ModeCumulativeTime - The cumulative amount of time, in seconds, that the machine has been in each mode. The array index for a mode is the Unit mode value. The value is the cumulative elapsed time the machine has spent in each mode since its timers and counters were reset. The values roll over to 0 at 2,147,483,647.

StateCurrentTime - The current amount of time, in seconds, that the machine has been in each state for each mode. The first array index for is the Unit mode value, the second array index is the state value.. The values roll over to 0 at 2,147,483,647.

StateCumulativeTime - The cumulative amount of time, in seconds, that the machine has been in each state for each mode. The first array index for is the Unit mode value, the second array index is the state value. The value is the cumulative elapsed time the machine has spent in each mode and state since its timers and counters were reset. The values roll over to 0 at 2,147,483,647

ProdConsumedCount - Represents the material used/consumed in the production machine. An example of tag usage would be the number of bags consumed in a filler, or bagger packaging machine, or the amount of linear length used, or the number caps used. This tag can be used locally or remotely if needed. The extent of the array is typically limited to the number of raw materials needed to be counted. The array is typically used for unit machines that run multiple raw materials.

ProdProcessedCount - Represents the products processed in the production machine. An example of tag usage would be the number of products that were made, including all good and defective products. The structure of the *ProdProcessedCount* is the same as the *ProdConsumedCount.* The length of the array is typically limited to the number of products that need to be counted. The number of products processed minus the defective count is the number of non-defective products made by the machine. The array index of # = 0 should be reserved for the count of the number of units from the primary production stream.

ProdDefectiveCount - Represents the products marked as defective in the production machine. The structure of the *ProdDefectiveCount* is the same as the *ProdConsumedCount.* The length of the array is typically limited to the number of products that need to be counted. The number of products processed minus the defective count is the number of non-defective products made by the machine. The array index of # = 0 should be reserved for the count of the number of units from the primary production stream.

AccTimeSinceReset - Represents the amount of time, in seconds, since the last reset of all counters as triggered. Counters that are reset are:

- UnitName.Admin.ModeCurrentTime[#]
- UnitName.Admin.ModeCumulativeTime[#]
- UnitName.Admin.StateCurrentTime[#,#]
- UnitName.Admin.StateCumulativeTime[#,#]
- UnitName.Admin.ProdConsumedCount[#].Count
- UnitName.Admin.ProdProcessedCount[#].Count
- UnitName.Admin.ProdDefectiveCount[#].Count
- UnitName.Admin.AccTimeSinceReset

MachDesignSpeed - Represents the maximum design speed of the machine in primary packages per minute for the package configuration being run. This speed is NOT the maximum speed as specified by the manufacturer, but rather the speed of the machine is designed to run in its installed environment.

6.3.5 StateMachines Overview

The Figure 14 provides an overview the *StateMachines* that are part of the model.

Figure 14 - PackML StateMachines Overview

A key point in PackML *StateMachines* is that all of the *StateMachines* defined in PackML shall require that the optional *AvailableTransition* and *AvailableStates* component of the FiniteStateMachineType be provided on all instance of the *StateMachine.* This allows *Clients* to understand the available *States* and *Transitions* for the given instance of the *StateMachine*. A *StateMachine* may restrict the *States* and *Transition* that are currently available. The following figure provides an overview of the PackML States. The *Clearing State* is commonly the initial sub-state that will be the starting point for the *Clear* parent state. The *Running State* commonly will use the *Resetting State* as the initial state, but not all instance of the *Running State* sub-state model will include *Resetting*, so no initial state is defined for the *Running State*. The only valid initial *States* for this model is the *Idle* or *Resetting States*. This is *Server* dependant. The initial state for the system is *Cleared* with the sub state of *Clearing.* Figure 15 provide an overview of the states and transitions in the *StateMachine.*

Figure 15 - PackML States

6.3.6 PackMLBaseStateMachineType

The *PackMLBaseStateMachineType* is the top level *StateMachine* for PackML. It is illustrated in Figure 16. The TR-88 specification does not define an initial *State* for this *StateMachine*, but typically the state machine uses either the *Aborted* or *Cleared State* as an initial *State*. Annex B provide recommended display names for the various states.

Figure 16 - PackMLBaseStateMachineType illustration

The *PackMLBaseStateMachineType* defines the available states in a PackML system. The type is defined in Table 6. *StateTypes* and *TransitionTypes* only exist in the type system, thus they do not have a modelling rule.

Table 6 – PackMLBaseStateMachineType Definition

Attribute	**Value**				
BrowseName	PackMLBaseStateMachineType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**Data Type**	**TypeDefinition**	**Modelling Rule**
Subtype of the *FiniteStateMachineType* defined in OPC 10000-5					
HasComponent	Variable	0:AvailableTransitions	NodeId[]	BaseDataVariableType	Mandatory
HasComponent	Variable	0:AvailableStates	NodeId[]	BaseDataVariableType	Mandatory
HasComponent	Object	Aborting		StateType	
HasComponent	Object	Aborted		StateType	
HasComponent	Object	Cleared		InitialStateType	
HasComponent	Object	MachineState		PackMLMachineStateMachineType	Mandatory
HasComponent	Object	AbortedToCleared		TransitionType	
HasComponent	Object	AbortingToAborted		TransitionType	
HasComponent	Object	ClearedToAborting		TransitionType	
HasComponent	Method	Abort	Defined in 6.7.5		Optional
HasComponent	Method	Clear	Defined in 6.7.6		Optional

The *AvailableTransitions* and *AvailableStates* are optional variables in the *FiniteStateMachine*, but they are overridden in the *PackMLBaseStateMachine* and are made *Mandatory.* The *PackMLBaseStateMachine* does include a sub-state machine that provides sub-states for the *Cleared State*.

Aborting - The ABORTED state can be entered at any time in response to the *Abort* command or on the occurrence of a machine fault. The aborting logic will bring the machine to a rapid safe stop. Operation of the emergency stop will cause the machine to be tripped by its safety system. It will also provide a signal to initiate the ABORTING State. The value of this *StateType* is 8.

Aborted - This state maintains machine status information relevant to the Abort condition. The machine can only exit the ABORTED state after an explicit Clear command, subsequently to manual intervention to correct and reset the detected machine faults. The value of this *StateType* is 9.

Cleared – this state exposes the MachineState sub StateMachine and state associated with this substate machine. The value of this *StateType* is 19.

MachineState – A PackMLMachineStateMachineType defined in section 6.3.7.

Abort – a *Method* to trigger a change of state to *Aborting*. This will affect all sub-states in cleared state. Defined in 6.7.5.

Clear – a *Method* to trigger a change of state to the *Cleared*. Defined in *6.7.6.*

Table 7 defines the available *Transitions* in the PackMLBaseStateMachineType.

Table 7 – PackMLBaseStateMachineType transitions

BrowseName	**References**	**BrowseName**	**TypeDefinition**
Transitions			
AbortedToCleared	FromState	Aborted	StateType
	ToState	Cleared	StateType
	HasCause	Clear	*Method*
AbortingToAborted	FromState	Aborting	StateType
	ToState	Aborted	StateType
ClearedToAborting	FromState	Cleared	StateType
	ToState	Aborting	StateType
	HasCause	Abort	*Method*

The OPC UA *StateMachine* model supports *HasEffect* references (and subtypes of these references) that provide linkages to actions (*Alarms* or other changes) that are triggered by *StateMachine Transitions*. At this point no *HasEffects* have been defined in the OPC UA PackML model, but future version of this model will include standard *HasEffect References*. Vendors can define their own *HasEffect* references.

6.3.7 PackMLMachineStateMachineType

The *PackMLMachineStateMachineType* defines the machine level state machine. It is illustrated in Figure 17.

The TR-88 specification does not define an initial *State* for this *StateMachine*, but typically the state machine uses *Stopped State* as an initial *State.* Annex B provides recommended display names for the various states.

Figure 17 - PackMLMachineStateMachineType illustration

Table 8 defines the *PackMLMachineStateMachineType*. *StateTypes* and *TransitionTypes* only exist in the type system, thus they do not have a modelling rule.

Table 8 – PackMLMachineStateMachineType Definition

Attribute	**Value**				
BrowseName	PackMLMachineStateMachineType				
IsAbstract	False				
Reference	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *FiniteStateMachineType* from OPC 10000-5.					
HasComponent	Variable	0:AvailableTransitions	NodeId[]	BaseDataVariableType	Mandatory
HasComponent	Variable	0:AvailableStates	NodeId[]	BaseDataVariableType	Mandatory
HasComponent	Object	Stopped		StateType	
HasComponent	Object	Stopping		StateType	
HasComponent	Object	Clearing		StateType	
HasComponent	Object	Running		StateType	
HasComponent	Object	ExecuteState		PackMLExecuteStateMachineType	Mandatory
HasComponent	Object	StoppingToStopped		TransitionType	
HasComponent	Object	ClearingToStopped		TransitionType	
HasComponent	Object	StoppedToRunning		TransitionType	
HasComponent	Object	RunningToStopping		TransitionType	
HasComponent	Method	Stop	Defined in 6.7.7		Optional
HasComponent	Method	Reset	Defined in 6.7.8		Optional

The *AvailableTransitions* and *AvailableStates* are optional variables in the *FiniteStateMachine*, but they are overridden in the *PackMLMachineStateMachineType* and are made *Mandatory*. The *PackMLMachineStateMachineType* does include a sub-state machine that provides sub-states for the Run State.

Stopped - The machine is powered and stationary after completing the STOPPING state. All communications with other systems are functioning (if applicable). The value of this StateType is 2

Stopping - This state executes the logic which brings the machine to a controlled stop as reflected by the STOPPED state. The value of this StateType is 7.

Clearing - Initiated by a state command to clear faults that may have occurred when ABORTING, and are present in the ABORTED state. The value of this StateType is 1.

Running – the *State* that allows the *ExecuteState* machine to become active, enabling sub-states provided by this *StateMachine.* The value of this StateType is18.

ExecuteState – *StateMachine* that provides additional sub states.

Stop – A *Method* to trigger a change of state to *Stopping*. This will affect all sub-states in *Run* state. Defined in 6.7.7

Reset – A *Method* to trigger a change of state to *Running,* enabling all of the sub-states of *Running* and the respective *Methods* that they expose. Defined in 6.7.8

The transitions are defined in Table 9.

Table 9 – PackMLMachineStateMachineType transitions

BrowseName	**References**	**BrowseName**	**TypeDefinition**
Transitions			
StoppedToRunning	FromState	Stopped	StateType
	ToState	Running	StateType
	HasEffect		
	HasCause	Reset	*Method*
StoppingToStopped	FromState	Stopping	StateType
	ToState	Stopped	StateType
	HasEffect		
ClearingToStopped	FromState	Clearing	StateType
	ToState	Stopped	StateType
	HasEffect		
RunningToStopping	FromState	Running	StateType
	ToState	Stopping	StateType
	HasEffect		
	HasCause	Stop	*Method*

6.3.8 PackMLExecuteStateMachineType

The *PackMLExecuteStateMachineType* provides all of the base states defined in PackML. It is illustrated in Figure 18. The TR-88 specification does not define an initial State for this StateMachine, but typically the state machine use either the *Idle* or *Resetting State* as an initial *State.* Annex B provide recommended display names for the various states.

Figure 18 – PackMLExecuteStateMachineType illustration

The *PackMLExecuteStateMachineType* is defined in Table 10. *StateTypes* and *TransitionTypes* only exist in the type system, thus they do not have a modelling rule.

Table 10 – PackMLExecuteStateMachineType Definition

Attribute	**Value**				
BrowseName	PackMLExecuteStateMachineType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the Finite*StateMachineType* defined in OPC 10000-5					
HasComponent	Variable	0:AvailableTransitions	NodeId[]	BaseDataVariableType	Mandatory
HasComponent	Variable	0:AvailableStates	NodeId[]	BaseDataVariableType	Mandatory

HasComponent	Object	Resetting		StateType	
HasComponent	Object	Idle		StateType	
HasComponent	Object	Starting		StateType	
HasComponent	Object	Suspending		StateType	
HasComponent	Object	Suspended		StateType	
HasComponent	Object	Unsuspending		StateType	
HasComponent	Object	Holding		StateType	
HasComponent	Object	Held		StateType	
HasComponent	Object	Unholding		StateType	
HasComponent	Object	Execute		StateType	
HasComponent	Object	Completing		StateType	
HasComponent	Object	Complete		StateType	
HasComponent	Object	ResettingToIdle		TransitionType	
HasComponent	Object	IdleToStarting		TransitionType	
HasComponent	Object	StartingToExecute		TransitionType	
HasComponent	Object	ExecuteToSuspending		TransitionType	
HasComponent	Object	SuspendingToSupended		TransitionType	
HasComponent	Object	SuspendedToUnsuspending		TransitionType	
HasComponent	Object	UnsuspendingToExecute		TransitionType	
HasComponent	Object	ExecuteToHolding		TransitionType	
HasComponent	Object	HoldingToHeld		TransitionType	
HasComponent	Object	HeldToUnholding		TransitionType	
HasComponent	Object	UnholdingToExecute		TransitionType	
HasComponent	Object	ExecuteToCompleting		TransitionType	
HasComponent	Object	CompletingToComplete		TransitionType	
HasComponent	Object	CompleteToResetting		TransitionType	
HasComponent	Object	StartingToHolding		TransitionType	
HasComponent	Object	UnsuspendingToHolding		TransitionType	
HasComponent	Object	SuspendedToHolding		TransitionType	
HasComponent	Object	SuspendingToHolding		TransitionType	
HasComponent	Object	UnholdingToHolding		TransitionType	
HasComponent	Method	Reset	Defined in Clause 6.7.8		Optional
HasComponent	Method	ToComplete	Defined in Clause 6.7.9		Optional
HasComponent	Method	Start	Defined in Clause 6.7.10		Optional
HasComponent	Method	Unhold	Defined in Clause 6.7.11		Optional
HasComponent	Method	Suspend	Defined in Clause 6.7.12		Optional
HasComponent	Method	Hold	Defined in Clause 6.7.14		Optional
HasComponent	Method	Unsuspend	Defined in Clause		Optional

This *FiniteStateMachine* supports multiple *Active* states. It also supports 19 *Transitions* and a *Method* for transition between states. The transitions are described in Table 11.

Resetting: In response to a *Reset* command, the unit/machine will transition to *Resetting* from either *Stopped* or *Complete*. In this state the unit/machine attempts to clear any standing errors or stop causes. If successful, the unit/machine transitions to *Idle*. No hazardous motion should happen in this state. The value of this *StateType* is 15

Idle: The unit/machine is in an error-free state, waiting to start. The unit/machine transitions automatically to *Idle* after all steps necessary for *Resetting* have been completed. All conditions achieved during *Resetting* are maintained. A *Start* command will transition the unit/machine from *Idle* to *Starting*. The value of this *StateType* is 4.

Starting: The unit/machine completes all steps necessary to begin execution of the active machine mode. A *Start* command will cause the unit/machine to transition from Idle to *Starting*. The unit/machine will transition automatically from *Starting* to *Execute* once all required steps have been completed. The value of this *StateType* is 3.

Suspending: The unit/machine will transition from *Execute* to *Suspending* if conditions external to the unit/machine require a pause in production. Such conditions include faults to upstream or downstream equipment. The decision to *Suspend* may be made be a supervisory system monitoring the production line conditions or by unit/machine sensors detecting downstream blockages or upstream product scarcity. (In the former case, the unit/machine is 'blocked"; in the latter case, the unit/machine is "starved".) After all steps required to suspend the machine have been completed, the unit/machine will automatically transition to *Suspended* state. The value of this *StateType* is 13.

Suspended: The unit/machine is paused, waiting for external process conditions to clear. In this state, the unit/machine shall not produce product, but may, if required, dry-cycle. Once external conditions have returned to normal, the unit/machine will transition to *Unsuspending*, typically without operator intervention. The value of this *StateType* is 5.

Unsuspending: After all external process conditions that caused the unit/machine to suspend have cleared, the unit/machine completes all steps required to resume execution of the active machine mode. Once all required actions to unsuspend the unit/machine have been completed, the unit/machine will automatically transition to *Execute* state. The value of this *StateType* is 14.

Holding: The unit/machine will transition from *Execute* to *Holding* if conditions internal to the unit/machine require a pause in production. Such conditions would include low levels on materials required for production or other minor issues requiring operator service, for example. The decision to hold may be made automatically by the unit/machine itself or by an operator. After all steps required to hold the machine have been completed, the unit/machine will transition automatically to *Held* state. The value of this *StateType* is 10.

Held: The unit/machine is paused, waiting for internal process conditions to clear. In this state, the unit/machine shall not produce product, though it may, if required, dry-cycle. A transition to *Unholding* will occur once internal machine conditions have cleared or if the *Unhold* command is initiated by an operator. The value of this *StateType* is 11.

Unholding: After all internal process conditions that caused the unit/machine to hold have cleared, the unit/machine completes all steps required to resume execution of the active machine mode. Once all required actions to unhold the machine have been completed, the unit/machine will transition automatically to *Execute* state. The value of this *StateType* is 12.

Execute: If the unit/machine is actively carrying out the behaviour or activity defined by the selected mode, then the machine is in *Execute* state. If the unit/machine is in production mode, for example, this means that the machine is producing product. The value of this *StateType* is 6.

Completing: Once the process associated with the current mode has reached a defined threshold (e.g. the required number of products for the current job have been produced), the unit/machine transitions from *Execute* to *Completing*. In this state all steps necessary to shut down the current process are carried out. The machine then transitions automatically to *Complete* state. The value of this *StateType* is 16.

Complete: *Complete* indicates the process associated with the active mode has come to its defined end. The unit/machine will wait in this state until a *Reset* command is issued (in which case it will transition to *Resetting*), or until the unit/machine is *Stopped* or *Aborted*. The value of this *StateType* is 17.

The *Transitions* are described in Table 11. This *FiniteStateMachine* also supports six *Method*s, for transitioning between states. This *StateMachine* includes transition to Holding from Unholding, *Starting*, *Unsuspending*, *Suspended*, *Suspending*, all of which are extension to the ISA-TR88.00.02-2015 specification.

Table 11 – PackMLExecuteStateMachineType transitions

BrowseName	**References**	**BrowseName**	**TypeDefinition**
Transitions			
ResettingToIdle	FromState	Resetting	StateType
	ToState	Idle	StateType
IdleToStarting	FromState	Idle	StateType
	ToState	Starting	StateType
	HasCause	Start	*Method*
StartingToExecute	FromState	Starting	StateType
	ToState	Execute	StateType
ExecuteToSuspending	FromState	Execute	StateType
	ToState	Suspending	StateType
	HasCause	Suspend	*Method*
SuspendingToSuspended	FromState	Suspending	StateType
	ToState	Suspended	StateType
SuspendedToUnsuspending	FromState	Suspended	StateType
	ToState	Unsuspending	StateType
	HasCause	Unsuspend	*Method*
UnspendingToExecute	FromState	Unsuspending	StateType
	ToState	Execute	StateType
ExecuteToHolding	FromState	Execute	StateType
	ToState	Holding	StateType

	HasCause	Hold	*Method*
StartingToHolding	FromState	Starting	StateType
	ToState	Holding	StateType
	HasCause	Hold	*Method*
SuspendingToHolding	FromState	Suspending	StateType
	ToState	Holding	StateType
	HasCause	Hold	*Method*
SuspendedToHolding	FromState	Suspended	StateType
	ToState	Holding	StateType
	HasCause	Hold	*Method*
UnsuspendingToHolding	FromState	Unsuspending	StateType
	ToState	Holding	StateType
	HasCause	Hold	*Method*
UnholdingToHolding	FromState	Unholding	StateType
	ToState	Holding	StateType
	HasCause	Hold	*Method*
HoldingToHeld	FromState	Holding	StateType
	ToState	Held	StateType
HeldToUnholding	FromState	Held	StateType
	ToState	Unholding	StateType
	HasCause	Unhold	*Method*
UnholdingToExecute	FromState	Unholding	StateType
	ToState	Execute	StateType
ExecuteToCompleting	FromState	Execute	StateType
	ToState	Completing	StateType
	HasCause	Complete	*Method*
CompletingToComplete	FromState	Completing	StateType
	ToState	Complete	StateType
CompleteToResetting	FromState	Complete	StateType
	ToState	Resetting	StateType
	HasCause	Reset	*Method*

6.4 Variables and VariableTypes

6.4.1 InterlockVariableType

This *Variable* reflects an interlock. It is defined in Table 12.

Table 12 – InterlockVariableType definition

Attribute	Value				
BrowseName	InterlockVariableType				
IsAbstract	False				
ValueRank	Scalar				
DataType	Boolean				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseDataVariableType* defined in OPC 10000-5					

Instances of the *InterlockType VariableType* shall have a *BrowseName* that provides a descriptive name of the interlock. The instance of the *VariableType* shall include a description that provide further information about the interlock. For example, in a labelling machine, if the bin for label becomes empty, an interlock may exist that has a name of LabelStockEmpty with a description of "the label stock for the green label is empty". This *Variable* would become active (set to true) when the label stock becomes empty, which triggers the machine to become interlocked.

6.4.2 PackMLCountVariableType

The *PackMLCountVariableType* provides the variable representation of the *PackMLCountDataType*. It is defined in Table 13

Table 13 - PackMLCountVariableType definition

Attribute	Value				
BrowseName	PackMLCountVariableType				
IsAbstract	False				
ValueRank	Scalar				
DataType	PackMLCountDataType				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseDataVariableType* defined in OPC 10000-5.					
HasComponent	Variable	ID	Int32	BaseDataVariable	Mandatory
HasComponent	Variable	Name	String	BaseDataVariable	Mandatory
HasComponent	Variable	Unit	EUInformation	BaseDataVariable	Mandatory
HasComponent	Variable	Count	Int32	BaseDataVariable	Mandatory
HasComponent	Variable	AccCount	Int32	BaseDataVariable	Mandatory

ID – an integer identifier for the variable.

Name - A string description of the material.

Unit - OPC UA engineering unit information for the count.

Count – the count associated with the item.

AccCount – The accumulated count associated with the item.

6.5 DataTypes

6.5.1 Overview

This section defines any enumeration or structure that are defined as part of the PackML specification.

6.5.2 ProductionMaintenanceModeEnum

The *ProductionMaintenanceModeEnum* describes the predefined modes. This is a default mode enumeration. A *Server* may define additional enumeration that describe the modes they support, but any such enumeration must include "Produce" as enumeration 1 and if Maintenance or Manual are include, they must be 2 and 3 respectively. Any additional mode must start at 4 or greater. If vendor specific or end user specific mode enumerations are included they shall be defined as a subtype of enumeration. The ProductionMaintenanceModeEnum is the default enumeration that shall be used if no vendor or end user mode enumeration is defined. The *ProductionMaintenanceModeEnum* is defined in Table 14.

Table 14 – ProductionMaintenanceModeEnum values

Value	Description
Invalid_0	This is an invalid mode
Produce_1	Machine is in production mode
Maintenance_2	Machine is in maintenance mode
Manual_3	Machine is in manual mode

where the following definition apply:

- Produce corresponds to the PackML Production Mode which is routine production.
- Maintenance corresponds to the PackML Maintenance Mode which is the ability to run a machine independent of other machine in a production line.
- Manual corresponds to the PackML Manual Mode which provides direct control of the individual machine elements.

6.5.3 PackMLCountDataType

The *PackMLCountDataType* is used to generate summary information about the system. The information depending on the use might be related to produced product, defective materials or any other information that needs to be tracked. This *DataType* also has a corresponding *VariableType* (see 6.4.2). It is formally defined in Table 15

Table 15 – PackMLCountDataType Structure

Name	Type	Description
PackMLCountDataType	Structure	
ID	Int32	A user defined value that represents the consumed (processed or defective) material. Typically, this is an SKU number or a user material master number.
Count	Int32	The amount of consumed (processed or defective) material on the current production job.
AccCount	Int32	The cumulative count value of the material produced (or consumed). This counter gives the user a non-resetting counter that may be used for OEE calculations

6.5.4 PackMLDescriptorDataType

The *PackMLDescriptorDataType* provides the PackML Parameter structure. This *DataType* also has a corresponding *Variable* type (see **Error! Reference source not found.**). The *PackMLDescriptorDataType* is formally defined in Table 16

Table 16 – PackMLDescriptorDataType Structure

Name	Type	Description
PackMLDescriptorDataType	Structure	
ID	Int32	A unique number assigned to the parameter.
Name	String	The name of the parameter
Unit	EUInformation	OPC UA engineering unit information
Value	Float	This is the numeric value of the parameter
	.	

6.5.5 PackMLIngredientsDataType

The *PackMLIngredientsDataType* provides the PackML Parameter structure. The *PackMLIngredientsDataType* is formally defined in Table 17.

Table 17 – PackMLIngredientsDataType Structure

Name	Type	Description
PackMLIngredientsDataType	Structure	
IngredientID	Int32	A unique number assigned to the ingredient.
Parameters	PackMLDescriptorDataType[]	The array of Parameters that correspond to the ingredient

6.5.6 PackMLProductDataType

The *PackMLProductDataType* provides the PackML product information. The *PackMLProductDataType* is formally defined in Table 18.

Table 18 – PackMLProductDataType Structure

Name	Type	Description
PackMLProductDataType	Structure	
ProductID	Int32	A unique number assigned to the product.
ProcessVariables	PackMLDescriptorDataType[]	The array of Process variables associated with this product
Ingredients	PackMLIngredientsDataType[]	The array of ingredients associated with this product.

6.5.7 PackMLRemoteInteraceDataType

The *PackMLRemoteInterfaceDataType* provides the PackML remote connection information. The *PackMLRemoteInterfaceDataType* is formally defined in Table 19.

Table 19 – PackMLRemoteInterfaceDataType Structure

Name	Type	Description
PackMLRemoteInterfaceDataType	Structure	This datatype is used with the *RemoteCommand Method* defined in 6.7.15. See Table 55 for a complete description of the structure elements.
Number	Int32	
ControlCmdNumber	Int32	
CmdValue	Int32	
Parameter	PackMLDescriptorDataType[]	

6.6 ReferenceTypes

6.6.1 HasInterlock

This reference type is used to point to an *InterlockVariableType*. It is defined in Table 20

Table 20 – HasInterlock reference type

Attributes	Value		
BrowseName	HasInterlock		
InverseName	InterlockFor		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of HasComponent defined in OPC 10000-5			

6.6.2 HasAlarm

This reference type is used to point to an Alarm which is of *PackMLAlarmVariableType* . It is defined in Table 21.

Table 21 – HasAlarm reference type

Attributes	Value		
BrowseName	HasAlarm		
InverseName	AlarmFor		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of HasComponent defined in OPC 10000-5			

6.6.3 HasAlarmHistory

This reference type is used to point to an *AlarmHistory* which is of *PackMLAlarmVariableType* . It is defined in Table 22

Table 22 – HasAlarmHistory reference type

Attributes	Value
BrowseName	HasAlarmHistory

InverseName	AlarmHistoryFor		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of HasComponent defined in OPC 10000-5			

6.6.4 HasWarning

This reference type is used to point to an *Warning* which is of *PackMLAlarmVariableType* . It is defined in Table 21.

Table 23 – HasAWarning reference type

Attributes	**Value**		
BrowseName	HasWarning		
InverseName	WarningFor		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of HasComponent defined in OPC 10000-5			

6.6.5 HasStopReason

This reference type is used to point to a *StopReason* which is of *PackMLAlarmVariableType* . It is defined in Table 21.

Table 24 – HasStopReason reference type

Attributes	**Value**		
BrowseName	HasStopReason		
InverseName	StopReasonFor		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of HasComponent defined in OPC 10000-5			

6.7 Methods

6.7.1 Overview

This section provides definition of the method used in this specification. These methods are referenced from more than one location or are part of more than one object in some cases. The functionality for the method is the same for all objects

6.7.2 SetUnitMode Method

This *Method* allows an OPC UA *Client* to change the mode of the unit. *Parameters* are defined in Table 25

Signature

```
SetUnitMode(
     [in] NodeId ModeSelection
     [in] Int32 RequestedMode
    );
```

Table 25 - SetUnitMode Method Parameters

Argument	**Description**
ModeSelection	The NodeId of the enumeration that describes the available modes
ModeSelectionRequestedMode	The requested mode from the list of available modes in the enumeration.

Method result codes are defined in Table 26

Table 26 - SetUnitMode Method Result Codes

Result Code	**Description**
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.)
Bad_ArgumentsMissing	See OPC 10000-4 – Services for the description of this result code (The *Client* did not specify all of the input arguments for the *Method.*)
Bad_TooManyArguments	See OPC 10000-4 – Services for the description of this result code (The *Client* specified more input arguments than defined for the *Method.*)
Bad_InvalidArgument	See OPC 10000-4 – Services for the description of this result code. (Used to indicate in the operation level results that one or more of the input arguments are invalid. The inputArgumentResults contain the specific status code for each invalid argument.)
Bad_TypeMismatch	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that an input argument does not have the correct data type.)

Table 27 specifies the *AddressSpace* representation for the *SetUnitMode Method*. *SetUnitMode* includes an array of *InputArguments*, where the input argument details are provided in Table 25.

Table 27 - SetUnitMode Method AddressSpace Definition

Attribute	**Value**				
BrowseName	SetUnitMode				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory

6.7.3 SetMachSpeed Method

This *Method* allows an OPC UA *Client* to change the speed of the machine or unit. *Parameters* are defined in Table 28

Signature

```
SetMachSpeed(
     [in] Float RequestedMachineSpeed
    );
```

Table 28 - SetMachSpeed Method Parameters

Argument	Description
RequestedMachineSpeed	The target machine speed

Method result codes are defined in Table 29

Table 29 - SetMachSpeed Method ResultCodes

Result Code	Description
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.)
Bad_ArgumentsMissing	See OPC 10000-4 – Services for the description of this result code (The *Client* did not specify all of the input arguments for the *Method.*)
Bad_TooManyArguments	See OPC 10000-4 – Services for the description of this result code (The *Client* specified more input arguments than defined for the *Method.*)
Bad_InvalidArgument	See OPC 10000-4 – Services for the description of this result code. (Used to indicate in the operation level results that one or more of the input arguments are invalid. The inputArgumentResults contain the specific status code for each invalid argument.)
Bad_TypeMismatch	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that an input argument does not have the correct data type.)

Table 30 specifies the *AddressSpace* representation for the *SetMachSpeed Method*. *SetMachSpeed* includes an array of *InputArguments*, where the input argument details are provided in Table 28.

Table 30 – SetMachSpeed Method AddressSpace Definition

Attribute	Value				
BrowseName	SetMachSpeed				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory

6.7.4 SetProduct Method

This *Method* allows an OPC UA *Client* to change product associated with this PackML system.

Signature

```
SetProduct(
      [in] PackMLProductDataType [] Products
      );
```

Table 31 - SetProduct Method Parameters

Argument	Description
Products	This structure is an array of product definition, which includes the ProductId, ProcessVariables array and Ingredient array. See 6.5.6 for a definition of the *DataType*.

Method result codes are defined in Table 32

Table 32 - SetProduct Method Result Codes

Result Code	Description
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.)
Bad_ArgumentsMissing	See OPC 10000-4 – Services for the description of this result code (The *Client* did not specify all of the input arguments for the *Method.*)
Bad_TooManyArguments	See OPC 10000-4 – Services for the description of this result code (The *Client* specified more input arguments than defined for the *Method.*)
Bad_InvalidArgument	See OPC 10000-4 – Services for the description of this result code. (Used to indicate in the operation level results that one or more of the input arguments are invalid. The inputArgumentResults contain the specific status code for each invalid argument.)
Bad_TypeMismatch	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that an input argument does not have the correct data type.)

Table 33 specifies the *AddressSpace* representation for the *SetProduct Method*. *SetProduct* includes an array of *InputArguments*, where the input argument details are provided in Table 31.

Table 33 – SetProduct Method AddressSpace Definition

Attribute	Value				
BrowseName	SetProduct				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory

6.7.5 Abort Method

This *Method* is used as part of the *PackMLBaseStateMachineType*. It allows an OPC UA *Client* to change the state of this state machine to the *Aborting* state.

Signature

```
Abort(
     );
```

Method result codes are defined in Table 34.

Table 34 - Abort Method result codes

Result Code	Description
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.)

Table 35 specifies the *AddressSpace* representation for the *Abort Method*. *Abort* has no input or output parameters and has no referenced objects or variables.

Table 35 – Abort Method AddressSpace Definition

Attribute	Value				
BrowseName	Abort				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**

6.7.6 Clear Method

This *Method* is used as part of the *PackMLBaseStateMachineType*. It allows an OPC UA *Client* to change the state of this state machine to the *Cleared* state.

Signature

```
Clear(
    );
```

Method result codes are defined in Table 36.

Table 36 - Clear method result codes

Result Code	Description
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.)

Table 37 specifies the *AddressSpace* representation for the *Clear Method*. *Clear* has no input or output parameters and has no referenced objects or variables.

Table 37 – Clear Method AddressSpace Definition

Attribute	**Value**				
BrowseName	Clear				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**

6.7.7 Stop Method

This *Method* is used as part of the *PackMLMachineStateMachineType*. It allows an OPC UA *Client* to change the state of this state machine to the *Stopping* state.

Signature

```
Stop(
    );
```

Method result codes are defined in Table 38

Table 38 - Stop Method result codes

Result Code	**Description**
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.)

Table 39 specifies the *AddressSpace* representation for the *Stop Method*. *Stop* has no input or output parameters and has no referenced objects or variables.

Table 39 – Stop Method AddressSpace Definition

Attribute	**Value**				
BrowseName	Stop				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**

6.7.8 Reset Method

This *Method* is used as part of the *PackMLExecuteStateMachineType*. It allows an OPC UA *Client* to change the state of this state machine to the *Resetting* state.

Signature

```
Reset(
    );
```

Method result codes are defined in Table 40.

Table 40 - Reset Method result codes

Result Code	**Description**
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.)

Table 41 specifies the *AddressSpace* representation for the *Reset Method*. *Reset* has no input or output parameters and has no referenced *Objects* or *Variables*.

Table 41 – Reset Method AddressSpace Definition

Attribute	**Value**				
BrowseName	Reset				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**

6.7.9 ToCompleted Method

This *Method* is used as part of the *PackMLExecuteStateMachineType*. It allows an OPC UA *Client* to change the state of this state machine to the *Resetting* state.

Signature

```
ToCompleted(
    );
```

Method result codes are defined in Table 42.

Table 42 - ToCompleted Method result codes

Result Code	Description

Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.)

Table 43 specifies the *AddressSpace* representation for the *ToComplete Method*. *Complete* has no input or output parameters and has no referenced objects or variables.

Table 43 – ToCompleted Method AddressSpace Definition

Attribute	**Value**				
BrowseName	ToCompleted				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**

6.7.10 Start Method

This *Method* is used as part of the *PackMLExecuteStateMachineType*. It allows an OPC UA *Client* to change the state of this state machine to the *Starting* State and send parameters at the same time. Which is an extension to the ISA-TR88.00.02-2015 specification. The parameter is defined in Table 44

Signature

```
Start(
     [in] PackMLDescriptorDataType Parameter[]
    );
```

Table 44 - Start Method Parameters

Argument	**Description**
`Parameter`	The array of parameter that can be used by the method

Method result codes are defined in Table 45.

Table 45 - Start Method result codes

Result Code	**Description**
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.)
Bad_ArgumentsMissing	See OPC 10000-4 – Services for the description of this result code (The Client did not specify all of the input arguments for the *Method*.)
Bad_TooManyArguments	See OPC 10000-4 – Services for the description of this result code (The Client specified more input arguments than defined for the *Method*.)
Bad_InvalidArgument	See OPC 10000-4 – Services for the description of this result code. (Used to indicate in the operation level results that one or more of the input arguments are invalid. The inputArgumentResults contain the specific status code for each invalid argument.)
Bad_TypeMismatch	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that an input argument does not have the correct data type.)

Table 46 specifies the *AddressSpace* representation for the *Start Method*. *Start* includes an array of *InputArguments*, where the input argument details are provided in Table 44.

Table 46 – Start Method AddressSpace Definition

Attribute	**Value**				
BrowseName	Start				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory

6.7.11 Unhold Method

This *Method* is used as part of the *PackMLExecuteStateMachineType*. It allows an OPC UA *Client* to change the state of this state machine to the *Unholding* state.

Signature

```
Unhold(
    );
```

Method result codes are defined in Table 47

Table 47 - Unhold Method result codes

Result Code	**Description**
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.)

Table 48 specifies the *AddressSpace* representation for the *Unhold Method*. *Unhold* has no input or output parameters and has no referenced objects or variables.

Table 48 – Unhold Method AddressSpace Definition

Attribute	**Value**				
BrowseName	Unhold				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**

6.7.12 Suspend Method

This *Method* is used as part of the *PackMLExecuteStateMachineType*. It allows an OPC UA *Client* to change the state of this state machine to the *Suspending* state.

Signature

```
Suspend(
    );
```

Method result codes are defined in Table 49

Table 49 - Suspend Method result codes

Result Code	Description
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.)

Table 50 specifies the *AddressSpace* representation for the *Suspend Method. Suspend* has no input or output parameters and has no referenced objects or variables.

Table 50 – Suspend Method AddressSpace Definition

Attribute	**Value**				
BrowseName	Suspend				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**

6.7.13 Unsuspend Method

This *Method* is used as part of the *PackMLExecuteStateMachineType*. It allows an OPC UA *Client* to change the state of this state machine to the *Unsuspending* state.

Signature

```
    Unsuspend(
        );
```

Method result codes are defined in Table 51

Table 51 - Unsuspend Method result codes

Result Code	**Description**
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.)

Table 52 specifies the *AddressSpace* representation for the *Unsuspend Method. Unsuspend* has no input or output parameters and has no referenced *Objects* or *Variables*.

Table 52 – Unsuspend Method AddressSpace Definition

Attribute	**Value**				
BrowseName	Unsuspend				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**

6.7.14 Hold Method

This *Method* is used as part of the *PackMLExecuteStateMachineType*. It allows an OPC UA *Client* to change the state of this state machine to the *Holding* state.

Signature

```
    Hold(
        );
```

Method result codes are defined in Table 53

Table 53 - Hold Method result codes

Result Code	**Description**
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.)
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)

Table 54 specifies the *AddressSpace* representation for the *Hold Method*. *Hold* has no input or output parameters and has no referenced objects or variables.

Table 54 – Hold Method AddressSpace Definition

Attribute	**Value**				
BrowseName	Hold				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**

6.7.15 RemoteCommand Method

This *Method* is used to issue a command to the UA *Server* that can then be passed on to either other internal system as illustrated in Figure 19

Figure 19 - Remote Command and Internal systems

or it can be used to pass information on to an upstream or downstream system as illustrated in Figure 20. In both cases, it is up to the UA *Server* and/or the underlying system to determine when the command is actually passed on.

Figure 20 - Remote Command and Upstream/Downstream systems

The *RemoteCommand* Method parameters are defined in Table 55

Signature

```
    RemoteCommand (
        [in] Int32 Number,
        [in] Int32 ControlCmdNumber,
        [in] Int32 CmdValue,
        [in] PackMLDescriptorDataType [] Parameter
    );
```

Table 55 - RemoteCommand Method Parameters

Argument	**Description**
Number	This is the unique number for the downstream/upstream unit machine using a common tag structure as the unit machine. The number should correspond to a number on the communication network, such as network ID, or IP address identifier. This number corresponds to the "information sender" that is setting the command data in the RemoteInterface[#] structure of the unit machine.
ControlCmdNumber	A user defined command number associated with coded value from a remote unit. This number is a coded value sent from one node on the network to another. The value can be associated with a unit mode change request, speed change request, a state change request, etc.

CmdValue	This is the command value associated with the ControlCmdNumber above. The command value may be the speed requested, state change, etc.
	
	Example:
	
	For an upstream machine designated as #2 a control command number of 5 may be related to the speed setting value for the machine. A value of 400 can be used to modify the remote machine setpoint.
	
	Command.RemoteInterface[1].Number = 2
	
	Command.RemoteInterface[1].ControlCmdNumber = 5
	
	Command.RemoteInterface[1].CmdValue = 400

Parameter	The parameter tags associated to commanded remote interface are typically used for command parameters that are given to the unit machine from remote machines. The parameters are typically needed for coordinating the unit machine or production with other machines. The parameter value may be anything from machine limit parameters to temperatures and counter presets. The parameters are typically limited to machine parameters as product and process parameters are described in later tags.

Method result codes are defined in Table 56.

Table 56 - RemoteCommand Method result codes

Result Code	**Description**
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.)
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_ArgumentsMissing	See OPC 10000-4 – Services for the description of this result code (The Client did not specify all of the input arguments for the *Method.*)
Bad_TooManyArguments	See OPC 10000-4 – Services for the description of this result code (The Client specified more input arguments than defined for the *Method.*)
Bad_InvalidArgument	See OPC 10000-4 – Services for the description of this result code. (Used to indicate in the operation level results that one or more of the input arguments are invalid. The inputArgumentResults contain the specific status code for each invalid argument.)
Bad_TypeMismatch	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that an input argument does not have the correct data type.)

Table 57 specifies the *AddressSpace* representation for the *RemoteCommand Method. RemoteCommand* includes an array of *InputArguments*, where the input argument details are provided in Table 55.

Table 57 – RemoteCommand Method AddressSpace Definition

Attribute	Value				
BrowseName	RemoteCommand				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory

6.7.16 SetInterlock Method

This *Method* allows an OPC UA *Client* to set an interlock associated with this PackML system. The parameters for the *SetInterlock Method* are defined in Table 58.

Signature

```
SetInterlock(
     [in] NodeId InterlockId,
     [in] Boolean State);
```

Table 58 - SetInterlock Method Parameters

Argument	Description
InterlockId	The NodeId of the interlock to set or reset.
State	The state that the targeted interlock should be set to. True is set to interlocked, false is not interlocked.

Method result codes are defined in Table 59

Table 59 - SetInterlock Method result codes

Result Code	Description
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.)
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_ArgumentsMissing	See OPC 10000-4 – Services for the description of this result code (The Client did not specify all of the input arguments for the *Method.*)
Bad_TooManyArguments	See OPC 10000-4 – Services for the description of this result code (The Client specified more input arguments than defined for the *Method.*)
Bad_InvalidArgument	See OPC 10000-4 – Services for the description of this result code. (Used to indicate in the operation level results that one or more of the input arguments are invalid. The inputArgumentResults contain the specific status code for each invalid argument.)
Bad_TypeMismatch	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that an input argument does not have the correct data type.)

Table 60 specifies the *AddressSpace* representation for the *SetInterlock Method. SetInterlock* includes an array of *InputArguments*, where the input argument details are provided in Table 58.

Table 60 – SetInterlock Method AddressSpace Definition

Attribute	Value				
BrowseName	SetInterlock				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory

6.7.17 SetParameter Method

This *Method* allows an OPC UA *Client* to set the parameters that are by the machine.

Signature

```
SetParameter(
     [in] PackMLDescriptorDataType[] Parameter
    );
```

Table 61 - SetParameter Method Parameters

Argument	Description
Parameter	The array of parameter that can be used by the method

Method result codes are defined in Table 62

Table 62 - SetParameter Method result codes

Result Code	Description
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)
Bad_NotImplemented	See OPC 10000-4 – Services for the description of this result code. (Requested operation is not implemented.)
Bad_NodeIdUnknown	See OPC 10000-4 – Services for the description of this result code. (Used to indicate that the specified Object is not valid)
Bad_InvalidState	See OPC 10000-4 – Services for the description of this result code. (The operation cannot be completed because the Object is closed, uninitialized or in some other invalid state.). If a machine determines that it is not in a state that allows parameter changes this error is returned.
Bad_MethodInvalid	See OPC 10000-4 – Services for the description of this result code. (The *Method* id does not refer to a *Method* for the specified Object.)

Table 63 specifies the *AddressSpace* representation for the *SetParmeter Method. SetParameter* includes an array of *InputArguments*, where the input argument details are provided in Table 61.

Table 63 – SetParameter Method AddressSpace Definition

Attribute	Value				
BrowseName	SetParameter				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory

6.8 Alarms

6.8.1 Overview

The section defines alarms. Alarms in PackML are provided via a set of tags. Alarms in OPC UA are provided via events and a full alarming system. This is a preferred method for providing alarms, but it does require some additional overhead. To allow the PackML information model to be implemented in smaller devices it was decided to initially support the existing Tag based representation of Alarm information, and to add the OPC UA *Event* based definition of alarms in the next release. *Servers* shall be able to report alarms in both manners, and *Clients* can use the manner that is most appropriate for them. Some smaller device might find it easier to just access the tags for Alarm information, but display system or other HMIs would probably make use of the OPC UA *Event* based system for alarming.

6.8.2 Alarm Tags

6.8.2.1 Overview

The following *VariableType* is used to report alarms in the PackML unit. In addition, the data is also available as a structured datatype, which is much like an event.

6.8.2.2 PackMLAlarmVariableType

The *PackMLAlarmVariableType* provides the PackML tag alarm structure as individual variables as well as the structure. It is defined in Table 64.

Table 64 – PackMLAlarmVariableType definition

Attribute	**Value**				
BrowseName	PackMLAlarmVariableType				
IsAbstract	False				
ValueRank	Scalar				
DataType	PackMLAlarmDataType				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseDataVariableType* defined in.					
HasComponent	Variable	ID	Int32	BaseDataVariable	Mandatory
HasComponent	Variable	Value	Int32	BaseDataVariable	Mandatory
HasComponent	Variable	Message	String	BaseDataVariable	Mandatory
HasComponent	Variable	Category	Int32	BaseDataVariable	Mandatory
HasComponent	Variable	DateTime	UtcTime	BaseDataVariable	Mandatory
HasComponent	Variable	AckDateTime	UtcTime	BaseDataVariable	Mandatory
HasComponent	Variable	Trigger	Boolean	BaseDataVariable	Mandatory

6.8.2.3 PackMLAlarmDataType

The PackMLAlarmDataType provides the PackML tag alarm structure. It is formally defined in Table 65. This *DataType* also has a corresponding *VariableType* (see 6.8.2.2).

Table 65 – PackMLAlarmDataType Structure

Name	**Type**	**Description**
PackMLAlarmDataType	Structure	
ID	Int32	A unique number assigned to each type of alarm, stop or warning.
Value	Int32	An alarm, stop or warning message number associated to the ID to allow for user specific detail or to break down the Alarm.ID to greater detail
Message	String	The actual text of the alarm, stop or warning for those machines capable of providing string information
Category	Int32	A user defined value which indicates what type of alarm, stop or warning has occurred. E.g. electrical, mechanical, process limit, …
DateTime	UtcTime	The date and time that the alarm, stop or warning occurred
AckDateTime	UtcTime	The date and time that the alarm, stop or warning was Acknowledged,
Trigger	Boolean	This variable is true when the alarm is active

6.8.3 Alarm Events

[note: This section will be defined in a future release]

7 Profile

7.1 Conformance Unit

7.1.1 Overview

This section defines *ConformanceUnits* that are specific to the OPC UA PackML Information model. These *ConformanceUnits* are separated into *ConformanceUnits* that are *Server* specific and those that are Client specific.

7.1.2 Server

Table 66 defines the Server based *ConformanceUnits*.

Table 66 – PackML Server Information Model

Category	Title	Description
Server	PackML Base Functionality	The server supports the *BaseObjectModel*. This includes exposing all mandatory objects, variables and methods.
Server	PackML Base TagID	The Server supports the TagID Property
Server	PackML Base Version	The Server supports the Version property and DataType
Server	PackMLBase Admin	The Server support all of the mandatory items in the PackMLAdminObjectType
Server	PackML Base Status	The Server support all of the mandatory items in the PackMLStatusObjectType
Server	PackML State Information	The server supports the *BaseStateMachine*. This include the list of AvailableStates and AvailableTransitions. It also includes all mandatory states and any method associated with the states. The certification will include a list of all states and transitions supported by the *StateMachine*. This include the mandatory Sub-*statemachines*
Server	PackML State Abort	The server supports the Abort Method
Server	PackML State Clear	The server supports the Clear Method
Server	PackML State Stop	The server supports the Stop Method
Server	PackML State Reset	The server supports the Reset Method
Server	PackML State Complete	The server supports the Complete Method
Server	PackML State Start	The server supports the Start Method
Server	PackML State Unhold	The server supports the Unhold Method
Server	PackML State Suspend	The server supports the Suspend Method
Server	PackML State Hold	The server supports the Hold Method
Server	PackML State Unsuspend	The server supports the Unsuspend Method
Server	PackML SetUnitMode	The Server supports the SetUnitMode Method
Server	PackML Set Product	The Server supports the SetProduct Method
Server	PackML Machine Speed	The Server supports the SetMachSpeed Method
Server	PackML RemoteCommand	The Server supports the RemoteCommand Method
Server	PackML SetInterlock	The Server supports the SetInterlock Method
Server	PackML Status UnitModeRequested	The Server support the UnitModeChangeInProgress flag
Server	PackML Status UnitModeChangeInProgress	The Server support the UnitModeChangeInProgress flag
Server	PackML Status State	The Server supports the StateRequested and StateChangeInProgress flag
Server	PackML Status Interlock	The Server includes support for at least one MaterialInterlock and the summary MaterialInterlocked flag.
Server	PackML Status RemoteParamaters	The Server supports exposing of the configured RemoteParameters
Server	PackML Status Product	The Server supports exposing of the configured product data
Server	PackML Admin Alarm	The server supports the PackML Alarm instance including the Alarm extent.
Server	PackML Admin Alarm History	The server supports the PackML AlarmHistory including the Alarm history extent.
Server	PackML Admin Warning	The server supports the PackML warning instance including the warning extent.
Server	PackML Admin Stop	The server supports the PackML stop reason including the