rexroth
A Bosch Company

ctrlX AUTOMATION: PackML

CXA_PackML_Toolkit

Edition 01

1 PackML

1.1 Background

PackML, or Packaging Machine Language, is an effort by OMAC[1] to standardize the interface between a packaging machine and its peers and supervisors. Primary objectives include bringing common ”look and feel” and operational consistency to all machines that make up a packaging line.

PackML provides:

- Overall equipment effectiveness or OEE data
- Root cause analysis or RCA data
- Recipe management
- Standard defined machine states and operational flow

ISA-88, the larger batch industry standard upon which PackML is based, defines a so-called physical model, which describes the hierarchy of physical assets of a commercial enterprise.

Enterprise: Entity or company that owns the facilities

Site: Facility or plant floor

Area: Subdivision of the site or plant floor

Production Line or Process Cell: Set of machines (see UN below) linked together to produce or process something

---

Unit/Machine (UN): Collection of related modules (see EM below) that carry out one or more processing activities

Equipment Module (EM): Group of control modules operating together to perform a specific function

Control Module (CM): The lowest level of control where a single function is executed.

Although not required by ISA-TR88.00.02, a common design pattern when implementing PackML is to utilize a modular program architecture based on the lower three levels of the ISA-88 physical model. According to this hierarchy, a machine, or unit/machine, is comprised of a set of equipment modules, each of which is further divided into a set of control modules. Infeed stations, sorters, fillers, coaters might be examples of equipment modules. An individual servo axis driving a sorter, say, would be a typical control module. How exactly one subdivides a machine into equipment modules is a matter of experience, convenience and to some extent, whim. In most cases there is not a single correct answer. In any case, it is often preferable for the software to be written in a modular way that reflects this subdivision.

CXA_PackML_Toolkit provides a set of tools to support PackML and the modular design pattern based on the ISA-88 physical model. The library includes a PackML state/mode handler and a modular event handler which folds nicely into this design pattern.

[1] Organization for Machine Automation and Control. For more information see www.omac.org.

2 Disclaimer

Use of this software is permitted only under the terms and conditions defined in article *Terms and Conditions for the Provision of Products of Bosch Rexroth AG Free of Charge*, included in the software installation package.

The software is licensed under the MIT License. Source code available by request.

3 Terms and definitions

PackML: a communication interface for discrete manufacturing (e.g. packaging machinery) described by ANSI/ISA-TR88.00.02-2015; sometimes used to describe additional methods of standardizing machine “look and feel” or PLC coding design patterns.

State model (finite): an abstract model or machine that can be in exactly one of a finite number of states at any given time. The model may change or transition from one state to another in response to some input.

Unit/machine control mode: a way or manner in which a manufacturing machine is used. In the context here, common modes include Production, Maintenance, Manual, etc.

Mode manager: software that governs the transition from one machine mode to another.

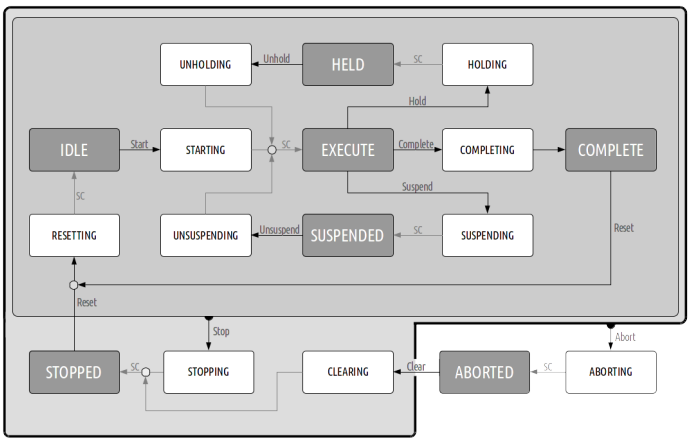
PackTags: a set of naming conventions defined by ISA-TR88.00.02 for software variables associated to data elements of the unit/machine. Where appropriate, the unit of measurement associated with an individual data element is specified.

ISA-88 physical model: a hierarchy describing the physical assets of a commercial enterprise. See ANSI/ISA-S88.01-1995. In this model, a Process Cell comprises one or more Units responsible for producing a specific commodity.

4 State machine handling

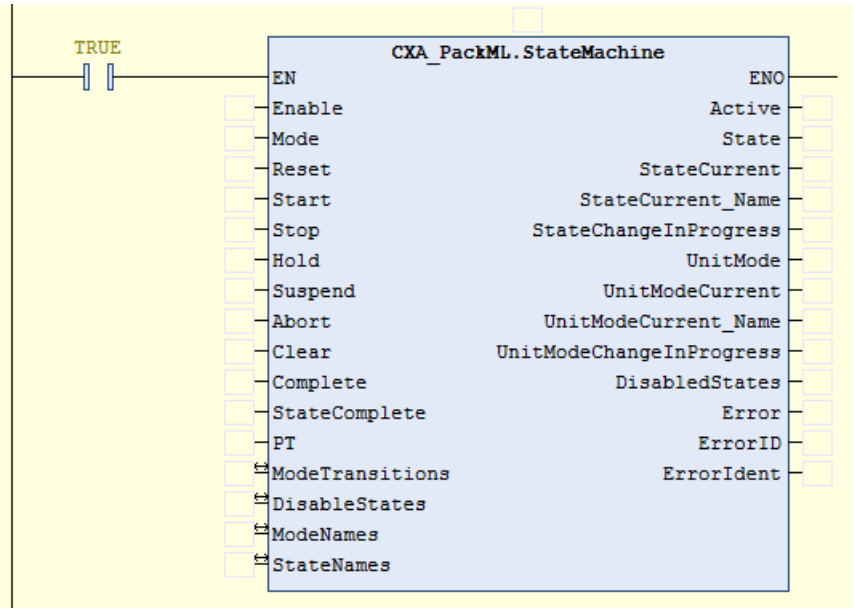
4.1 StateMachine

Function block *StateMachine* provides a configurable state machine following the PackML model. The function block includes a mode manager, supporting up to 32 separate modes. The basic state-model is shown below:



Note that the state model consists of 17 distinct states (ABORTED, CLEARING, STOPPED, RESETTING, etc.) and 11 transition commands (Clear, Reset, Start, Stop, Abort, Hold, Unhold, Suspend, Unsuspend, Complete and StateComplete), where a particular transition initiates action on the state machine only if the associated predecessor is active. For example, if the state machine is in IDLE state, then the Start command will force a transition of the state model from IDLE state to STARTING state. If a state other than IDLE is active, then the Start command has no effect.

For a complete definition of the state model, see ANSI/ISA-TR88.00.02-2015. Copies of the document may be ordered from www.isa.org.



I/O	*Name*	*Type*	*Comment*
VAR_IN_OUT	ModeTransitions	ARRAY[0..x] UDINT	Defines for each mode states for which mode transitions are allowed. Array element 1 corresponds to mode 1. State permissives are defined bitwise, with a value of one indicating that a given mode transition is allowed. See section below for more detailed information. Note that x is defined by library parameter C_UNIT_MODE_COUNT.
	DisableStates	ARRAY[0..x] UDINT	Defines for each mode the states which are disabled. State status is defined bitwise, with a bit value of one indicating that the state is disabled. See section below for more detailed information.
	ModeNames	ARRAY[0..x] STRING	User-definable mode names.
	StateNames	ARRAY[0..17] STRING	User-definable state names.

I/O	*Name*	*Type*	*Comment*
VAR_INPUT	Enable	BOOL	Function block enable
	Mode	DINT	Commanded mode
	Reset	BOOL	Transition from STOPPED or COMPLETE to RESETTING
	Start	BOOL	Transition from IDLE to STARTING
	Stop	BOOL	Transition from active state to STOPPING. Does not apply to STOPPED, CLEARING, ABORTED or ABORTING states.
	Hold	BOOL	Transition from Execute to Holding. Unhold when false.
	Suspend	BOOL	Transition from Execute to Suspending. Unsuspend when false.
	Abort	BOOL	Transition from active state to ABORTING. Does not apply to ABORTED state.
	StateComplete	BOOL	Transition from any ACTING state to next state.

I/O	*Name*	*Type*	*Comment*
VAR_OUTPUT	Active	BOOL	Function block active
	State	DINT	State status defined bitwise.
	StateCurrent	DINT	State status defined as numerical sequence.
	StateCurrent_Name	STRING	User-defined name associated to current state. See StateNames.
	StateChangeIn Progress	BOOL	True for single scan upon state change
	UnitMode	DINT	Mode status defined bitwise.
	UnitModeCurrent	DINT	Mode status defined as numerical sequence.
	UnitModeCurrent _Name	STRING	User-defined name associated to current mode. See ModeNames.
	UnitModeIn Progress	BOOL	True for single scan upon mode change.
	DisabledStates	UDINT	Disabled states, defined bitwise, in current mode.
	Error	BOOL	Internal error.
	ErrorID	ERROR_CODE	ID associated to current error.
	ErrorIdent	ERROR_STRUCT	Additional information associated to current error.

Comments

1. ISA–TR88.00.02 enumerates the states used in the PackML model as follows:

0 Undefined	6 Execute	12 Unholding
1 Clearing	7 Stopping	13 Suspending
2 Stopped	8 Aborting	14 Unsuspending
3 Starting	9 Aborted	15 Resetting
4 Idle	10 Holding	16 Completing
5 Suspended	11 Held	17 Complete

This enumeration is used explicitly in the StateCurrent output, but also used implicitly when defining the bitwise output State. For example, if the machine is in IDLE state, then

StateCurrent = 4, State = b0000 0000 000**1** 0000.

2. ISA-TR88.00.02 reserves the following modes:

0 Undefined

1 Production

2 Maintenance

3 Manual

Additional modes are not named by the standard. Mode names may be defined by the user in ModeNames.

As in the case above, if Manual mode is active, then

UnitModeCurrent = 3, ModeCurrent = b0000 0000 0000 **1**000.

3. Mode transitions may be initiated using input Mode. Mode transitions are normally allowed only when the machine is not running (e.g. STOPPED or ABORTED states), but ISA_TR88.00.02 does not provide any definite specification here. The function block itself allows the user to configure without restriction states for which mode changes are allowed. To allow a mode transition in a given state, the bit associated to this state must be true for both ModeTransitions[*current mode*] and ModeTransitions[*target mode*].

For example, if the machine is currently producing mode (UnitModeCurrent = 1), aborted state (StateCurrent = 9), then the machine may be transitioned to manual mode (UnitModeCurrent = 3) provided:

ModeTransitions[1] = bxxxx xx**1**x xxxx xxxx, ModeTransitions[3] = bxxxx xx**1**x xxxx xxxx

If bit 9 of either ModeTransitions[1] or ModeTransitions[3] is zero, then the mode change is not allowed. In this case, changing Mode from a value of 1 to a value ot 3 will result in an error, with

Error = TRUE
ErrorID = OTHER_ERROR
ErrorIdent.Table = USER1_TABLE
ErrorIdent.Additional1 = 16#A0000001

StateMachine functions as a stand-alone state model and has no internal or implicit interaction with any other program components. The basic assumption is that the user will manage how code or machine functionality is called based on the global machine state.

Multiple instances of the function block may be employed without issue, for example in cases where multiple machines are controlled by a single PLC, but in these cases the user must handle each state model separately.

4. CXA_PackML_Toolkit provides a sample data structure, PACKML_STATE_TYPE that simplifies the use of output State. The data structure is defined as

```
Type PACKML_STATE_TYPE
UNION
  B: PACKML_STATES;
  EN: PACKML_STATES_ENUM;
END_UNION
END_TYPE
```

Here type PACKML_STATES_ENUM is the enumeration described above and PACKML_STATES is the bitwise listing of the PackML states as described previously:

```
TYPE PACKML_STATES:
STRUCT
  UNDEFINED: BIT;
  CLEARING: BIT;
  STOPPED: BIT;
  STARTING: BIT;
  IDLE: BIT;
  SUSPENDED: BIT;
  EXECUTE: BIT;
  STOPPING: BIT;
  ABORTING: BIT;
  ABORTED: BIT;
  HOLDING: BIT;
  HELD: BIT;
  UNHOLDING: BIT;
  SUSPENDING: BIT;
  UNSUSPENDING: BIT;
  RESETTING: BIT;
  COMPLETING: BIT;
  COMPLETE: BIT;
END_STRUCT
END_TYPE
```

Assume stPML_State is a variable of type PACKML_STATE_TYPE. If we assign StateMachine.State to stPML_State.EN, we may then access the current PackML state using the bit structure. For example, our state machine is in Starting state if and only if stPML_State.B.STARTING = TRUE.

4. StateMachine includes a helper method, *setDefaultConfig*, that will pre-assign to inputs ModeTransitions, DisableStates, ModeNames and StateNames canonical values.

```
StateMachine_0.setDefaultConfig(ModeTransitions, DisableStates, ModeNames, StateNames);
```

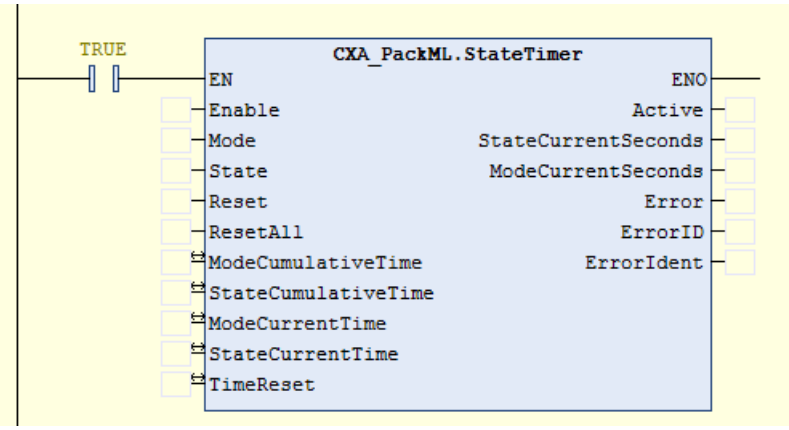
ModeNames and StateNames are assigned names as described in Comments 1 and 2 above. In Production mode, no states are disabled. In Maintenance mode, Suspended and Complete and their associated transitions are disabled. In Manual mode, Suspended, Complete and Held status plus their associated transitions are disabled. Mode transitions are allowed in Idle, Stopped and Aborted states.

Use of *setDefaultConfig* is optional. Alternately, users can configure these arrays manually as desired.

StateTimer

StateTimer records the period of time the state machine remains in any given state or mode as specified in ISA-TR88.00.02. Both cumulative and current times are recorded, where by current we mean “active or most recent”. (Specifically, the current timers are reset each time a given state is re-entered.)

All times are recorded in whole seconds. Note that elapsed times less than one are truncated. Array elements follow the PackML state order as defined in the previous section.



I/O	*Name*	*Type*	*Comment*
VAR_IN_OUT	ModeCumulative Time	ARRAY[0..x] DINT	Cumulative elapsed time in seconds per mode. Array element [1] corresponds to Producing mode. Note that x is defined by library parameter C_UNIT_MODE_COUNT.
	StateCumulative Time	ARRAY[0..x,0..17] UDINT	Cumulative elapsed time in seconds per mode and state. Array element [1,1] corresponds to producing mode, clearing state.
	ModeCurrentTime	ARRAY[0..x] STRING	Current elapsed time in seconds per mode. Array element [1] corresponds to Producing mode. Array values are reset upon reentering corresponding mode.
	StateCurrentTime	ARRAY[0..x,0..17] STRING	Current elapsed time in seconds per mode and state. Array element [1,1] corresponds to Producing mode, Clearing state. Array values are reset upon reentering corresponding mode and state.
	TimeReset	DINT	Elapsed time since last Reset or ResetAll.

I/O	*Name*	*Type*	*Comment*
VAR_INPUT	Enable	BOOL	Function block enable
	Mode	DINT	Active PackML mode.
	State	DINT	Active PackML state.
	Reset	BOOL	Reset current and cumulative timers for active mode.
	ResetAll	BOOL	Reset all timers.

I/O	*Name*	*Type*	*Comment*
VAR_OUTPUT	Active	BOOL	Function block active
	StateCurrentSeconds	DINT	Elapsed time in current state.
	StateCurrentMode	DINT	Elapsed time in current mode.
	Error	BOOL	Internal error.
	ErrorID	ERROR_CODE	ID associated to current error.
	ErrorIdent	ERROR_STRUCT	Additional information associated to current error.

Comments

Comments:

1. In order to maintain timer values through power cycles, cumulative and current timer tags should be defined as RETAIN PERSISTENT.

2. Timer values are specified in ISA-TR88.00.02 as DINT, not UDINT, presumably to support a wider variety of control platforms. Because of this, timer values are capped at 2,147, 483,647 and automatically reset to zero once this threshold is reached. The following warnings are supported:

adStateCumulativeTime rollover warning!

Error = TRUE
ErrorID = OTHER_ERROR
ErrorIdent.Table = USER1_TABLE
ErrorIdent.Additional1 = 16#A0000001

adModeCumulativeTime rollover warning!

Error = TRUE
ErrorID = OTHER_ERROR
ErrorIdent.Table = USER1_TABLE
ErrorIdent.Additional1 = 16#A0000002

dTimeReset rollover warning!

Error = TRUE
ErrorID = OTHER_ERROR

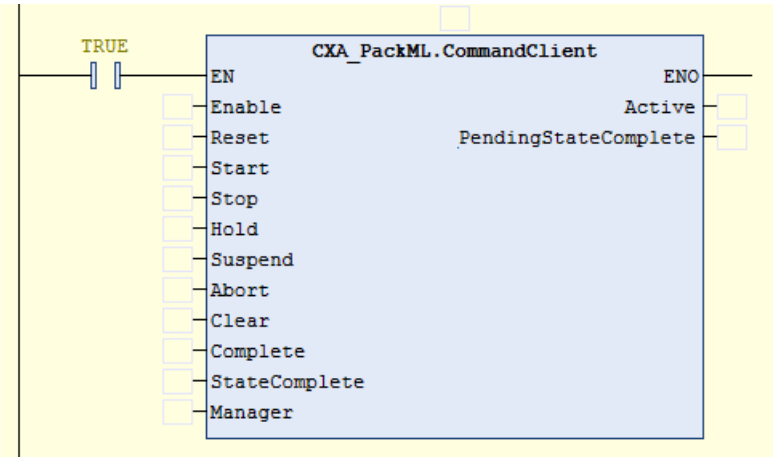
ErrorIdent.Table = USER1_TABLE

ErrorIdent.Additional1 = 16#A0000003

3. StateTimer operates completely independently from any other PackML functionality described here and may be omitted if desired. In addition, multiple instances are allowed in cases where multiple state machines are present in the code.

5 CommandClient

CommandClient, together with its companion function block, *CommandManager*, provide a modular mechanism for communicating state commands (Start, Reset, Stop, etc.) from the control module level through the equipment module level to the unit/machine level. In addition, the StateComplete status from each module may also be communicated. The advantage here is that there is no need to sum the individual state commands explicitly over the set of submodules. This point will be returned to in a later section. See Appendix: Make2Pack.



I/O	*Name*	*Type*	*Comment*
VAR_INPUT	Enable	BOOL	Function block enable
	Reset	BOOL	Issue Reset command to Manager
	Start	BOOL	Issue Start command to Manager
	Stop	BOOL	Issue Stop command to Manager
	Hold	BOOL	Issue Hold command to Manager
	Suspend	BOOL	Issue Suspend command to Manager
	Abort	BOOL	Issue Abort command to Manager
	Clear	BOOL	Issue Clear command to Manager
	Complete	BOOL	Issue Complete command to Manager
	StateComplete	BOOL	Indicate StateComplete to Manager
	Manager	ICommand	Manager: May be an instance of CommandClient or CommandManager

I/O	*Name*	*Type*	*Comment*
VAR_OUTPUT	Active	BOOL	Function block active
	PendingState Complete	UDINT	Bitwise indication of StateComplete status over subscribed clients.

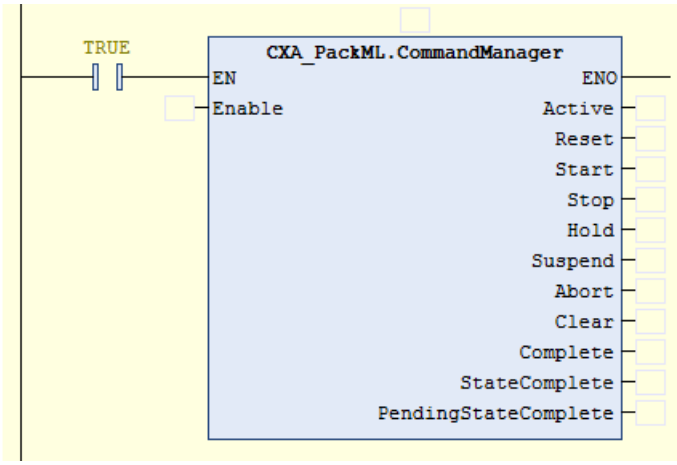
Comments:

1. ComandClient may act as both client and manager. For example, an instance of this function block on the control module level may report to another instance of this block on the equipment module level. The number of instances of CommandClient reporting to a given instance of CommandClient is limited to library parameter C_EQUIPMENT_MANAGER_COUNT.

2. PendingStateComplete describes the status of the StateComplete input of each subscribed client. Specifically, if the first subscribed client or any of its subclients is not StateComplete, then bit 0 of PendingStateComplete will have a value of 1.

6 CommandManager

CommandManager provides for a top-level summation of all *CommandClient* blocks defined in the program. The design intent is that there is only a single instance of *CommandManager* and that all *CommandClient* report to it, either directly or indirectly via nesting.



I/O	*Name*	*Type*	*Comment*
VAR_INPUT	Enable	BOOL	Function block enable

I/O	*Name*	*Type*	*Comment*
VAR_OUTPUT	Active	BOOL	Function block active
	Reset	BOOL	At least one subscribed CommandClient issuing Reset command
	Start	BOOL	At least one subscribed CommandClient issuing Reset command
	Stop	BOOL	At least one subscribed CommandClient issuing Reset command
	Hold	BOOL	At least one subscribed CommandClient issuing Reset command
	Suspend	BOOL	At least one subscribed CommandClient issuing Reset command
	Abort	BOOL	At least one subscribed CommandClient issuing Reset command
	Clear	BOOL	At least one subscribed CommandClient issuing Reset command
	Complete	BOOL	At least one subscribed CommandClient issuing Reset command
	StateComplete	BOOL	All subscribed CommandClients are StateComplete
	PendingStateComplete	UDINT	Bitwise indication of StateComplete status over subscribed clients.

Comments

1. The outputs of function block CommandManager are not mapped automatically to the corresponding inputs of function block StateManager. This mapping must be done explicitly by the user. (This allows StateMachine to be used as a standalone without CommandClient, CommandManager.)

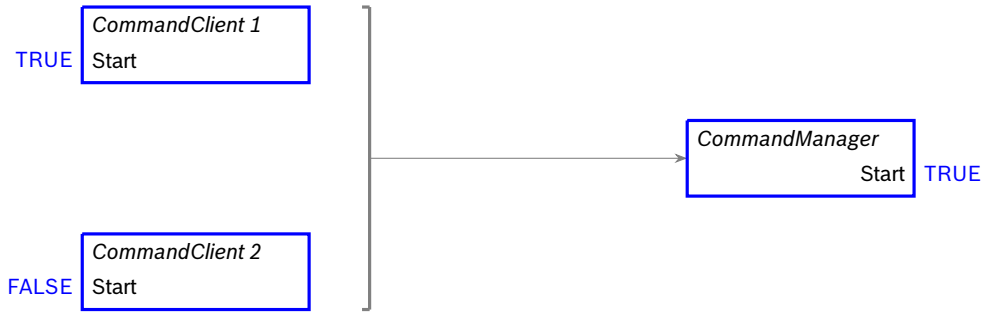


Figure 1: CommandManager output Start reports TRUE if any of the subscribed clients have input Start=TRUE. Inputs Reset, Stop, Hold, Suspend, Abort, Clear and Complete behave similarly. For these inputs the system may be thought of as a *distributed OR*.

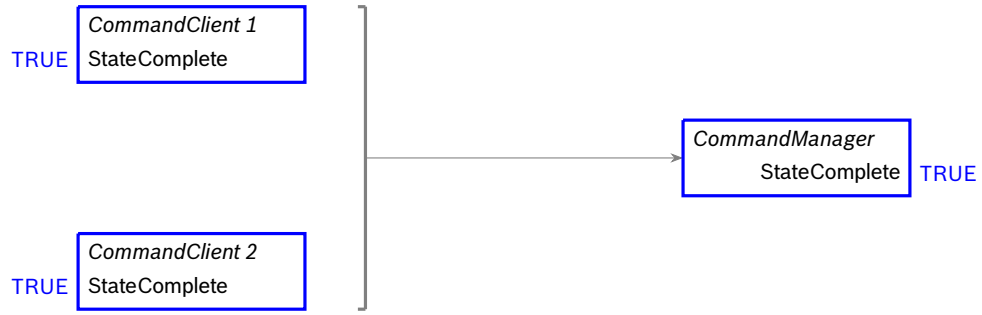


Figure 2: CommandManager output StateComplete reports TRUE if all of the subscribed clients (regardless of nest depth) have input StateComplete=TRUE. This input may be thought of as a *distributed AND*.

7 Event handling

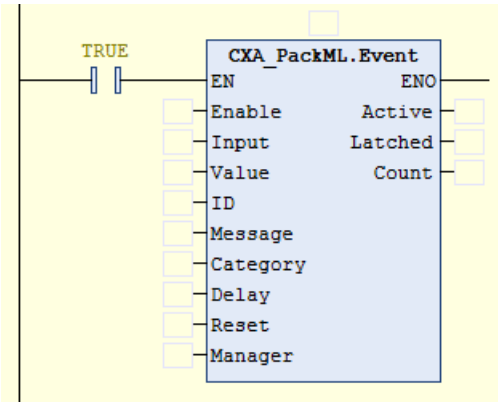
7.1 Event

Each instance of *Event* is responsible for monitoring a condition or set of conditions for which system notification should be made. For example, an instance of *Event* might monitor whether a guard door is open or closed or whether a given piece of hardware has faulted.

The data associated to each event corresponds to the Alarm data type as defined in ISA-TR88.00.02. This basic data type comprises the following components:

```
TYPE ALARM_TYPE:
STRUCT
    Trigger: BOOL;              // Event activation status
    ID: DINT;                   // Unique alarm identifier
    Value: DINT;                // User-defined additional alarm information
    Message: STRING;            // User-defined alarm text message
    Category: DINT;             // Alarm or event type or severity
    TimeEvent: DATE_AND_TIME;   // Time and date stamp describing event occurrence
    TimeAck: DATE_AND_TIME;     // Time and date stamp of event acknowledgement.
END_STRUCT
END_TYPE
```

The Event block is shown below. Note that ID, Value, Message, Category from ALARM_TYPE are defined as configuration inputs. *Event* assigns to the associated instance of *ALARM_TYPE* the assigned values of ID, Value, Message, Category. It also updates Trigger with the current value of Active. The timestamps are handled automatically.



Comments:

1. If an Event instance should auto-reset (i.e. reset automatically on the falling edge of Event.Input), set Event.Reset if a fixed value of TRUE.

2. Event includes a built-in debounce timer, meaning that an event instance will not fire or activate until Event.Input has been TRUE for the period of time defined by Event.Delay. Set Event.Delay to a fixed value of T#0S (zero seconds) if it is desired for the Event to fire directly on the rising edge of Event.Input.

I/O	*Name*	*Type*	*Comment*
VAR_INPUT	Enable	BOOL	Function block enable
	Input	BOOL	Event activation
	Value	DINT	User-specific value associated to given alarm or event
	ID	DINT	Alarm or event ID number. Must be non-zero.
	Message	STRING	Alarm or event message.
	Category	DINT	Alarm or event severity. Only values of 0 - 9 supported.
	Delay	TIME	Delay time of debounce timer.
	Reset	BOOL	Optional local reset. May be set to a fixed value of TRUE for auto-reset.
	Manager	IManager	Instance of EventManager.

I/O	*Name*	*Type*	*Comment*
VAR_OUTPUT	Active	BOOL	Alarm or event active.
	Latched	BOOL	Alarm or event active or not yet reset.
	Count	DINT	Alarm activation counter between Resets.

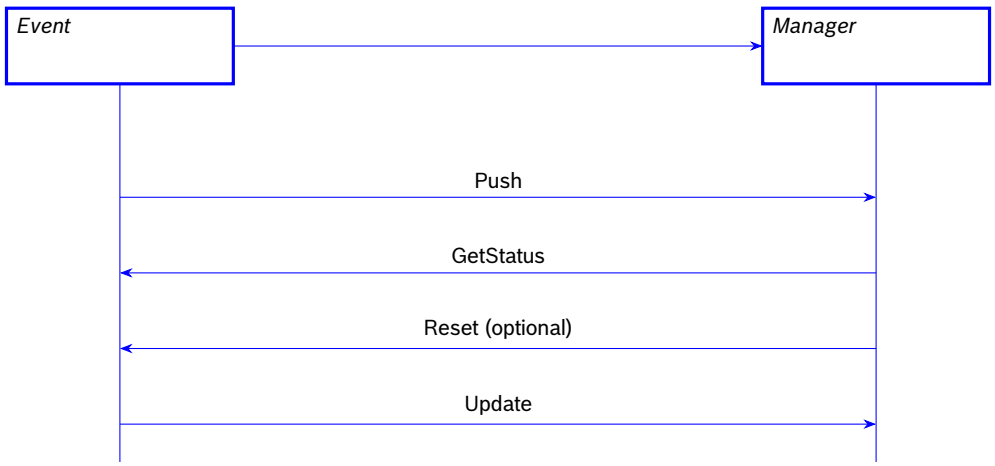
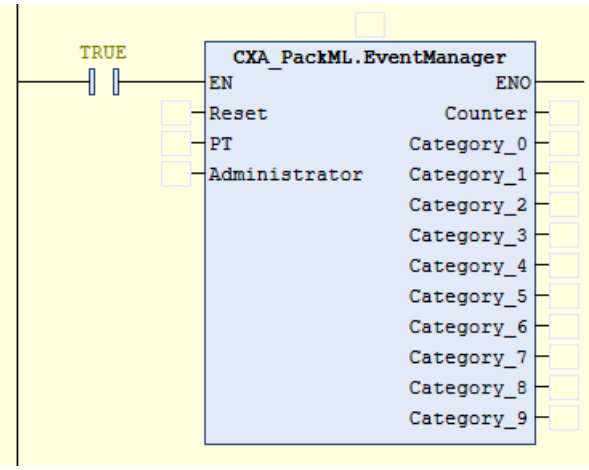


Figure 3: Internal services provided by Event-Manager model. On the rising edge of Event.Input, the Event instance pushes itself onto the Manager's internal linked list. Whenever required, the Manager will request the Event's actual status. Note that Event reset may be initiated locally or by the Manager. If an Event's status changes, it may notify the Manager via Update. These internal interactions are transparent to the user.

7.2 EventManager

An instance of EventManager is responsible for managing the lower-level event blocks (i.e instances of Event that reference this particular manager) and then passing any relevant information to an upper-level instance of EventAdministrator. Typically we would think of EventManager as residing on the equipment module level.

The management duties of this block include resetting the lower-level event blocks, monitoring when these blocks are no longer active, maintaining the active events in chronological order, reporting the status of the active event blocks according to category, etc.



I/O	*Name*	*Type*	*Comment*
VAR_INPUT	Reset	BOOL	Issue Reset to active subscribed events
	PT	TIME	Reset delay; allows for reset of hardware components before reset of associated Event instance
	Administrator	IAdministrator	Instance of EventAdministrator

I/O	*Name*	*Type*	*Comment*
VAR_OUTPUT	Counter	DINT	Count of active or latched subscribed events.
	Category_0	BOOL	At least one active lower-level Event block with Category=0.
	Category_1	BOOL	At least one active lower-level Event block with Category=1.
	Category_2	BOOL	At least one active lower-level Event block with Category=2.
	Category_3	BOOL	At least one active lower-level Event block with Category=3.
	Category_4	BOOL	At least one active lower-level Event block with Category=4.
	Category_5	BOOL	At least one active lower-level Event block with Category=5.
	Category_6	BOOL	At least one active lower-level Event block with Category=6.
	Category_7	BOOL	At least one active lower-level Event block with Category=7.
	Category_8	BOOL	At least one active lower-level Event block with Category=8.
	Category_9	BOOL	At least one active lower-level Event block with Category=9.

Comments:

1. There is no pre-defined limit to the number of event blocks which may subscribe to a given manager. However, the number of events reported by the associated Administrator (i.e. instance of EventAdministrator) is limited by library parameter C_EVENT_DATA_COUNT. See EventAdministrator.EventData. (Outputs Counter and Category_0 - Category_9 are not affected by this limit.)

2. In normal practice input Reset is raised in Clearing and Resetting states. This however is not required and the user is free to reset at any desired time.

3. *EventManager* includes a String property, Prefix, which is concatenated as a prefix to all messages of the lower-level *Event* blocks. The design intent here is to help distinguish the events associated to the various instances of EventManager. If desired, Prefix may be left blank.

```
EventManager_0.Prefix:= 'EM_0';
```

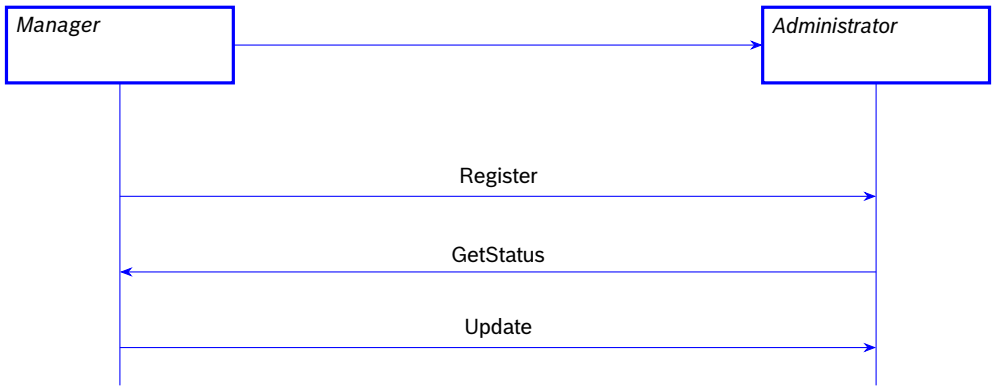
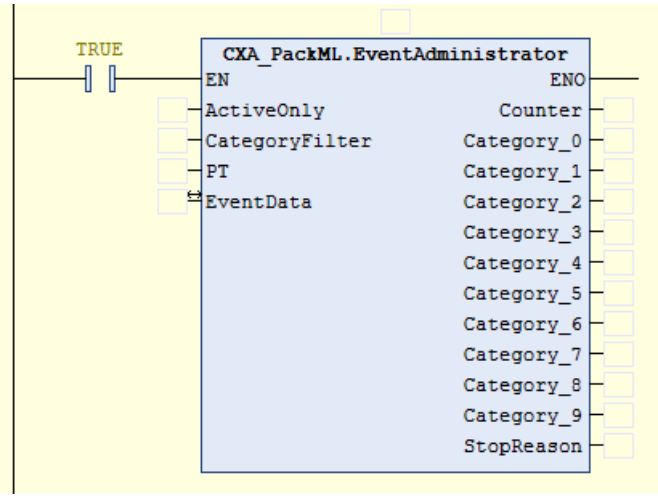


Figure 4: Internal services provided by Manager-Administrator model. On first call, the Manager instance pushes itself onto the Administrator's internal client array. Whenever required, the Administrator will request the Manager's actual status. These internal interactions are transparent to the user.

7.3 EventAdministrator

EventAdministrator collects all active events in each of the subscribed instances of EventManager and sorts them in chronological order. Two filtering mechanisms are also provided, which allow the user to show only active events or show events of particular alarm categories. After processing, the events are stored in array EventData.



I/O	*Name*	*Type*	*Comment*
VAR_IN_OUT	EventData	ARRAY[0..x] ALARM	Array of active events (after filter). Here x is library parameter C_EVENT_DATA_COUNT - 1.

I/O	*Name*	*Type*	*Comment*
VAR_INPUT	ActiveOnly	BOOL	Include in EventData only active events
	CategoryFilter	WORD	Bitwise category filter for EventData. Set to 0b0011 1111 1111 to include all event categories.
	PT	TIME	EventData update timer delay. Set PT = 0 to update EventData on each change in underlying data.

I/O	*Name*	*Type*	*Comment*
VAR_OUTPUT	Counter	DINT	Count of active or latched subscribed events.
	Category_0	BOOL	At least one active lower-level Event block with Category=0.
	Category_1	BOOL	At least one active lower-level Event block with Category=1.
	Category_2	BOOL	At least one active lower-level Event block with Category=2.
	Category_3	BOOL	At least one active lower-level Event block with Category=3.
	Category_4	BOOL	At least one active lower-level Event block with Category=4.
	Category_5	BOOL	At least one active lower-level Event block with Category=5.
	Category_6	BOOL	At least one active lower-level Event block with Category=6.
	Category_7	BOOL	At least one active lower-level Event block with Category=7.
	Category_8	BOOL	At least one active lower-level Event block with Category=8.
	Category_9	BOOL	At least one active lower-level Event block with Category=9.
	StopReason	Alarm	Initial event in EventData.

Comments:

1. The size of the EventData array may be redefined using library parameter C_EVENT_DATA. The default size is 64.

8 Logger

CXA_PackML_Toolkit includes an internal logger which may be used to track the event history. When fired, Events are sent automatically to the logger without intervention by the user. The library includes a single instance of the logger, _log. No other instances should be defined.

8.1 GetStatus()

Use the logger's GetStatus method to access the event history:

```
CXA_PackML._log.GetStatus(Start, EventHistory);
```

Here, EventHistory is a user-defined array of type CXA_PackML.ALARM_TYPE. Users may adjust the size of the array as required, but its size should not exceed that of the logger internal buffer. See library parameter C_EVENT_BUFFER. GetStatus will attempt to fully populate EventHistory with values from the internal buffer, beginning at the Start offset. Set Start=0 to begin with the oldest event entry in the buffer.

8.2 Acknowledge()

To set the AckTimeDate timestamp of the event entries in the log, call the Acknowledge method:

```
CXA_PackML._log.Acknowledge();
```

8.3 Optional buffer

By default, Logger uses an internal buffer instantiated in the library itself. This buffer will be cleared on power cycle. If it is desired to retain this data over a power cycle, an optional, user-defined buffer may be used.

```
VAR_GLOBAL PERSISTENT RETAIN
    Buffer: ARRAY[0..(CXA_PackML.C_EVENT_BUFFER - 1)] OF CXA_PackML.ALARM_TYPE;
END_VAR

CXA_PackML._log(Buffer:= Buffer);
CXA_PackML._log.SetIndex();
```

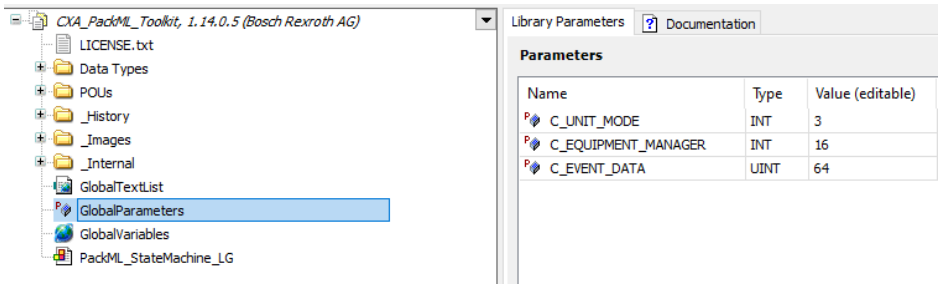
Note that the SetIndex method should be called once on startup after the optional Buffer is defined. If the optional buffer is not used, these steps may be omitted.

9 Library parameters

CXA_PackML_Toolkit includes four library parameters that allow certain critical array sizes to be adjusted by the user. Note that increasing these values will increase the memory and load requirements of the application.

Parameter	*Type*	*Preset*	*Comment*
C_UNIT_MODE	INT	3	Number of unit/machine modes. See StateManager.
C_EQUIPMENT_MANAGER	INT	16	Max. number of equipment managers reporting to unit/machine. See EventAdministrator, CommandManager (used internally).
C_EVENT_DATA	UINT	64	Size of EventData array. See EventAdministrator.
C_EVENT_BUFFER	UINT	1000	Size of event log ring buffer. See Logger.

The values may be edited in the Application's Library Manager. See column *Value (editable)*.



10 Namespace

Library attribute *LanguageModelAttribute* is set to 'qualified-access-only', meaning that library element references must include the namespace (CXA_PackML). For example, to declare an instance of the Event function block write:

```
_event : CXA_PackML.Event;
```

11 Connecting event and state handling

To allow for maximum flexibility, the event and state handlers described in the previous sections operate independently without any internal connections. Users must connect the two explicitly and are free to do so in different ways to suit their needs.

We describe one option here.

First we assign to a given Event category an associated PackML state command. The assignment shown here is non-normative. Users are may categorize events in any convenient way.

Event category	*PackML state command*
0, 1	Abort
2	Stop
5	Hold
6	Suspend
7	Complete

Next we map outputs from the event handler to inputs of the state handler. It is often most convenient to handle this mapping in the equipment manager. See figure below.

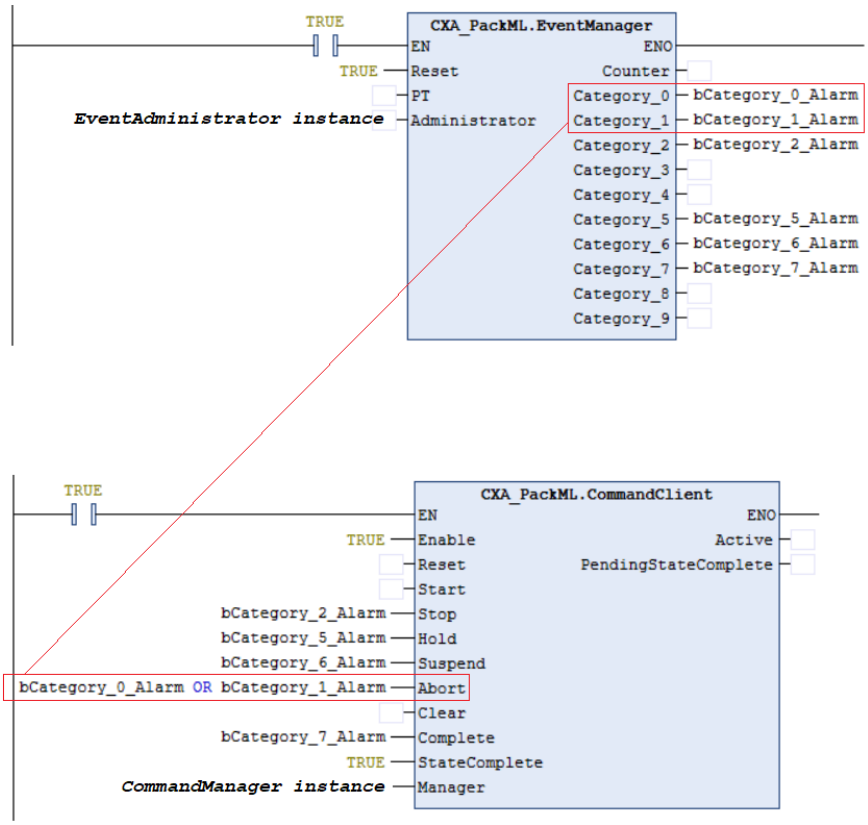


Figure 5: To connect the event and state handlers, we must explicitly tie outputs from the event handler blocks to corresponding inputs of the state handler blocks. Here the user is free to define this correspondence as required.

Similarly, the CommandManager, if it is used, must be explicitly tied to the StateMachine as shown below.

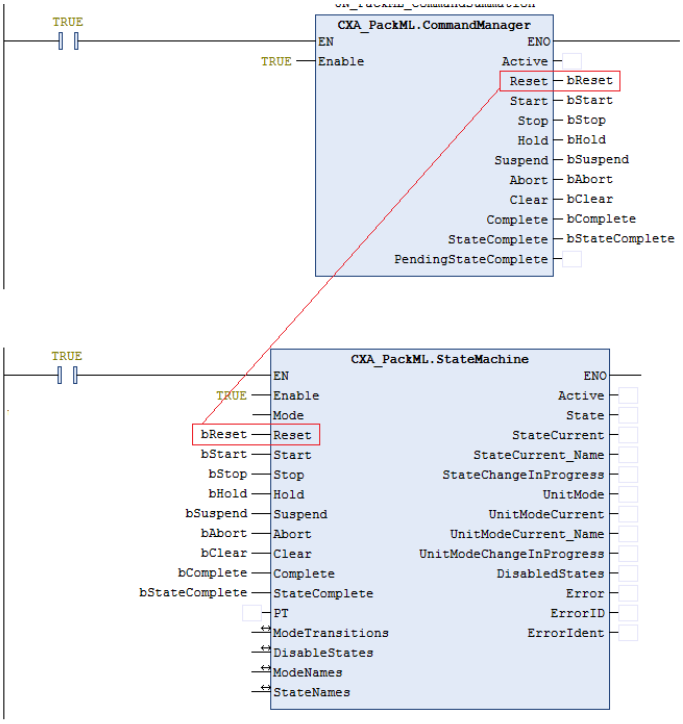


Figure 6: CommandManager is not tied internally to the StateModel. To connect the two, wire the outputs of the CommandManager to the corresponding inputs of the StateModel as shown.

Appendix: Make2Pack

The OMAC PackML Implementation Guide [1] describes a design pattern for modular programming sometimes referred to as *Make2Pack*. The general pattern reflects the hierarchical structure shown below:

Unit/machine (UN)

Equipment module (EM)

Control module (CM)

Here equipment modules would be large subdivisions of the machine (Infeed, Outfeed, Capper, etc.), each comprised of a series of further sub-components called control modules.

CommandClient, together with its parent block, CommandManager, uses a transparent client/ manager scheme to support the Make2Pack design pattern. The PackML transition commands (Reset, Clear, Start, Stop, State-Complete, etc.) may be set within a given control module. These values are then reported automatically to the next level manager, where they are summed and subsequently passed on to the unit/machine. The advantage here is that modules may be added or removed without modification to any global code.

The event handler also follows this conceptual hierarchy.

One natural way to realize this design pattern is to group code for each of the modules described (unit/machine, equipment, control) into individual programs[2]. In this case it is recommended for the calling tree to reflect the Make2Pack hierarchy: The unit/machine program calls each of the equipment module programs; each of the equipment module programs calls the programs associated to each of its control modules.

[2] Here we mean *program* in the specific sense defined by IEC 611131-3.

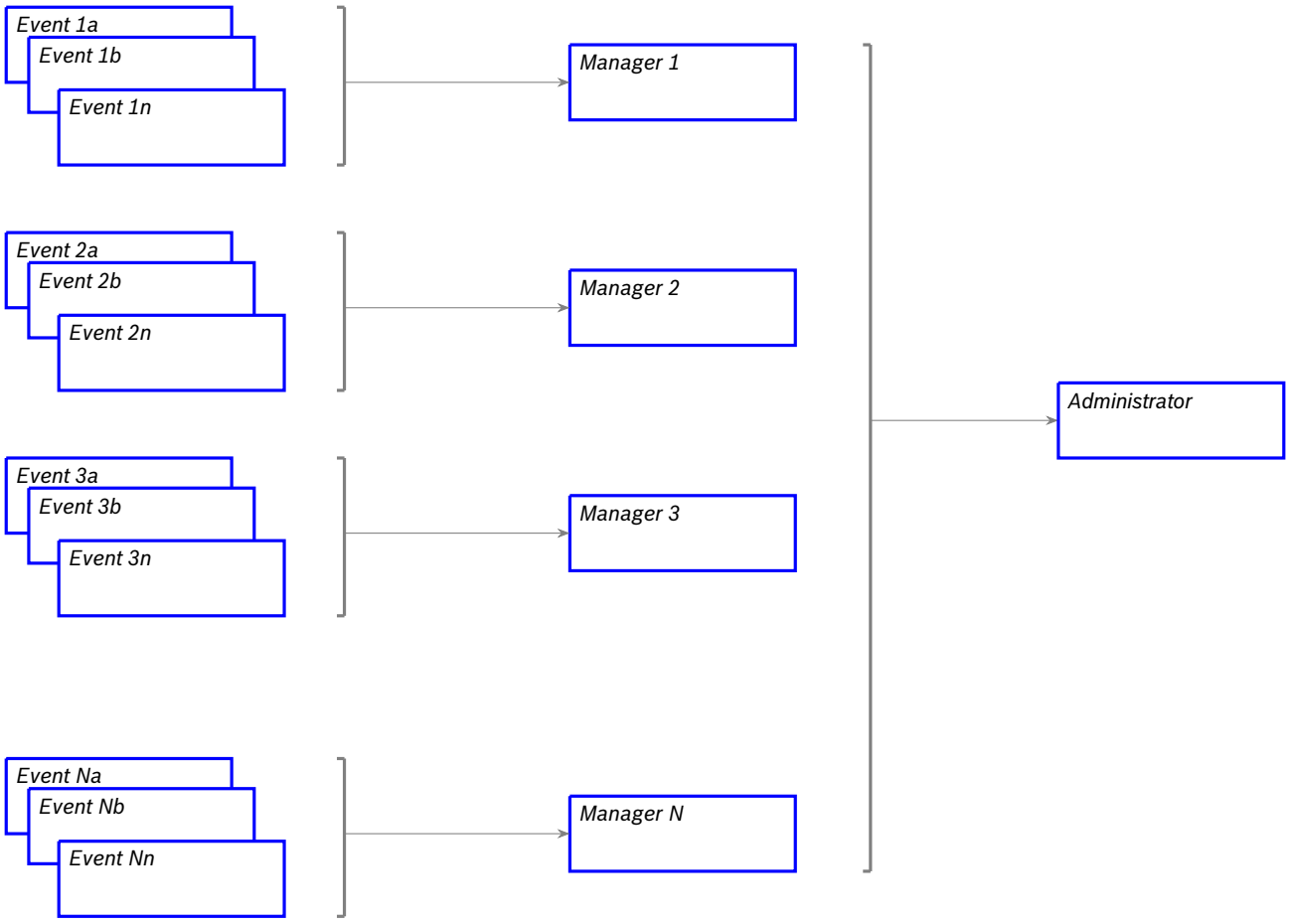


Figure 7: The event handler is divided into a 3-tiered hierarchy: Events report to Managers, which in to report to a single Administrator. Conceptually, Events live on the control module level, Managers live on the equipment module level and the Administrator lives on the unit/machine level.

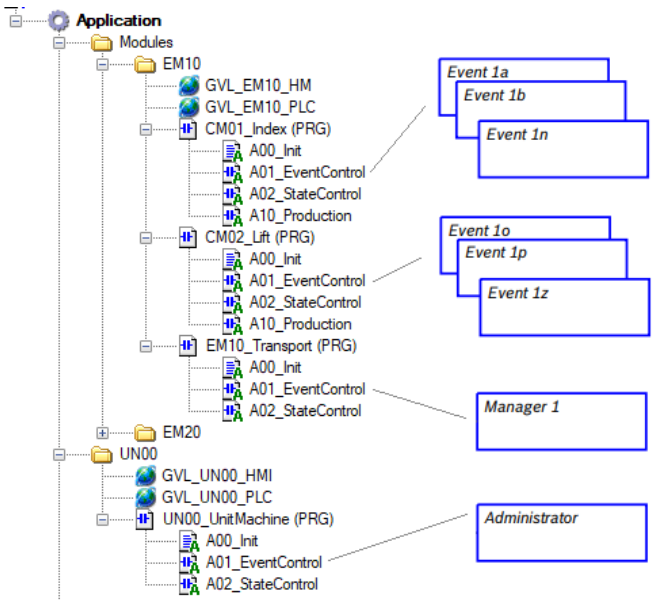


Figure 8: Sample layout of the event handler hierarchy. Note that equipment modules together with their associated control modules are contained in folders. At each level, blocks specific to the event handler are contained in easily recognizable actions.

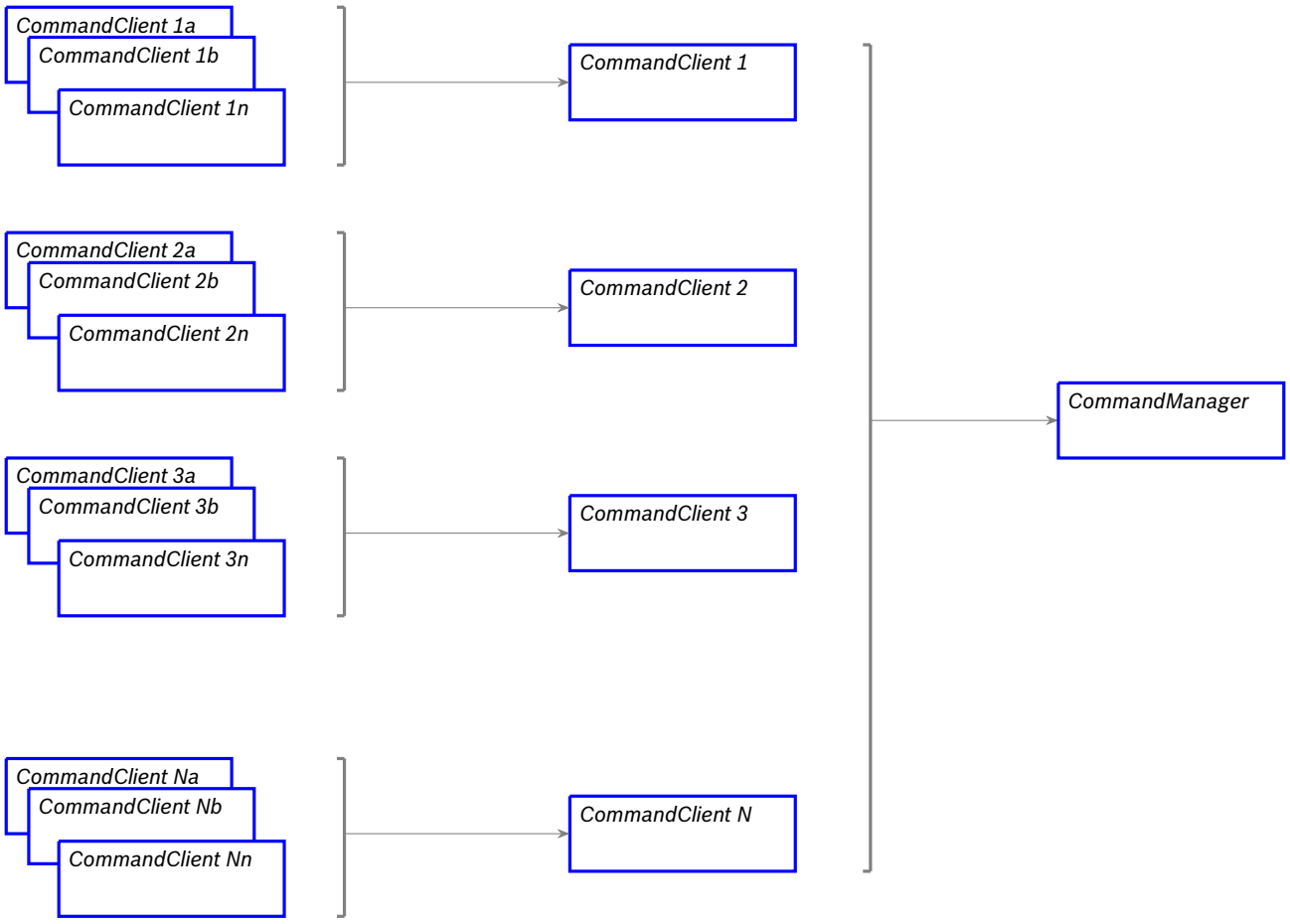


Figure 9: The PackML state command handler also supports a 3-tiered structure. In this case, the CommandClient is used on both ”control module” and ”equipment module” levels. Commands are summed at the unit/machine level using a single CommandManager block. This structure allows modules to directly control the machine state without explicit summation of global variables on a single rung. (Code that explicitly refers to variables from disparate control modules is an anti-modular design pattern: The structure described here allows us to avoid it.)

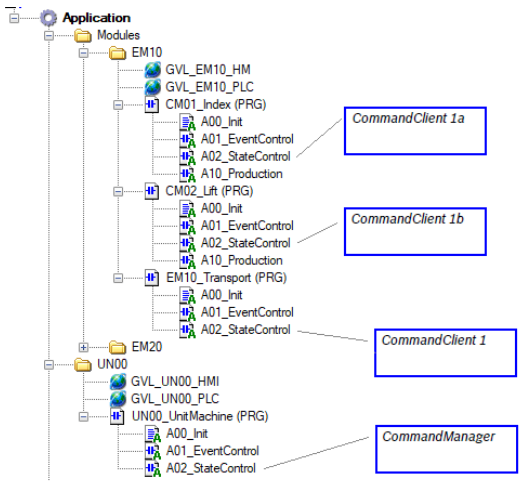


Figure 10: Sample layout of the PackML command hierarchy. As in the case of the event handler, at each level, blocks specific to the command handler are contained in easily recognizable actions.

References

[1] C. Noekleby, *PackML Unit/Machine Implementation Guide, Part 1: PackML Interface State Manager*, 1st ed., OMAC, 2017.

[2] Wikipedia contributors. (2021) PackML — Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/PackML