RFC 82 - Control Unit

Confidential, Draft

62 Pages

Abstract

The purpose of this RFC is to propose generic interface representation of devices. These interfaces can be considered as more concrete extension over the already existing OSGi Device Access specification.

0 Document Information

0.1 Table of Contents

0 Document Information **2**
0.1 Table of Contents 2
0.2 Terminology and Document Conventions 3
0.3 Revision History 3

1 Introduction **5**

2 Application Domain **6**

3 Problem Description **6**
3.1 Description 6
3.2 Use Cases 8

4 Requirements **15**
4.1 Provide Unified Device Representations 15
4.2 Provide Protocol Independence 15
4.3 Monitoring Device Data 15
4.4 Provide Means for Device Management and Control 16
4.5 Optional Device Meta-Typing 16
4.6 Organize Complex Units in Hierarchies 16

5 Technical Solution **17**
5.1 Overview 17
5.1.1 Device Functionality 17
5.1.2 Device Identification 18
5.1.3 Device Monitoring 18
5.1.4 Device Meta Typing 18
5.2 Special Control Unit Actions 18
5.2.1 Attribute Setters 18
5.2.2 Control Unit Finders 18
5.2.3 Control Unit Constructors 18
5.2.4 Control Unit Destructors 19
5.3 System Architecture 19
5.3.1 Provider Layer 19
5.3.2 Admin Layer 20
5.3.3 Application Layer 20
5.4 Class Diagram 21
5.4.1 org.osgi.service.cu 21

5.4.2 org.osgi.service.cu.admin 22
5.4.3 org.osgi.service.cu.admin.spi 22
5.5 Use-Case Realization 23
5.5.1 Monitor and adjust device states 23
5.5.2 The system performs illegal operation on the unit 24
5.5.3 Work with unknown device 25
5.5.4 Collecting device state changes or managing actual device states 26
5.6 API 27
5.6.1 Package org.osgi.service.cu 27
5.6.2 Package org.osgi.service.cu.admin 35
5.6.3 Package org.osgi.service.cu.admin.spi 48

6 Drawbacks and Issues 55

7 Considered Alternatives 57
7.1 Java Beans 57
7.1.1 Advantages 57
7.1.2 Drawbacks 57
7.2 Diagnostics RFC 58
7.3 OSGi UPnP API 58
7.4 Wire Admin API 58
7.5 SyncML DM 58
7.5.1 Objectives 58
7.5.2 Device Representation 59
7.5.3 Mapping between SyncML DM & Control Unit 60
7.5.4 SyncML DM Restrictions 61

8 Security Considerations 61

9 Document Support 62
9.1 References 62
9.2 Author's Address 62
9.3 Acronyms and Abbreviations 62
9.4 End of Document 62

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [1].

`Source code is shown in this typeface.`

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	14 May 2004	Valentin Valchev, ProSyst Software AG, v_valchev@prosyst.bg
1.1	17 May 2004	Updated requirements section
1.2	25 January 2005	Valentin Valchev - Removed Metatyping description, RFC 69 is referred. - Moved Alternative Solution into the Main Body - Respectively updated the use-cases, class diagram ant the API - Resolved issues are removed.
1.3	10 February 2005	Valentin Valchev - Moved ControlUnitAdmin, ControlUnitListener and HierachyListener to a new package named "org.osgi.service.cu.admin". - "org.osgi.service.cu.spi" package is renamed to "org.osgi.service.cu.admin.spi" - ControlConstants interface is renamed to ControlUnitConstants - The StateVariableLister.RECEIVE_VALUES_ON_REGISTER constant is moved to ControlUnitConstants as EVENT_AUTO_RECEIVE - ControlUnit.queryStateVariable() now may throw Exception if the value cannot be retrieved. - Constructor, Desctructor, Finder constants are moved to the ControlUnitAdmin interface. - Properties that previously started with "mbs." now start with "osg."
1.4	1 March 2005	Valentin Valchev All CU-related methods now throws ControlUnitException.
1.5	7 March 2005	Nickola Jetchev - Constants that previously started with "osg." now start with "org.osgi.". - API JavaDoc improved.

Revision	Date	Comments
1.6	14 March 2005	Nickola Jetchev - ControlUnitConstants.STATE_VARIABLES_LIST constant's value changed to "org.osgi.control.var.list". - ControlUnitException.UNDETERMINED_APPLICATION_ERROR constant renamed to UNDETERMINED_ERROR. - ControlUnitException.getApplicationException renamed to getNestedException - Hierarchy events now also fired when a new control unit, which has a parent specified, is registered/unregistered not only when registered control unit changes its position in the control units' hierarchy
1.7	22 March 2005	Nickola Jetchev - Methods in ControlUnitAdmin and ControlUnitFactory that previously were throwing IllegalArgumentException now throw ControlUnitAdminException instead. - Constant NO_SUCH_CONTROL_UNIT_TYPE added to ControlUnitAdminException.
1.8	30 March 2005	Nickola Jetchev - JavaDoc of ControlUnitAdmin's and ControlUnitFactory's findControlUnits, createControlUnit, destroyControlUnit, invokeAction and queryStateVariable methods updated to clarify when ControlUnitAdminException should be thrown.
1.9	06 April 2005	Nickola Jetchev - Part of the Application Domain section moved to the Problem Description section - Technical and implementation details removed from the Use Cases. - Various grammatical and stylistic errors fixed.

1 Introduction

The purpose of this RFC is to propose generic interface representation of diverse type of resources (devices, software or hardware components, etc.) along with a common way to manage and monitor these resources.

These interfaces can be considered as more concrete extension over the already existing OSGi Device Access specification.

The purpose of the proposal is to unify the way in which devices are accessed by OSGi services. The current OSGi Device Specification allows the developers to define their own API for each device type. Nowadays, there can be different representation not only for different kind of devices, but also for functionally identical devices that use different home networking interfaces.

The proposed API aims to reduce the overhead of creating new classes and unify the access to the home or vehicle, etc. independently of the underlying communication protocol.

It is obvious that there is no possibility to define a separate interface for each type of device. Thus, the proposed device abstraction provides methods to query the device attribute and/or execute some device-specific actions. In addition, the API relies on RFC 69 to provide optional user-friendly information about that device and strict Meta-typing of the attributes and action arguments. Applications that have *priory* knowledge about the device - of its actions and attributes - will not use the optional Meta typing.

That general representation allows performing diagnostics and automation actions on newly installed devices.

2 Application Domain

This document proposes a unified representation of diverse type of resources (devices communicating over different protocols, software or hardware components, etc.) along with a common way to manage and monitor these resources.

2.1 Description

The OSGi Framework should be the place where devices and applications can interoperate with each other easily and without much configuration hassle.

The first step in this direction was the device manager, which solves the driver installation problems.

The next step could be a generic device interface.

The problem with the existing system is that all device drivers install different interfaces for the representation of the devices. If you want to write a general-purpose application that uses all the devices (e.g. diagnostics or home automation application) there is a big overhead to support all the different protocol interfaces (UPnP, OneWire, X10, EIB, EHS, MOST, etc.).

Using Java reflection to query the device interface methods is not a good solution too - the information got in this way is not very user-friendly and is not quite suitable for showing in a diagnostics application.

- Reflection does not show any user-friendly information like human-readable description of the device or relations between devices.

- There is no control over the parameter values apart from the basic java type control. A more flexible and strict control over the passed values is desirable (e.g. min, max values, enumerated values, named enumerated values) which cannot be achieved with reflection.

- Localization is out of the question with simple reflection.

- Sometimes drivers require parameters to methods to be not of simple java types, but of specific wrapper classes around them. This makes the task of writing a generic visual diagnostics application even more complex.

- Some drivers (e.g. EIB, EHS drivers) don't present the devices on their network as device objects with attributes and actions. They simply provide an interface for sending commands to the device network. This is a quite different, low-level view of a home network. To extract the current state of a device status variable (or even simply to list the devices) with reflection the technician will have to have detailed knowledge about the device protocol to know which command to send.

OSGi is good and flexible platform that can be extended by different services. Flexibility is one of its best features but too much flexibility is a problem. Applications, that are required to perform diagnostics and control over various home or in-vehicle devices should to know the exact interface that is used for communication. In most cases these interfaces are bound to the underlying communication protocol used by the device itself.

With each new device, such applications become more and more complex, because they use different Java interfaces to those devices and respectively a different way to manage them.

It's obvious that we need a general representation of devices that provides unified device access and control through a single set of Java interfaces, but is flexible enough to cover all vendor features that might be added to devices.

It's obvious that developer will be happy to use only one **unified device representation** API. However, the vendors still must be able to add their own, specific functionality to the devices and shouldn't be bound to a limited set of methods that should be implemented.

In addition the unified device representation should hide the underlying protocol providing **protocol-independent device access**.

We need solution that allows **querying and collecting device diagnosis data**, as well **device management and control**.

When complex control unit cannot have only one single representation, it could be organized as **hierarchy of control units** to simplify the access to its functionally independent subsystems.

The units should be accessible from remote locations. Their actions should be also exported so the user or a backend system could **control and manage the devices remotely**.

The API however should be self-descriptive, so the user could work with the control unit without having any *priori* knowledge of its operation. For a very limited environment, the meta-typing information should not be required at the gateway, but only at the backend, remote side, which performs the management operations.

2.2 Use Cases

Use Case Name	**Manage Control Unit**

Feature					
Actors	*Operator*	*End User*	*Certified System Engineer*	*Other Sub-Systems*	
	X		X		
Description	The intent of the use case is to perform control and monitoring of device connected to the system.				
Flow of events	**Basic flow - Monitor and adjust device states** The diagnostician may adjust device parameters: • For diagnostics & monitoring purpose. • For control and testing purposes 1. This use case starts when the user requests to view or change device parameters. 2. The system displays device parameter values 3. The user invokes fault-management or control actions. 4. The system saves the changes, and updates the device state. 5. The use case ends.				
Must have devices	A device connected to the system.				
May have devices	*N/A*				
Prior conditions	Service is running.				
Post conditions	**Basic flow:** 1. Device parameters are changed. **Alternative flows:** 1. Errors are logged and device parameters are stored.				
Alternative flow 1	**The system performs illegal operation on the control unit** 1. This use case starts when the operator request action execution 2. The system performs invalid operation 3. The system indicates the errors 4. The use case ends.				

Alternative flow 2	**Work with unknown device** 1. This use case starts when the diagnostician requests a record. 2. The system starts the collection of device data 3. The system shall be able to provide the following data on a continuous basis on screen. 4. The use case ends.
Alternative flow 3	**Collecting device state changes or managing actual device states** Specific prior-conditions: • The system provides a device with state that is supposed to measure some expenses (like power consumption) • The collection period is customizable. 1. This use case starts when the operator requests to collect information. 2. The user may specify which states are monitored 3. The system notifies the collector application of monitored state changes 4. The collector application stores the changes into a persistent memory 5. The use case ends.
Notes	
Future considerations	Data collection The system provides a facility to output a file with the collected device data.
Lower level Use Cases	*N/A*
Date	13/05/04
Status	Draft

Use Case Name	**Create Control Unit**

Feature					
Actors	*Operator*	*Admin Tool*	*Certified System Engineer*	*Control Unit Provider*	
	X	*X*		*X*	
Description	The use-case is intended to describe creation of dynamic control units.				

Flow of events	**Basic flow - Monitor and adjust device states** 1. This use case starts when the Administration Tool performs a request to the system for a device to be contacted and managed as control unit. 2. Administration Tool obtains reference to the control unit provider which can create control units upon request. 3. The Operator chooses one of the methods for creating control units supplied by the control unit provider and fills in any the registration parameters required for the chosen method. 4. The Administrator Tool sends request to the system to create a control unit, which in its turn redirects the request to the appropriate control unit provider. 5. New control unit is created, that represents service configuration. 6. The use case ends.
Must have devices	A control unit provider capable of crating control units upon request.
May have devices	*N/A*
Prior conditions	• Service is running. • Management System is connected to the gateway • Operator is already aware of the control unit meta-typing
Post conditions	**Basic flow:** • A new control unit is created **Alternative flows:** • Errors are logged and device parameters are stored.
Alternative flow 1	**The provider doesn't support creation of control units** 1. This use case starts when the operator tries to create a new control unit 2. The provider that is doesn't support creation of control units upon request indicates error by throwing exception 3. The use case ends.
Alternative flow 2	**Illegal arguments are passed to a method for creating control units** 1. This use case starts when the operator tries to create a new control unit 2. Illegal arguments to methods for creating control units are signaled by throwing exception. 3. The use case ends

Notes	
Future considerations	
Lower level Use Cases	*N/A*
Date	14/05/04
Status	Draft

Use Case Name	**Attributes Event Notification**

Feature					
Actors	*Operator*	*Admin Tool*	*Certified System Engineer*	*Control Unit Provider*	
	X	*X*		*X*	
Description	The use-case is intended to state notifications.				
Flow of events	**Basic flow - Monitoring** 1. This use case starts when the Operator sends request for monitoring state changes, using the Administration Tool. 2. An attribute is changed 3. The control unit provider notifies the system. 4. The system delivers the event to the Administration Tool. 5. The Administration Tool updates the current values. 6. The use case ends.				
Prior conditions	• Service is running. • Management System is connected to the gateway • Operator is already aware of the control unit meta-typing • There is a provided control unit whose states are changing during the use case execution				
Post conditions	**Basic flow:** • The Operator gets notified about state changes				
Notes					
Future considerations	*N/A*				
Lower level Use Cases	*N/A*				
Date	14/05/04				
Status	Draft				

Use Case Name	Searching through Control Unit

Feature					
Actors	*Operator*	*Admin Tool*	*Certified System Engineer*	*Control Unit Provider*	
	X	X		X	
Description	The use-case is intended to describe factory specific finder methods that implement custom and efficient search of control units based on certain configurable filters				
Flow of events	**Basic flow - Monitor and adjust device states** 1. This use case starts when the Administration Tool requests the system to search for control units with a given criteria. 2. The Administration Tool constructs the search parameters using the 'finder' actions regarding its input arguments definition. With these search parameters it requests the system to find a control unit. 3. The system redirects this request to the corresponding control unit provider. 4. The control unit provider performs an effective search - for example if the control units are just some kind of data - log records or captures of cameras stored on a hard drive, and returns the units that match the specified criteria to the system. 5. The system returns the control units to the user, thus ending the use case.				
Must have devices	A ControlUnitFactory that supports finder methods.				
May have devices	*N/A*				
Prior conditions	• Service is running. • Management System is connected to the gateway • Operator is already aware of the control unit meta-typing				
Post conditions	**Basic flow:** • Search result is expected **Alternative flows:** • Errors are logged				
Alternative flow 1	**The factory doesn't support searching of control units** 1. This use case starts when the operator tries to search for a control unit. 2. The factory that doesn't supports 'finder' method indicates error by				

	throwing exception 3. The use case ends.
Alternative flow 2	**Illegal arguments are passed to finder method** 1. This use case starts when the operator tries to search for a control unit 2. Illegal arguments to finder methods are signaled by throwing exception. 3. The use case ends
Notes	
Future considerations	
Lower level Use Cases	*N/A*
Date	14/05/04
Status	Draft

Use Case Name	**Invoking Actions without explicit Control Unit Object creation**

Feature					
Actors	*Operator*	*Admin Tool*	*Certified System Engineer*	*Control Unit Provider*	
	X	*X*		*X*	
Description	The use-case is intended to describe the invoking of actions through the ControlUnitFactory.				
Flow of events	**Basic flow - Monitor and adjust device states** 1. This use case starts when the Operator requests the system to modify configuration control unit. 2. The Administration Tool locates the factory that provides control unit that wraps configuration services. 3. Then, it redirects the invocation request to the factory. 4. The control unit factory modifies the configuration without explicit creation of the control unit object that is only used for unified representation. 5. The use case ends				
Must have devices	A ControlUnitFactory for the type				
May have devices	*N/A*				
Prior conditions	• Service is running.				

	• Management System is connected to the gateway • Operator is already aware of the control unit meta-typing
Post conditions	**Basic flow:** • Configuration is modified **Alternative flows:** • Errors are logged
Alternative flow 1	**Illegal arguments are passed to action** 1. The use case starts from point 3 in default flow. 2. Illegal arguments to actions are signaled by throwing exception.. 3. The use case ends.
Notes	
Future considerations	
Lower level Use Cases	*N/A*
Date	14/05/04
Status	Draft

3 Requirements

3.1 Provide Unified Device Representations

The Control Unit API provides general, unified representation of devices but doesn't limit the vendor to extend the functionality of devices, adding new attributes and actions. That goal is achieved without modification of the proposed API.

3.2 Provide Protocol Independence

If there are OSGi services, that are providing visual interface for those devices, they must not be aware of the underlying protocol but for the core functionality. However, there could be additional services for network management that must be able to access the functions, allowing to control certain aspects of the network configuration of the device – including its address or a network group to which it belongs.

Example: The developer shouldn't see any distinctions between two functionally equal devices (like two ovens of the same model) that are connected to different networks - one using EHS and the other EIB.

3.3 Monitoring Device Data

The user must be able to query the value of control unit attribute at any time. He must be also allowed to register a listener that will sniff for change of concrete attribute or group of attributes, even if they are from different devices.

It's up to the implementation to perform either a network operation or use a cached value, when attribute is queried.

The listeners are registered using the White Board model that is recommended by OSGi [6]. Those listeners would enable development of statistical applications that monitor device attribute changes in order to measure performance, consumption, etc. Ability to collect data, although that is not a goal of that specification is an important feature.

If an application wants to receive notifications by the control unit admin service it registers an appropriate service in the OSGi registry. There are three types of events required by the applications:

- To receives notifications about adding and removing of control unit instances inside the framework.
- To receive notifications about changes in the control units hierarchy.
- To receive notifications about changing the value of a control unit state variable / attribute. When monitoring the values of the control unit, the developer must be allowed to choose from the following monitoring options:
    - All attributes of specific device
    - Specific attribute of a device
    - Attribute changes from group of devices

- Receive changes from all devices

If notifications are to be received synchronously, the application should specify this explicitly. The default behavior is to deliver them asynchronously.

3.4 Provide Means for Device Management and Control

Device control is achieved through support of actions. The actions are supposed to have input and output parameters. Each action has an ID that uniquely identifies it in the scope of the corresponding control unit type.

The available actions are not subject of this specification but they might include:

- Actions for start self-diagnostic tests
- Actions to control the behavior of the application

3.5 Optional Device Meta-Typing

The general representation of device cannot be used if there is no Meta-typing that specifies the types and allowed values of attributes and action parameters.

Usually, the device itself is running where the OSGi framework is deployed, but the user accesses the gateway through the operator's site. In this situation the Meta-typing is not needed at the gateway side and is required only be the operator.

Considering the fact, that the meta-typing information usually engages a lot of resources, it is strongly recommended to give the ability to ship those user-friendly descriptions separately from the control unit.

3.6 Organize Complex Units in Hierarchies

Control units may be arranged hierarchically - every control unit instance may have one or more child control units and one or more parent control units. Such layout may be convenient for logical grouping of units, and is especially useful for representing more complex resources - devices, hardware and software systems, which may be decomposed to a hierarchy of sub-components, achieving arbitrary level of granularity.

It is the responsibility of a child control unit to "attach" itself under the appropriate parent(s) in the tree. It means that the parent control units do not need to know their sub-control units, but the child control unit has to know its parent(s).

Both parent and child control units of a given control unit may be of different types.

4 Technical Solution

4.1 Overview

4.1.1 Device Functionality

The solution is based on the assumption that devices can be generally represented by a **set of attributes and actions**.

- A control unit attribute has an identifier and a value. The identifier is a `String` that uniquely identifies the variable among the other variables of the same control unit. The value is an object of one of the allowed types. This value may change dynamically during the lifecycle of the control unit but its type has to be always the same.
- An action is characterized by an identifier, a set of input arguments (possibly zero) and a set of output arguments (possibly zero). The action identifier is a `String` that uniquely identifies the action among the other actions supported by the same control unit. The input and output arguments are ordered as a sequence of objects with the same or different types. Action arguments can be considered the same as attribute definitions.

It is supposed that there are some applications that are already aware of the kind of attributes and actions available from the device. Other applications, that are not aware of the provided device functionality, use special services, which provide meta-typing information about all devices of the same kind.

In addition, this technical solution tries to resolve some drawbacks related to the implementation, which are:

- Every control unit has to implement an event delivery mechanism
- Every control unit has to track the event listeners
- A user application has to track all control units that it is interested in

To solve those problems the proposed model introduces an interface named `ControlUnitAdmin`. That adapter is used by all applications and services to query the available control units and to retrieve their instances.

This simplifies the control unit development, because implementers only need to send events to the `ControlUnitAdmin`. The `ControlUnitAdmin` furthermore is responsible for tracking the registered listeners, for filtering the listeners which must receive the changes, and for delivering the change in either asynchronous or synchronous manner.

Additionally, a `ControlUnitFactory` interface is introduced. Control unit factories are mend to provide dynamic creation and/or removal of control units.

Also, these factories are capable of directly invoking a control unit action without requiring control unit object to be created.

4.1.2 Device Identification

Each instance of a control unit has a **type** and an **ID** associated with it. The type of the control unit identifies its interface. There may be many control unit instances with the same type, which means that they have the same set of attributes and action identifiers, but may have different states. The ID of the control unit instance has to be unique in the scope of the type.

4.1.3 Device Monitoring

The set of identifiers and corresponding types of the state variables and actions supported by a control unit instance determines the **interface** of the control unit. The current values of all state variables supported by a control unit determine the current **state** of the control unit instance.

The control unit abstraction provides a unified, well-known way to monitor the state and to execute an operation on a control unit. Thus, although there is a great flexibility in what operations can be invoked on control units and the state variables they provide, applications can manage control units in a uniform way, without the need to bind with specific knowledge about the particular underlying resources.

The solution presented in this RFC uses the recommended Whiteboard model [6] to notify the interested parties that attributes of certain control units have been changed.

4.1.4 Device Meta Typing

- The control unit meta-typing is based on RFC 69 - Metatyping for Control Units and Diagnostics. It introduces a new extension to the OSGi Meta Typing Service and provides meta information that is usually required to fully describe the attributes and the parameters of control unit actions – like minimum, maximum, step values for numbers and hidden attribute for passwords or other sensitive data

4.2 Special Control Unit Actions

There are several actions with predefined semantic, which may optionally be supported by control unit providers.

4.2.1 Attribute Setters

The control unit provider of a particular CU type may support several actions whose IDs are in the form “`$set.<attribute_id>`”, where `<attribute_id>` must be an identifier of an attribute, supported by the control units of the same type. Such actions must have one input argument of the same type as the corresponding attribute and must not have output arguments. Invoking such actions must change the value of the control unit attribute with the supplied argument.

4.2.2 Control Unit Finders

The control unit provider of a particular control unit type may optionally support one or more methods for filtering control units. These methods are called *finders* and are defined as an action whose ID starts with “`$find.`”. Every finder method may have different number and/or type of arguments.

Finders are executed through `findControlUnits(…)` method of the `ControlUnitFactory` interface. Therefore, only control unit providers that register `ControlUnitFactory` may support finder methods.

4.2.3 Control Unit Constructors

The control unit provider of a particular control unit type may optionally support one or more methods for explicit creation of control units. These methods are called *constructors* and are defined as an action whose ID starts with “`$create.`”. Every constructor method may have different number and/or type of arguments.

Constructors are executed through `createControlUnit (…)` method of the `ControlUnitFactory` interface. Therefore, only control unit providers that register `ControlUnitFactory` may support constructor actions.

4.2.4 Control Unit Destructors
The control unit provider of a particular CU type may optionally support explicit removal of control units. This is indicated by defining an action whose ID starts with "`$destroy`" without arguments, called destructor.

Destructor is executed through `destroyControlUnit(…)` method of the `ControlUnitFactory` interface. Therefore, only control unit providers that register `ControlUnitFactory` may support destructors.

4.3 System Architecture

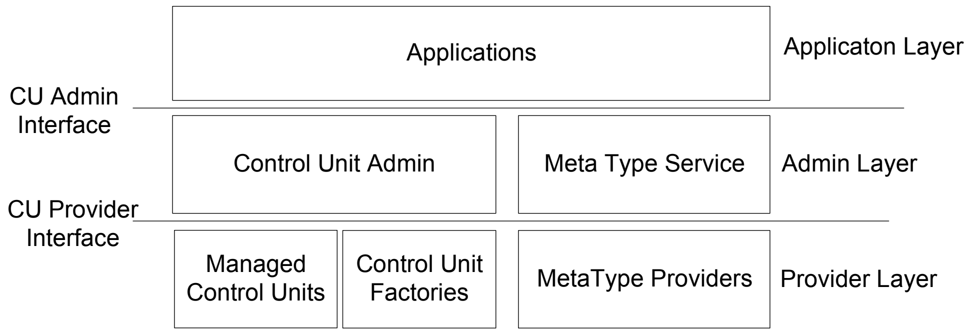


Figure 1: Control Unit Architecture

The control unit system is divided into three layers:

- Provider Layer
- Admin Layer
- Application Layer

4.3.1 Provider Layer
This layer provides control unit compliant implementation of the resources so they can be managed by control unit compliant applications. A control unit compliant resource can be a device, an application or an application module, an OSGi bundle, an OSGi service, a user, and so forth, whose state is represented as a set of attributes and its functionality is represented as a set of actions. This representation of resources is handled by the control unit providers. Control unit provider are the only ones that directly deal with the corresponding resources and abstract them in a uniform way, suitable for the control unit system.

Control units providers provide two types of resources - control units' implementation and control units Meta type.

4.3.1.1 Providing Control Unit Implementation
The implementation of a control unit is responsible for the actual representation of the underlying resource as a set of state variables and for executing the actions using the resource specific mechanisms.

There are three ways for providing implementation of control units.

- The first method is to register `ManagedControlUnit` service in the OSGi framework. This approach is suitable for providing control unit implementation of a single resource, because the `ManagedControlUnit` represents a single control unit instance.

- The second one is to register a `ControlUnitFactory` service instead. The `ControlUnitFactory` provides multiple control units instances of exactly one type and is suitable for representing a variable number of uniform resources. This second approach of providing control units has the additional benefit that large number of control units can be provided, without overloading the OSGi framework registry.

Managing units through factory methods (instead of obtaining units from factory and working with them directly) allows further optimizations. Factories can also provide methods for creating and destroying control units. This can be useful when some prior information is needed to start managing a resource. For example, a user wants to manage a device. There is a factory, providing control units, which manage this device, but the user is the only one having the information how the device can be contacted. When he wants to manage a device the user asks the factory to create the control unit for it supplying the contact information and thereafter the device can be managed through the newly created control unit. When the device is disconnected or no further management is needed the control unit can be destroyed. Some custom searching methods for retrieving control units is another useful option a factory may provide. Such methods would give an optimal and efficient way of finding the desired control units.

- The third one is to create and register a simple `ControlUnit`. In this case the developer must implement by himself the event delivery mechanism because the control unit admin cannot handle direct ControlUnit implementations.

Some custom searching methods for retrieving control units is another useful option a factory may provide. Such methods would give an optimal and efficient way of finding the desired control units.

4.3.1.2 Providing Control Unit Metatyping

See RFC 69 - Metatyping for Control Units and Diagnostics.

4.3.2 Admin Layer

The Admin Layer directly manipulates control unit providers and makes control units provided by them available to the applications. The applications do not interact with the providers directly. They are using the admin layer instead.

The Admin Layer consists of two components:

- `ControlUnitAdmin` – responsible for dealing with providers of control unit implementations – `ManagedControlUnit` and `ControlUnitsFactory` instances.
- Meta-type service - responsible for dealing with control unit meta-typing providers. Meta-type service is described in detail in RFC62 – Metatype Update.

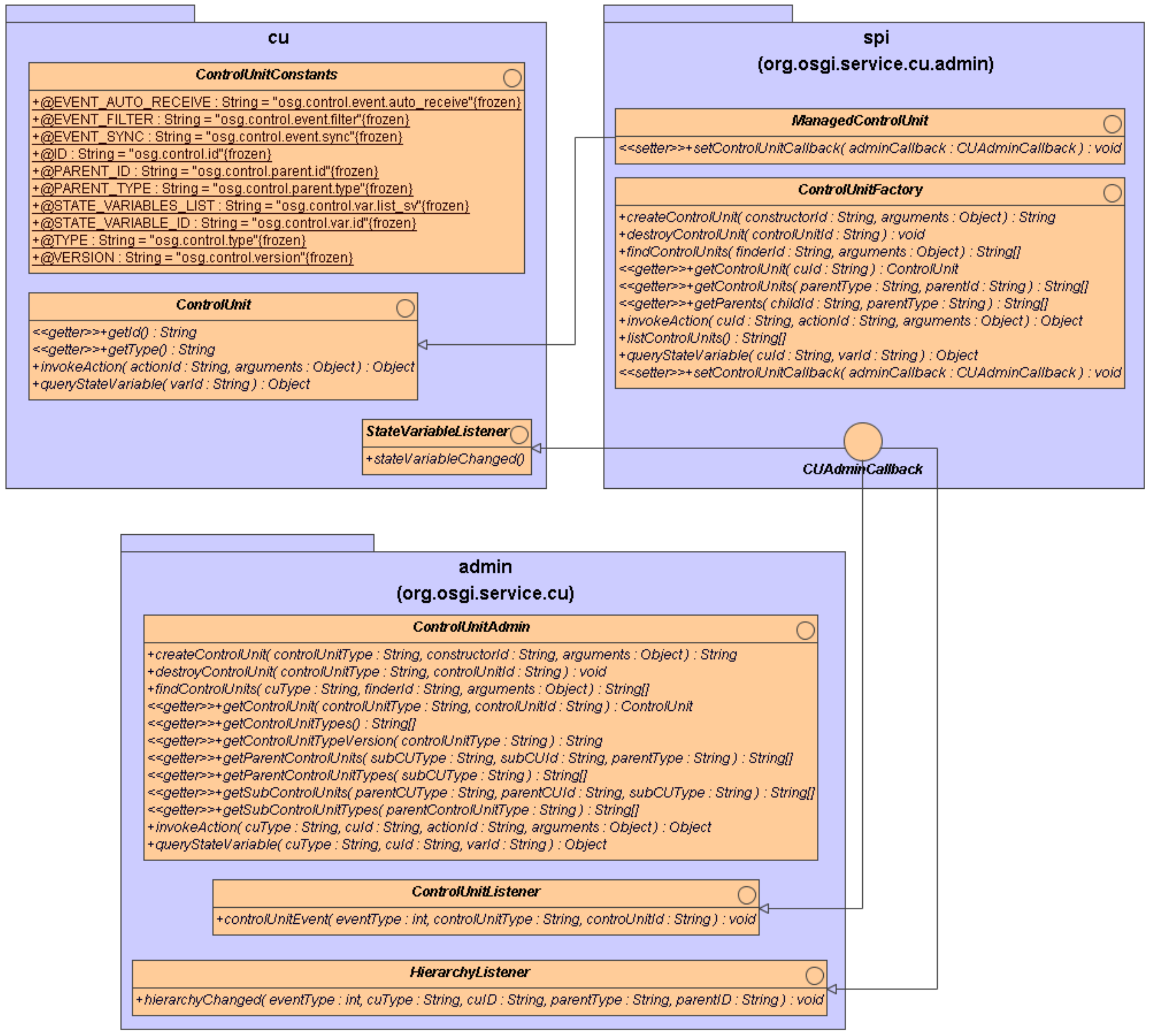
Control unit admin interacts with the control unit providers over a well-defined interface called control unit provider interface.

4.3.3 Application Layer

Application Layer consists of custom applications dealing with control units. These may be generic management applications, which provide control and monitoring of all resources available as control units in the OSGi framework, or may be specific applications, which interact with particular type(s) of control units.

Applications interact with the control unit admin over a well-defined interface called `ControlUnitAdmin`, which provides them methods for finding the control units, monitoring their state, invoke their actions, etc.

4.4 Class Diagram


cu
ControlUnitConstants
+@EVENT_AUTO_RECEIVE : String = "osg.control.event.auto_receive"{frozen}
+@EVENT_FILTER : String = "osg.control.event.filter"{frozen}
+@EVENT_SYNC : String = "osg.control.event.sync"{frozen}
+@ID : String = "osg.control.id"{frozen}
+@PARENT_ID : String = "osg.control.parent.id"{frozen}
+@PARENT_TYPE : String = "osg.control.parent.type"{frozen}
+@STATE_VARIABLES_LIST : String = "osg.control.var.list_sv"{frozen}
+@STATE_VARIABLE_ID : String = "osg.control.var.id"{frozen}
+@TYPE : String = "osg.control.type"{frozen}
+@VERSION : String = "osg.control.version"{frozen}
ControlUnit
<<getter>>+getId() : String
<<getter>>+getType() : String
+invokeAction(actionId : String, arguments : Object) : Object
+queryStateVariable(varId : String) : Object
StateVariableListener
+stateVariableChanged()
spi
(org.osgi.service.cu.admin)
ManagedControlUnit
<<setter>>+setControlUnitCallback(adminCallback : CUAdminCallback) : void
ControlUnitFactory
+createControlUnit(constructorId : String, arguments : Object) : String
+destroyControlUnit(controlUnitId : String) : void
+findControlUnits(finderId : String, arguments : Object) : String[]
<<getter>>+getControlUnit(cuId : String) : ControlUnit
<<getter>>+getControlUnits(parentType : String, parentId : String) : String[]
<<getter>>+getParents(childId : String, parentType : String) : String[]
+invokeAction(cuId : String, actionId : String, arguments : Object) : Object
+listControlUnits() : String[]
+queryStateVariable(cuId : String, varId : String) : Object
<<setter>>+setControlUnitCallback(adminCallback : CUAdminCallback) : void
CUAdminCallback
admin
(org.osgi.service.cu)
ControlUnitAdmin
+createControlUnit(controlUnitType : String, constructorId : String, arguments : Object) : String
+destroyControlUnit(controlUnitType : String, controlUnitId : String) : void
+findControlUnits(cuType : String, finderId : String, arguments : Object) : String[]
<<getter>>+getControlUnit(controlUnitType : String, controlUnitId : String) : ControlUnit
<<getter>>+getControlUnitTypes() : String[]
<<getter>>+getControlUnitTypeVersion(controlUnitType : String) : String
<<getter>>+getParentControlUnits(subCUType : String, subCUId : String, parentType : String) : String[]
<<getter>>+getParentControlUnitTypes(subCUType : String) : String[]
<<getter>>+getSubControlUnits(parentCUType : String, parentCUId : String, subCUType : String) : String[]
<<getter>>+getSubControlUnitTypes(parentControlUnitType : String) : String[]
+invokeAction(cuType : String, cuId : String, actionId : String, arguments : Object) : Object
+queryStateVariable(cuType : String, cuId : String, varId : String) : Object
ControlUnitListener
+controlUnitEvent(eventType : int, controlUnitType : String, controlUnitId : String) : void
HierarchyListener
+hierarchyChanged(eventType : int, cuType : String, cuID : String, parentType : String, parentID : String) : void

4.4.1 org.osgi.service.cu

This package defines the basic layer API of the of the control unit architecture. It is intended for use by the applications willing to control and monitor control units.

Control units abstraction provides common representation of a different kind resources - devices or device components, applications or an application components, OSGi bundles and services, users, and so forth. This common abstraction enables applications to monitor and control these resources in a uniform way even without having a prior knowledge of the exact resource characteristics.

The developers interested in receiving any state variable change event have to register a StateVariableListener service using the OSGi White Board approach.

With small exceptions, most of the ControlUnits are OSGi Services and can be accessed directly from the OSGi Service Registry.

4.4.2 org.osgi.service.cu.admin

The Admin package provides an additional, utility layer that hides the service provider layer and provides facility for simplified event delivering.

Because of the control unit factories and hierarchies, the package also defines two listeners – the ControlUnitListener and the HierarchyListener. The first one is required because the control units created from factories are not necessarily registered as OSGi services and this listener is the only way to track if a new unit has appeared or disappeared. The second one is required to track the movement of a single unit in the control unit hierarchy.

The Admin package also aids the users of control units by providing unified control unit repository. It operates like a service tracker for ControlUnits and provides the user access to every control unit in the framework. The actual benefit of this is that the developer doesn't need to write complex LDAP filters and to track multiple ControlUnit services, but instead is only interested in the ControlUnitAdmin service.

However, developers may still access the ControlUnits directly using the OSGi Service Registry.

4.4.3 org.osgi.service.cu.admin.spi

This package provides a **S**ervice **P**rovider **I**nterface – a mechanism for implementing manageable resources through control unit abstraction and making them available to the applications via the `ControlUnitAdmin` service. This package defines the interface between the `ControlUnitAdmin` service and resources and is not intended for direct usage by the applications, which use the control units.

A control unit resource can be a device or a component of a device, an application or an application module, OSGi bundle, OSGi service, a user, and so forth. To make it manageable via the control unit abstraction the resource must be wrapped by an instance of the `ControlUnit` interface and exported to the `ControlUnitAdmin` by `ControlUnit`, `ManagedControlUnit` or `ControlUnitFactory` service. More complex resources may be represented as a set of control units grouped in a tree like structured exported by several `ControlUnit`, `ManagedControlUnit` and/or `ControlUnitFactory` services.

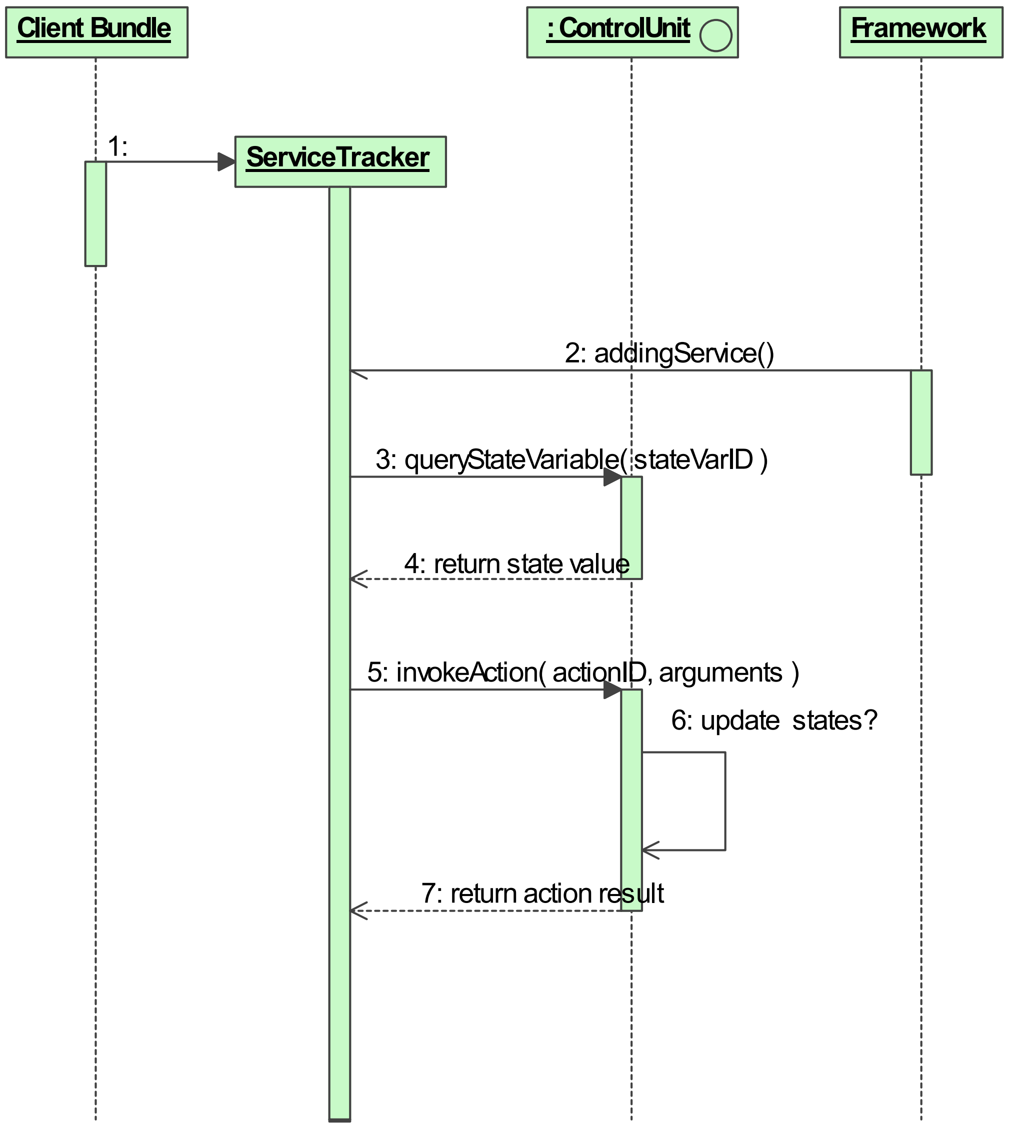
The ControlUnitFactory interface allows the user to create and destroy dynamic control units. Here is a typical situation:

The user has an oven that has an advanced receipt support. In order to add a new receipt to the oven it instructs the oven receipt factory to create a new control unit representing the new receipt. Unfortunately too many objects mean more memory, which is a problem for embedded devices. Fortunately, the factory may not create a new object but instead will only store its data in the right place. Then, when the user queries the receipt parameters, he may use the factory methods for querying the state variables. In this case there will not be any ControlUnit instance created. However, if the user requests the ControlUnit itself, and then queries the state variables from the object, a new instance is created.

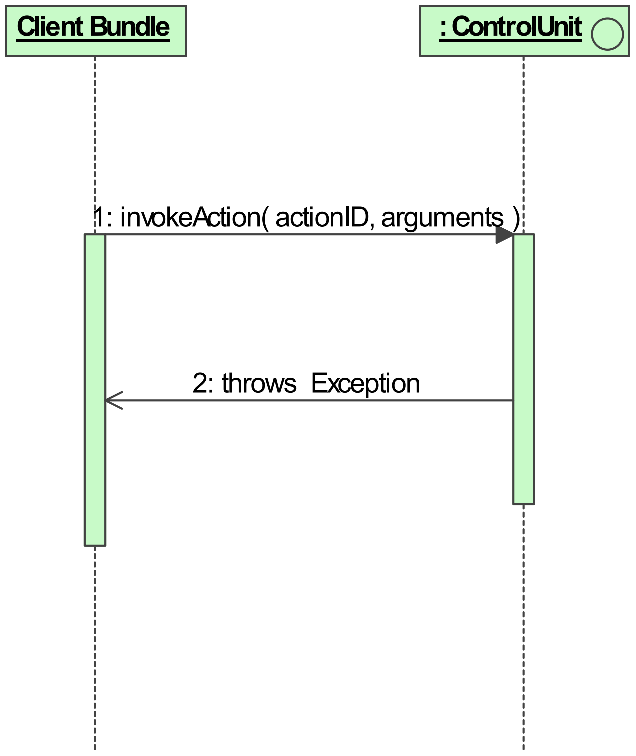
All interfaces are thoroughly described in section 4.6 - API.

4.5 Use-Case Realization

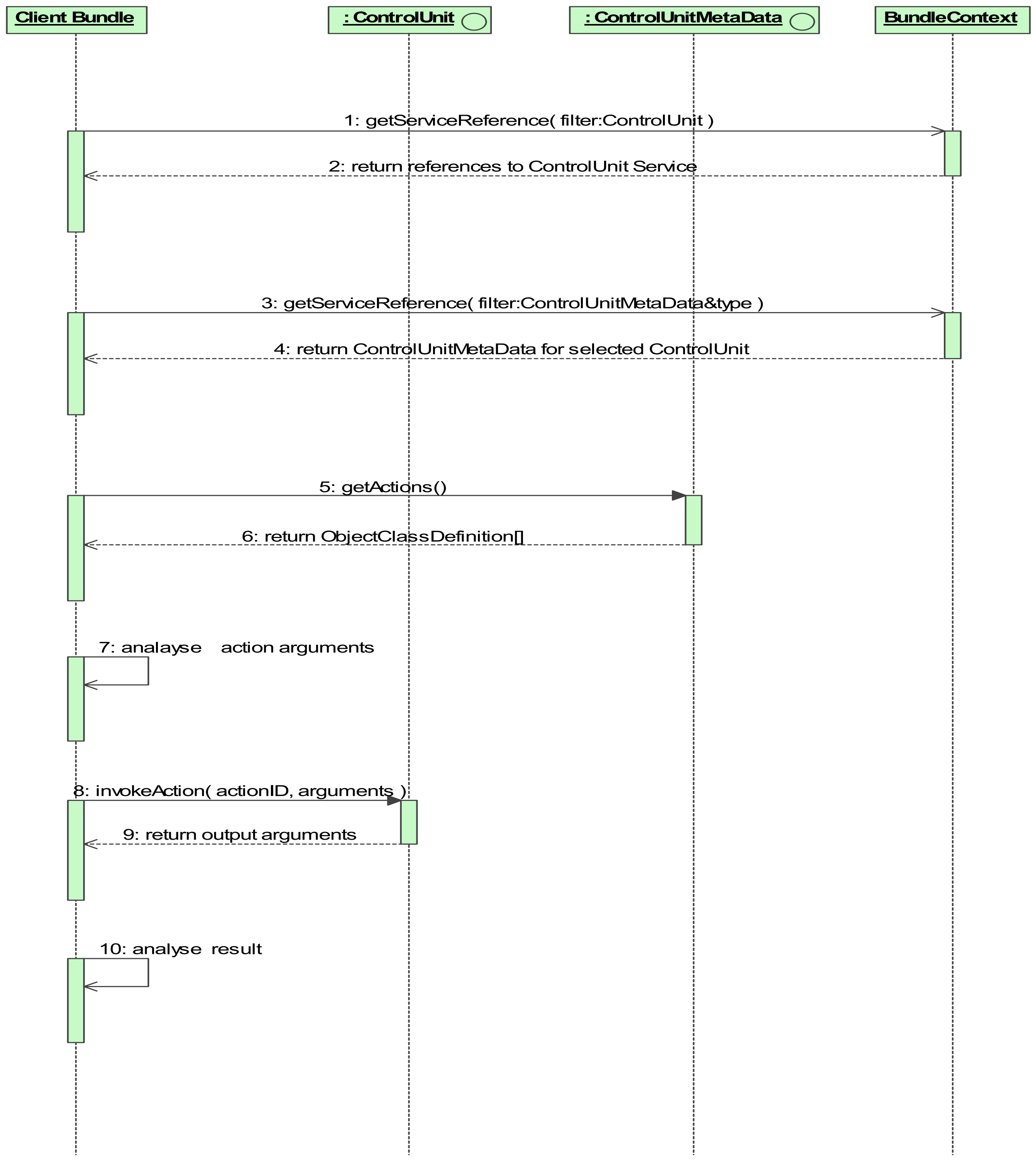
4.5.1 Monitor and adjust device states



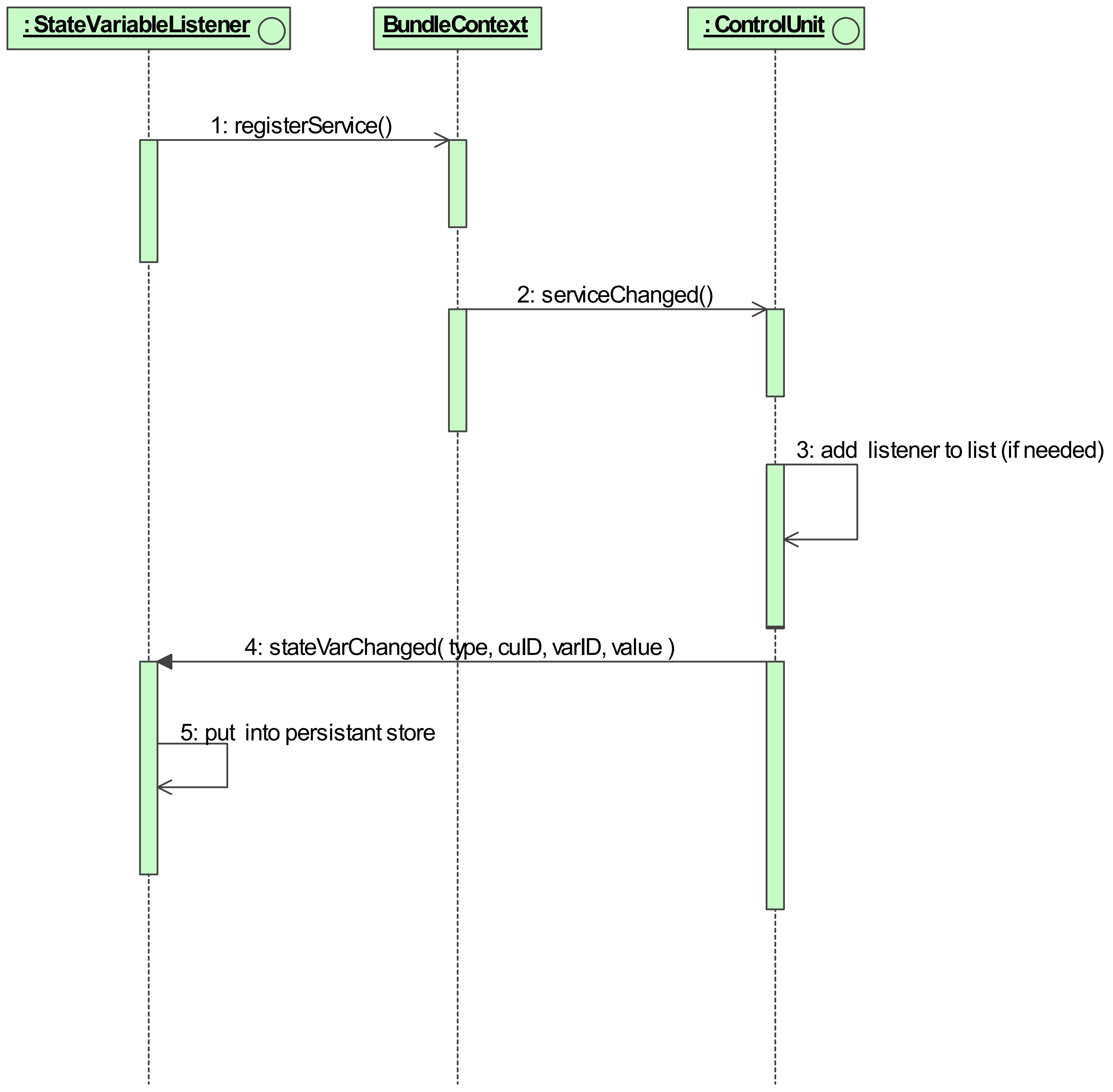
4.5.2 The system performs illegal operation on the control unit



4.5.3 Work with unknown device


Client Bundle
: ControlUnit
: ControlUnitMetaData
BundleContext
1: getServiceReference(filter:ControlUnit)
2: return references to ControlUnit Service
3: getServiceReference(filter:ControlUnitMetaData&type)
4: return ControlUnitMetaData for selected ControlUnit
5: getActions()
6: return ObjectClassDefinition[]
7: analayse action arguments
8: invokeAction(actionID, arguments)
9: return output arguments
10: analyse result

4.5.4 Collecting device state changes or managing actual device states


: StateVariableListener
BundleContext
: ControlUnit
1: registerService()
2: serviceChanged()
3: add listener to list (if needed)
4: stateVarChanged(type, cuID, varID, value)
5: put into persistant store

4.6 API

4.6.1 Package org.osgi.service.cu

4.6.1.1 Interface ControlUnitConstants

public interface **ControlUnitConstants**
This interface defines constants used as service registration properties and metadata attribute keys in the scope of Control Unit API.

Field Summary

static java.lang.String	**EVENT_AUTO_RECEIVE** StateVariableListeners, which wishing to receive upon registration the values of the state variables for which they are listening, should have this property set in their service registration properties (the value of the property doesn't matter).
static java.lang.String	**EVENT_FILTER** This service registration property may be used by the ControlUnitAdminListeners, StateVariableListeners and HierarchyListeners to filter the events they are interested in.
static java.lang.String	**EVENT_SYNC** This service registration property may be used by the ControlUnitAdminListeners, StateVariableListeners and HierarchyListeners to specify that events will be sent to them synchronously.
static java.lang.String	**ID** The ID property uniquely identifies a control unit instance in the scope of its TYPE.
static java.lang.String	**PARENT_ID** Property specifying the ID of a parent control unit.
static java.lang.String	**PARENT_TYPE** Property specifying the type of the parent control unit.
static java.lang.String	**STATE_VARIABLE_ID** This property may be used in the event filters of StateVariableListeners to specify the state variables which the listeners are interested in.
static java.lang.String	**STATE_VARIABLES_LIST** Control units must have state variable with ID equal to this constant, which value should be a string array containing the IDs of all other control unit's state variables.
static java.lang.String	**TYPE** The TYPE of a control unit specifies its type.
static java.lang.String	**VERSION** The VERSION property, when set, specifies the version of the meta-typing information used for unit description.

Field Detail

ID

public static final java.lang.String **ID**

The ID property uniquely identifies a control unit instance in the scope of its TYPE. Therefore, the pair (TYPE, ID) uniquely identifies a single control unit within the scope of the current OSGi framework.
The ID is a service registration property.
The value of this constant is "org.osgi.control.id".

TYPE

public static final java.lang.String **TYPE**

The TYPE of a control unit specifies its type. There may be many control units of the same type. All control units of one type have a same set of state variable ids and a same set of supported actions specified by the ObjectClassDefinition with the same ID. The type is a service registration property.
The value of this constant is "org.osgi.control.type".
See Also:
org.osgi.service.metatype.ObjectClassDefintion#getID()

VERSION

public static final java.lang.String **VERSION**

The VERSION property, when set, specifies the version of the meta-typing information used for unit description.
The version is a service registration property.
The value of this constant is "org.osgi.control.version".

PARENT_ID

public static final java.lang.String **PARENT_ID**

Property specifying the ID of a parent control unit.
The parent ID is a service registration property.
The value of this constant is "org.osgi.control.parent.id".

PARENT_TYPE

public static final java.lang.String **PARENT_TYPE**

Property specifying the type of the parent control unit.
The parent type is a service registration property.
The value of this constant is "org.osgi.control.parent.type".

EVENT_AUTO_RECEIVE

public static final java.lang.String **EVENT_AUTO_RECEIVE**

StateVariableListeners, which wishing to receive upon registration the values of the state variables for which they are listening, should have this property set in their service registration properties (the value of the property doesn't matter). If a listener has registered with such property and a new control unit with variables matching the filter of the listener is registered it will receive the values of these variables. The value of this constant is "org.osgi.control.event.auto_receive".

EVENT_FILTER

public static final java.lang.String **EVENT_FILTER**

This service registration property may be used by the ControlUnitAdminListeners, StateVariableListeners and HierarchyListeners to specify the events they are interested in.
The value of this constant is "org.osgi.control.event.filter".

EVENT_SYNC

public static final java.lang.String **EVENT_SYNC**

This service registration property may be used by the ControlUnitAdminListeners, StateVariableListeners and HierarchyListeners to specify that events will be sent to them synchronously.
The value of the property doesn't matter - if it's present events will be delivered synchronously to the corresponding listener. Otherwise events are delivered asynchronously.
The value of this constant is "org.osgi.control.event.sync"

STATE_VARIABLE_ID

public static final java.lang.String **STATE_VARIABLE_ID**

This property may be used in the event filters of StateVariableListeners to specify the state variables, which the listeners are interested in.
The value of this constant is "org.osgi.control.var.id".

STATE_VARIABLES_LIST

public static final java.lang.String **STATE_VARIABLES_LIST**

Control units must have state variable with ID equal to this constant, which value should be a string array containing the IDs of all other control unit's state variables.
The state variable with this ID is used by the ControlUnitAdmin when it has to send initial state variables values to StateVariableListeners which have registered with the EVENT_AUTO_RECEIVE service registration property.
The value of this constant is "org.osgi.control.var.list".

4.6.1.2 Interface ControlUnit

All Known Subinterfaces:

ManagedControlUnit

public interface **ControlUnit**

Control unit is an object, which provides formal representation of a certain resource (device, software or hardware components, etc.) so it can be managed in a uniform way by different applications.
The public interface of the control unit is represented by a set of valued attributes called state variables and set of operations called actions.

A control unit instance is characterized by its type and ID. The type of a control unit defines its allowed set of state variables and actions. The ID of the control unit instance identifies it uniquely in the scope of its type.
A control unit instance can be exported (made available) to the management applications either by registering ManagedControlUnit, which represents single control unit instance or by registering a ControlUnitFactory service, which maintains a set of control unit instances of the same type.
Control units may be arranged hierarchically - every control unit instance may have one or more sub control units and one or more parent control units. The implementers of the control units must avoid cycles in the control unit hierarchy. Organizing control units may be convenient for logical grouping of control units, but is especially useful for representing more complex resources - devices, hardware and software systems, which may be decomposed to a hierarchy of sub-components, achieving arbitrary level of granularity.

Method Summary	
java.lang.String	**getId**()

	Returns the ID of the control unit, which uniquely identifies it in the scope of its parent.
java.lang.String	**getType**() Returns the type of the control unit.
java.lang.Object	**invokeAction**(java.lang.String actionID, java.lang.Object arguments) Executes the specified action over this control unit.
java.lang.Object	**queryStateVariable**(java.lang.String stateVariableID) Returns the value of a specified state variable.

Method Detail

getId

public java.lang.String **getId**()

Returns the ID of the control unit, which uniquely identifies it in the scope of its parent.
Returns:
The ID of the control unit

---

getType

public java.lang.String **getType**()

Returns the type of the control unit. This type is used to retrieve metatype information from the metatype service.
Returns:
The type of the control unit

---

queryStateVariable

public java.lang.Object **queryStateVariable**(java.lang.String stateVariableID)
throws ControlUnitException

Returns the value of a specified state variable. State variables supported by a control unit and their types are defined by the metadata of the control unit.
Parameters:
stateVariableID - The ID of the variable
Returns:
The value of the variable
Throws:
ControlUnitException - if the state variable's value cannot be retrieved for some reason. ControlUnitException.getErrorCode() and ControlUnitException.getNestedException() methods can be used to determine the actual cause.
java.lang.NullPointerException - if the stateVariableID is null.

---

invokeAction

public java.lang.Object **invokeAction**(java.lang.String actionID,
java.lang.Object arguments)
throws ControlUnitException

Executes the specified action over this control unit. Actions supported by a control unit and the number and types of the input and output arguments of each action are defined by the metadata of the control unit.

Parameters:
actionID - the ID of the action
arguments - the input argument(s). If the argument is only one this is the argument itself. If the arguments are more then one, the value must be an Object array and arguments are retrieved from that array.
Returns:
The output argument(s) or null if the action does not return value.
Throws:
ControlUnitException - if error prevents the execution of the action. ControlUnitException.getErrorCode() and ControlUnitException.getNestedException() methods can be used to determine the actual cause.
java.lang.NullPointerException - if the actionID is null.

4.6.1.3 Interface StateVariableListener

All Known Subinterfaces:
CUAdminCallback

---

public interface **StateVariableListener**

StateVariableListeners are registered as OSGi Services. For ManagedControlUnit and ControlUnitFactory instances whiteboard pattern [6] is used to notify the listeners, i.e. ControlUnitAdmin is responsible for tracking these services and to deliver them the appropriate events. ManagedControlUnit and ControlUnitFactory instances notify the ControlUnitAdmin about state variable changes through the CUAdminCallback provided to them upon registration in the framework by the ControlUnitAdmin.
Simple control units (those implementing only the ControlUnit interface) on the other hand have the responsibility to track and notify the listeners on their own.

A service registration property may be used as a filter to limit the number of received events and to specify certain control unit and/or state variables to listen for. The key of this property is ControlUnitConstants.EVENT_FILTER whose value is a String representing an LDAP filtering expression. The properties, which may be used in the LDAP filter are ControlUnitConstants.TYPE, ControlUnitConstants.ID and ControlUnitConstants.STATE_VARIABLE_ID. The listener will be notified only for changes in the values of state variables which satisfy this filter.

If property ControlUnitConstants.EVENT_FILTER is not present, the listener will receive events for all units state variables changes.

Here are some examples:

- To listen for all state variable changes on a certain control unit, construct a filter containing the ID and the type of the selected unit.
- To listen for all state variable changes on a group of similar control units, give only the type of the units.
- To listen for a change on specific variable, put the state variable ID in the filter.

Listeners may use the ControlUnitConstants.EVENT_SYNC property to specify that events should be delivered synchronously.
A listener that has the property ControlUnitConstants.EVENT_AUTO_RECEIVE automatically receives the current values of all registered control units and state variables that match the current filter.

Method Summary	
void	**stateVariableChanged**(java.lang.String controlUnitType, java.lang.String controlUnitID, java.lang.String stateVariableID, java.lang.Object value) This is the listener's callback method.

All Page Within This Box

Method Detail

stateVariableChanged

```
public void stateVariableChanged(java.lang.String controlUnitType,
                                 java.lang.String controlUnitID,
                                 java.lang.String stateVariableID,
                                 java.lang.Object value)
```

This is the listener's callback method. It is called when a state variable of a certain control unit is changed.

Parameters:

controlUnitType – The control unit type.
controlUnitID – The ID of the control unit whose variable has changed.
stateVariableID – The ID of the changed state variable.
value - The new value of the state variable.

4.6.1.4 Class ControlUnitException

```
java.lang.Object
 └java.lang.Throwable
   └java.lang.Exception
     └org.osgi.service.cu.ControlUnitException
```

All Implemented Interfaces:

java.io.Serializable

Direct Known Subclasses:

ControlUnitAdminException

---

public class **ControlUnitException**
extends java.lang.Exception
Custom exception thrown from some control unit related methods.
It has an error code, defining the type of the error that occurred and an optional nested exception.

See Also:

Serialized Form

Field Summary

static int	**ILLEGAL_ACTION_ARGUMENTS_ERROR** This error code means that the user has invoked an action with an invalid argument.
static int	**NO_SUCH_ACTION_ERROR** This error code means that the user has tried to invoke non-existent action on the control unit.
static int	**NO_SUCH_STATE_VARIABLE_ERROR** This error code means that the user has tried to read the value of non-existent state variable.
static int	**UNDETERMINED_ERROR** Error code which signals that an undetermined error has occurred.

Constructor Summary

ControlUnitException(int errorCode) Constructs a new control unit exception with the given error code.
ControlUnitException(java.lang.String message, int errorCode)

Constructs a new control unit exception with the given message and error code.
ControlUnitException(java.lang.String message, java.lang.Throwable exception) Constructs a new exception with the specified message and assigned nested error.
ControlUnitException(java.lang.Throwable exception) Creates a new exception with assigned nested error.

Method Summary

int	**getErrorCode**() Returns the error code for this control unit exception.
java.lang.Throwable	**getNestedException**() Returns the nested exception, if there is any.
void	**printStackTrace**() Prints the stack trace of the nested exception too (if there is one)
void	**printStackTrace**(java.io.PrintStream s) Prints the stack trace of the nested exception too (if there is one)
void	**printStackTrace**(java.io.PrintWriter s) Prints the stack trace of the nested exception too (if there is one)
java.lang.String	**toString**() Returns a short description of this exception.

Methods inherited from class java.lang.Throwable
fillInStackTrace, getLocalizedMessage, getMessage

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

4.6.2 UNDETERMINED_ERROR

public static final int **UNDETERMINED_ERROR**

Error code which signals that an undetermined error has occurred. The nested exception should be checked for more information about the actual error.

NO_SUCH_ACTION_ERROR

public static final int **NO_SUCH_ACTION_ERROR**

This error code means that the user has tried to invoke non-existent action on the control unit.

NO_SUCH_STATE_VARIABLE_ERROR

public static final int **NO_SUCH_STATE_VARIABLE_ERROR**

This error code means, that the user has tried to read the value of non-existent state variable.

ILLEGAL_ACTION_ARGUMENTS_ERROR

public static final int **ILLEGAL_ACTION_ARGUMENTS_ERROR**

This error code means that the user has invoked action with an invalid argument.

Constructor Detail

ControlUnitException

public **ControlUnitException**(int errorCode)

Constructs a new control unit exception with the given error code.

Parameters:

errorCode - the error code

ControlUnitException

public **ControlUnitException**(java.lang.String message,
int errorCode)

Constructs a new control unit exception with the given message and error code.

Parameters:

message - the detail message

errorCode - the error code

4.6.3 ControlUnitException

public **ControlUnitException**(java.lang.Throwable exception)

Creates a new exception with the given nested exception.

The error code of the constructed exception will be UNDETERMINED_ERROR. The nested exception may be retrieved by the getNestedException() method.

Parameters:

exception - the actual nested exception

4.6.4 ControlUnitException

public **ControlUnitException**(java.lang.String message, java.lang.Throwable exception)

Constructs a new exception with the specified message and assigned nested error. The error code of the constructed exception will be UNDETERMINED_ERROR. The nested exception may be retrieved by the getNestedException() method.

Parameters:

message - detail message

exception - the nested exception

Method Detail

getErrorCode

public long **getErrorCode**()

Returns the error code for this control unit exception.

Returns:

The error code

4.6.5 getNestedException

public java.lang.Throwable **getNestedException**()

Returns the nested exception, if there is any.
Returns:
The nested exception or null if there is no nested exception

4.6.6 printStackTrace

public void **printStackTrace**()

Prints the stack trace of the nested exception too (if there is one)
See Also:
Throwable.printStackTrace()

4.6.7 printStackTrace

public void **printStackTrace**(java.io.PrintWriter s)

Prints the stack trace of the nested exception too (if there is one)
Parameters:
s - PrintWriter to use for output
See Also:
Throwable.printStackTrace(java.io.PrintWriter)

4.6.8 printStackTrace

public void **printStackTrace**(java.io.PrintStream s)

Prints the stack trace of the nested exception too (if there is one)
Parameters:
s - PrintStream to use for output
See Also:
Throwable.printStackTrace(java.io.PrintStream)

---

toString

public java.lang.String **toString**()

Returns a short description of this exception. The description is a concatenation of String representation of the error code, the error message (if there is one) and - if there is a nested exception - the result from its toString() method.
See Also:
Object.toString()

4.6.9 Package org.osgi.service.cu.admin

4.6.9.1 Interface ControlUnitAdmin

---

public interface **ControlUnitAdmin**
Represents the facade of the Control Unit Admin layer. This interface is available as a service in the OSGi registry and may be used by the applications to manage all Control Units exported in the OSGi framework. There must be exactly one such service registered in the OSGi framework.
See Also:
ControlUnit

Field Summary

static java.lang.String	**CONSTRUCTOR_PREFIX** Actions defined in the metadata of the control unit whose action ID starts with this prefix may be used for explicit creation of new control units.
static java.lang.String	**DESTRUCTOR** An action with ID equal to the value of this constant and taking a single argument of type String should be defined in the metadata of a control unit to specify that the corresponding type of control units can be explicitly removed.
static java.lang.String	**EVENT_TYPE** This service registration property may be used by ControlUnitAdminListeners and HierarchyListeners as attribute in their filter definition to narrow the type of events they wish to receive.
static java.lang.String	**FINDER_PREFIX** Actions defined in the metadata of the control unit that starts with this prefix, may be used for searching the control units.

Method Summary

java.lang.String	**createControlUnit**(java.lang.String controlUnitType, java.lang.String constructorID, java.lang.Object arguments) Explicitly creates a control unit instance with the specified type using the supplied constructor and returns the ID of the newly created control unit.
void	**destroyControlUnit**(java.lang.String controlUnitType, java.lang.String controlUnitID) Explicitly removes the control unit instance with the given type and ID.
java.lang.String[]	**findControlUnits**(java.lang.String controlUnitType, java.lang.String finderID, java.lang.Object arguments) Returns the IDs of the control units with the specified type, located by the finder method with the given ID and the given finder parameters.
ControlUnit	**getControlUnit**(java.lang.String controlUnitType, java.lang.String controlUnitID) Returns the control unit with the given type and ID.
java.lang.String[]	**getControlUnitTypes**() Returns all distinct types of control units currently exported in the framework.
java.lang.String	**getControlUnitTypeVersion**(java.lang.String controlUnitType) Returns the current version for the given type of control units currently exported in the framework.
java.lang.String[]	**getParentControlUnits**(java.lang.String childControlUnitType, java.lang.String childControlUnitID, java.lang.String parentControlUnitType) Returns the IDs of the exported control units with the given type that are parents of the control units with the specified type and ID.
java.lang.String[]	**getParentControlUnitTypes**(java.lang.String childControlUnitType) Returns the types of the exported control units that may be parents of the control units with the given type.
java.lang.String[]	**getSubControlUnits**(java.lang.String parentControlUnitType, java.lang.String parentControlUnitID, java.lang.String childControlUnitType) Returns the IDs of the children of the control unit, specified by its type and ID, that have the given type.
java.lang.String[]	**getSubControlUnitTypes**(java.lang.String parentControlUnitType)

	Returns the types of the exported control units that may be children of the control units with the given type.
java.lang.Object	**invokeAction**(java.lang.String controlUnitType, java.lang.String controlUnitID, java.lang.String actionID, java.lang.Object arguments) Executes the given action over the control unit with the given type and ID.
java.lang.Object	**queryStateVariable**(java.lang.String controlUnitType, java.lang.String controlUnitId, java.lang.String stateVariableID) Queries the control unit with the given type and ID for the value of the given state variable.

Field Detail

CONSTRUCTOR_PREFIX

public static final java.lang.String **CONSTRUCTOR_PREFIX**

Actions defined in the metadata of the control unit whose action ID starts with this prefix may be used for explicit creation of new control units.

These actions are called constructors and their action ID must be supplied as argument to the ControlUnitAdmin.createControlUnit or ControlUnitFactory.createControlUnit methods. A control unit type may have arbitrary number of constructors.
The value of this constant is "$create."

DESTRUCTOR

public static final java.lang.String **DESTRUCTOR**

An action with ID equal to the value of this constant and taking a single argument of type String should be defined in the metadata of a control unit to specify that the corresponding type of control units can be explicitly removed.

This action is called destructor. It takes as single argument the ID of the control unit to be destroyed and its responsibility is to remove the control unit from the framework and to free the resources used by it. A control unit type may have one or zero destructor.

The destructor may be invoked through the ControlUnitAdmin.destroyControlUnit or ControlUnitFactory.destroyControlUnit methods.
The value of this constant is "$destroy"

FINDER_PREFIX

public static final java.lang.String **FINDER_PREFIX**

Actions defined in the metadata of the control unit that starts with this prefix may be used for searching the control units.
The value of this constant is "$find."

EVENT_TYPE

public static final java.lang.String **EVENT_TYPE**

This service registration property may be used by ControlUnitAdminListeners and HierarchyListeners as attribute in their filter definition to narrow the type of events they wish to receive. Its value must be a

String representation of one of the possible values for the event type parameter of ControlUnitAdminListener.controlUnitEvent(int, java.lang.String, java.lang.String) for ControlUnitAdminListener and HierarchyListener.hierarchyChanged(int, java.lang.String, java.lang.String, java.lang.String, java.lang.String) for HierarchyListener. If the filter doesn't restrict the event types to be received, the listener will receive all events matching the control unit-filtering criterion.
The value of this constant is "org.osgi.control.event.type"

Method Detail

getControlUnitTypes

public java.lang.String[] **getControlUnitTypes**()

Returns all distinct types of control units currently exported in the framework.
Returns:
An array of Control Unit types or null, if there are no Control Unit types exported in the framework.

---

getControlUnitTypeVersion

public java.lang.String **getControlUnitTypeVersion**(java.lang.String controlUnitType) throws ControlUnitAdminException

Returns the current version for given type of control units currently exported in the framework.
Parameters:
controlUnitType - The control unit type
Returns:
The version, or null, if the given type has no version
Throws:
ControlUnitAdminException - if there is no such type of control units exported in the framework
java.lang.NullPointerException - if the control unit type is null.

---

findControlUnits

```
public java.lang.String[] findControlUnits(java.lang.String controlUnitType,
                                           java.lang.String finderID,
                                           java.lang.Object arguments)
                                    throws ControlUnitException
```

Returns the IDs of control units of the specified type, located by the finder method specified by it's ID an according the given finder parameters. Supported finder methods are specific to the type of the control unit and are specified in the control unit metadata as a special class of actions whose identifier starts with "$find.".
Every finder method may have different number and/or type of arguments, which are also specified in the metadata.
Supplying null as finder method ID and arguments to this method returns all IDs of all control units of the specified control unit type. Otherwise the exact searching of the control units is delegated to the ControlUnitFactory for this control unit type. Therefore, invoking this method with non-null finder method ID and arguments is not supported for finding control units exported by ManagedControlUnit services.
If there are no control units that satisfy the finder condition the method returns null.
Parameters:
controlUnitType - the type restriction of the control units list that is searched.
finderID - the ID of the finder method. Must start with "$find.".
arguments - the finder argument(s). If the argument is only one this is the argument itself. If the arguments are more than one, the value must be an Object array and arguments are retrieved from it.
Returns:
An array of ControlUnit identifiers

Throws:
ControlUnitException - if the search operation cannot be performed due to an error. ControlUnitException.getErrorCode() and ControlUnitException.getNestedException() methods can be used to determine the actual cause.
ControlUnitAdminException - if there is no such a type of control units available in the framework, searching is not supported by this type or it does not have a finder with the given finder method ID.
java.lang.NullPointerException - if the control unit type or finder method ID are null.

---

getControlUnit

public ControlUnit **getControlUnit**(java.lang.String controlUnitType,
java.lang.String controlUnitID) throws ControlUnitAdminException

Returns the control unit with the specified type and ID.
Parameters:
controlUnitType - the type of the control unit
controlUnitID - the ID of the control unit
Returns:
The ControlUnit identified by the specified type and ID pair or null if there is no such control unit exported in the framework.
Throws:
ControlUnitAdminException - if there is no such type of units exported in the framework
java.lang.NullPointerException - if the control unit type or ID is null.

---

getSubControlUnits

public java.lang.String[] **getSubControlUnits**(java.lang.String parentControlUnitType,
java.lang.String parentControlUnitID,
java.lang.String childControlUnitType)

Returns the IDs of the children of the control unit, specified by its type and ID, that have the given type. If both the parent control unit type and ID are null, this method returns the IDs of the control units with the specified type that have no parent.
Parameters:
parentControlUnitType - the type of the parent control unit.
parentControlUnitID - the ID of the parent control unit.
childControlUnitType - the type of the child control units.
Returns:
An array of child control units or null if there are no child control units found for this parent type and ID.
Throws:
java.lang.NullPointerException - if the type of the children control units is null.

---

getParentControlUnitTypes

public java.lang.String[] **getParentControlUnitTypes**(java.lang.String childControlUnitType)

Returns the types of the exported control units that are parents of the control units with the given type.
Parameters:
childControlUnitType - The type of the children control units
Returns:
The types of the parent control units or null if the given children control units have no parents
Throws:
java.lang.NullPointerException - if the type of the children control units is null.

---

getParentControlUnits

public java.lang.String[] **getParentControlUnits**(java.lang.String childControlUnitType,
java.lang.String childControlUnitID,
java.lang.String parentControlUnitType) throws
ControlUnitAdminException

Returns the IDs of the exported control units with the given type that are parents of the control unit with the given type and ID.
Parameters:
childControlUnitType - The child control unit type
childControlUnitID - The child control unit ID
parentControlUnitType - The parent control units type
Returns:
The IDs of the parent control units or null if the given child control unit has no parents of the given type
Throws:
ControlUnitAdminException - if there is no such child control unit provided in the framework
java.lang.NullPointerException - if any parameter is null.

---

createControlUnit

public java.lang.String **createControlUnit**(java.lang.String controlUnitType,
java.lang.String constructorID,
java.lang.Object arguments)
throws ControlUnitException

Explicitly creates control unit instance of a specified type using a supplied constructor and returns the ID of the newly created control unit. Supported ways for creating a control unit are specific to the type of the control unit and are specified in the corresponding metadata definition. For every control unit type there may be defined zero, one or more constructors. A constructor is defined as a special class of action whose identifier starts with "$create.".
Every constructor may have different number and/or types of arguments, which are also defined in the metadata.
The exact creation of the control unit is delegated to the ControlUnitFactory of this control unit type. Therefore this method is not supported for control units exported by ManagedControlUnit services.
Parameters:
controlUnitType - - the type of the control unit.
constructorID - the ID of the constructor. Must start with "$create.".
arguments - - the 'constructors' argument(s). If the argument is only one this is the argument itself. If the arguments are more than one, the value must be an Object array and arguments are retrieved from that array.
Returns:
The ID of the newly created control unit
Throws:
ControlUnitException - if the control unit cannot be created for some reason. ControlUnitException.getErrorCode() and ControlUnitException.getNestedException() methods can be used to determine the actual cause.
ControlUnitAdminException - if there is no such a type of control units available in the framework, creation is not supported for this type or it does not have a constructor with the given constructor ID.
java.lang.NullPointerException - if the control unit type or the constructor ID is null.

---

destroyControlUnit

public void **destroyControlUnit**(java.lang.String controlUnitType,

java.lang.String controlUnitID)
throws ControlUnitException

Explicitly removes the control unit instance with the given type and ID. Some type of control units may not support explicit removing. In that case this method throws ControlUnitAdminException. Support for explicit removing of control units is specified in the control unit metadata by the presence of an action with ID "$destroy".

Parameters:
controlUnitType - The type of the Control Unit.
controlUnitID – The control unit ID.
Throws:
ControlUnitException - if the control unit cannot be destroyed for some reason. ControlUnitException.getErrorCode() and ControlUnitException.getNestedException() methods can be used to determine the actual cause.
ControlUnitAdminException - if there is no such a type of control units available in the framework or destruction of control units is not supported for this type.
java.lang.NullPointerException - if the control unit type or control unit ID is null.

---

getSubControlUnitTypes

public java.lang.String[] **getSubControlUnitTypes**(java.lang.String parentControlUnitType)

Returns the types of the exported control units that are the children of the control units with the given type.

Parameters:
parentControlUnitType - The parent control unit type
Returns:
An array of the children control unit types
Throws:
java.lang.NullPointerException - if the parent control unit type is null.

---

queryStateVariable

public java.lang.Object **queryStateVariable**(java.lang.String controlUnitType,
java.lang.String controlUnitId,
java.lang.String stateVariableID)
throws ControlUnitException

Queries a control unit with a given type and ID for the value of a given state variable.

Parameters:
controlUnitType - the type of the control unit
controlUnitId - the ID of the control unit
stateVariableID - the ID of the variable
Returns:
The value of the variable
Throws:
ControlUnitException - if the state variable's value cannot be retrieved for some reason. ControlUnitException.getErrorCode() and ControlUnitException.getNestedException() methods can be used to determine the actual cause.
ControlUnitAdminException - if there is no such a type of control units available in the framework or a control unit with the given contro unit type and control unit ID does not exist.
java.lang.NullPointerException - if any parameter is null.

---

invokeAction

```
public java.lang.Object invokeAction(java.lang.String controlUnitType,
                                     java.lang.String controlUnitID,
                                     java.lang.String actionID,
                                     java.lang.Object arguments)
                              throws ControlUnitException
```

Executes a given action over a control unit with a given ID.

Parameters:
controlUnitType - the type of the control unit
controlUnitID - the ID of the control unit
actionID - the ID of the action
arguments - the input argument(s). If the argument is only one this is the argument itself. If the arguments are more than one, the value must be an Object array and arguments are retrieved from it.

Returns:
The output argument(s) or null if the action does not return value.

Throws:
ControlUnitException - if an error prevents the execution of the action. ControlUnitException.getErrorCode() and ControlUnitException.getNestedException() methods can be used to determine the actual cause.
ControlUnitAdminException - if there is no such a type of control units available in the framework or a control unit with the given controlUnitType and controlUnitID does not exist.
java.lang.NullPointerException - if either the control unit type, the control unit ID or the action ID is null.

4.6.9.2 Interface ControlUnitAdminListener

All Known Subinterfaces:
CUAdminCallback

---

public interface **ControlUnitAdminListener**
Applications interested in receiving events for creation and/or deletion of a specified set of control unit instances may implement this interface and register it as a service in the OSGi registry. Control Unit Admin service is responsible for tracking these services and to deliver the appropriate events to them.
A service registration property may be used as a filter to limit the number of received events and to specify certain control unit to listen for. The key of the property is ControlUnitConstants.EVENT_FILTER whose value is a String representing an LDAP filtering expression. The properties that may be used in the LDAP filer are ControlUnitAdmin.EVENT_TYPE, ControlUnitConstants.TYPE and ControlUnitConstants.ID.
The filter states what types of events, and events for which control units to be received. The listener will only be notified for events regarding control units whose ID and type satisfy this filter, if the events match the event-filtering criterion. If property ControlUnitConstants.EVENT_FILTER is not present, the listener will receive events for all control units.
Listeners may use the ControlUnitConstants.EVENT_SYNC property to specify that events must be delivered to them synchronously.

Field Summary	
static int	**CONTROL_UNIT_ADDED** This constant is used as first parameter in controlUnitEvent method to indicate that a new control unit is available.
static int	**CONTROL_UNIT_REMOVED** This constant is used as first parameter in controlUnitEvent method to indicate that a control unit is not available anymore.
static int	**CONTROL_UNIT_TYPE_ADDED** This constant is used as first parameter in controlUnitEvent method to indicate that a new control unit

All Page Within This Box

	type is available, i.e. a control unit of a type (or a control unit factory providing control units of a type) not available before in the framework was registered. When the event is of this type the parameter of controlUnitEvent method specifying a control unit ID is null.
static int	**CONTROL_UNIT_TYPE_REMOVED** This constant is used as first parameter in controlUnitEvent method to indicate that control units of the given type are no more available, i.e. the last control unit of this type (or the factory providing control units of this type) was unregistered from the framework.

Method Summary

void	**controlUnitEvent**(int eventType, java.lang.String controlUnitType, java.lang.String controlUnitID) Invoked by the Control Unit admin service when a control unit instance has been created or removed.

Field Detail

CONTROL_UNIT_ADDED

public static final int **CONTROL_UNIT_ADDED**

This constant is used as first parameter in controlUnitEvent method to indicate that a new control unit is available.
The value of this constant is 1

CONTROL_UNIT_REMOVED

public static final int **CONTROL_UNIT_REMOVED**

This constant is used as first parameter in controlUnitEvent method to indicate that a control unit is not available anymore.
The value of this constant is 2

CONTROL_UNIT_TYPE_ADDED

public static final int **CONTROL_UNIT_TYPE_ADDED**

This constant is used as first parameter in controlUnitEvent method to indicate that a new control unit type is available, i.e. a control unit of a type (or a control unit factory providing control units of a type) not available before in the framework was registered.
When the event is of this type the parameter of controlUnitEvent method specifying a control unit ID is null.
The value of this constant is 3

CONTROL_UNIT_TYPE_REMOVED

public static final int **CONTROL_UNIT_TYPE_REMOVED**

This constant is used as first parameter in controlUnitEvent method to indicate that control units of the given type are no more available, i.e. the last control unit of this type (or the factory providing control units of this type) was unregistered from the framework.
When the event is of this type the parameter of controlUnitEvent method specifying a control unit ID is null.
The value of this constant is 4

Method Detail

controlUnitEvent

```
public void controlUnitEvent(int eventType,
                             java.lang.String controlUnitType,
                             java.lang.String controUnitID)
```

Invoked by the ControlUnitAdmin service when a control unit instance has been created or removed.
Parameters:
eventType - one of CONTROL_UNIT_ADDED, CONTROL_UNIT_REMOVED, CONTROL_UNIT_TYPE_ADDED or CONTROL_UNIT_TYPE_REMOVED
controlUnitType - the type of the control unit instance
controUnitID - the ID of the control unit instance

4.6.9.3 Interface HierarchyListener

All Known Subinterfaces:
CUAdminCallback

---

public interface **HierarchyListener**
Applications interested in receiving events for changes in the hierarchy of control units (attaching of a control unit to new parent or detaching it from old one) may implement this interface and register it as a service in the OSGi registry. Control Unit Admin service is responsible for tracking these services and for delivering the appropriate events to them.
A service registration property may be used as a filter to limit the number of received events and to specify certain control unit and/or event types to listen for. The key of this property is ControlUnitConstants.EVENT_FILTER whose value is a String representing an LDAP filtering expression.
The properties, which may be used in the LDAP filter are:

- ControlUnitAdmin.EVENT_TYPE - to limit the types of events received by the listener. Valid values are DETACHED and ATTACHED.
- ControlUnitConstants.TYPE and ControlUnitConstants.ID are used to limit events based on the child control units to which the events are relevant.
- ControlUnitConstants.PARENT_TYPE and ControlUnitConstants.PARENT_ID can be used to specify that the listener is only interested in receiving events about hierarchy changes regarding given parent control units.

Field Summary

Static int	**ATTACHED** This constant is used as first parameter in hierarchyChanged(int, java.lang.String, java.lang.String, java.lang.String, java.lang.String) method to indicate that the control unit has attached to the given parent.
Static int	**DETACHED** This constant is used as first parameter in hierarchyChanged(int, java.lang.String, java.lang.String, java.lang.String, java.lang.String) method to indicate that the control unit has detached from the given parent.

Method Summary

void	**hierarchyChanged**(int eventType, java.lang.String controlUnitType, java.lang.String controlUnitID, java.lang.String parentControlUnitType, java.lang.String parentControlUnitID) This callback method is invoked from the ControlUnitAdmin in order to notify the registered listeners for a new hierarchy event.

Field Detail

DETACHED

public static final int **DETACHED**

This constant is used as first parameter in hierarchyChanged(int, java.lang.String, java.lang.String, java.lang.String, java.lang.String) method to indicate that the control unit has detached from the given parent.
The value of this constant is 1

ATTACHED

public static final int **ATTACHED**

This constant is used as first parameter in hierarchyChanged(int, java.lang.String, java.lang.String, java.lang.String, java.lang.String) method to indicate that the control unit has attached to the given parent.
The value of this constant is 2

Method Detail

hierarchyChanged

```
public void hierarchyChanged(int eventType,
                             java.lang.String controlUnitType,
                             java.lang.String controlUnitID,
                             java.lang.String parentControlUnitType,
                             java.lang.String parentControlUnitID)
```

This callback method is invoked from the ControlUnitAdmin in order to notify the registered listeners for a new hierarchy event.
Hierarchy events are sent when a registered control unit changes its position in the control units' hierarchy or when a new control unit, which has a parent specified, is registered/unregistered.
ControlUnits and ManagedControlUnits change their position in the hierarchy by modifying their ControlUnitConstants.PARENT_TYPE and ControlUnitConstants.PARENT_ID service registration properties.
ControlUnitFactories should notify via the hierarchyChanged(...) method of CUAdminCallback when a control unit provided by it has changed its position in the hierarchy.

Parameters:

eventType - the type of the event - either ATTACHED or DETACHED.
controlUnitType - the type of the control unit for which the event is fired
controlUnitID - the ID of the control unit for which the event is fired
parentControlUnitType - the parent control unit's type, where the change occurred
parentControlUnitID - the parent control unit's ID, where the change occurred

4.6.9.4 Class ControlUnitAdminException

java.lang.Object

```
└ java.lang.Throwable
  └ java.lang.Exception
    └ org.osgi.service.cu.ControlUnitException
      └ org.osgi.service.cu.admin.ControlUnitAdminException
```

All Implemented Interfaces:
java.io.Serializable

---

public class **ControlUnitAdminException**
extends ControlUnitException
ControlUnitAdminException is a subclass of ControlUnitException, which provides additional codes for errors which may arise when operating over control units through the ControlUnitAdmin and ControlUnitFactories..
See Also:
Serialized Form

Field Summary

static int	**CREATION_NOT_SUPPORTED_ERROR** This error code means that someone has tried to create a control unit dynamically, but it's factory doesn't provide any constructor methods.
static int	**DESTRUCTION_NOT_SUPPORTED_ERROR** This error code means that the user has tried to destroy a control unit which factory doesn't define a destructor method.
static int	**NO_SUCH_CONSTUCTOR_ERROR** This error code means that the user attempted to create a control unit, but it's factory doesn't provide the requested constructor method.
static int	**NO_SUCH_CONTROL_UNIT_ERROR** This error code means that the user tried to perform an operation over non-existent control unit.
static int	**NO_SUCH_CONTROL_UNIT_TYPE_ERROR** This error code means that the user tried to perform an operation over non-existent control unit type.
static int	**NO_SUCH_FINDER_ERROR** This error code means that the user tried to perform a search, but the there is not finder method that matches the given finder ID.
static int	**SEARCHING_NOT_SUPPORTED_ERROR** This error code means that the user has tried to search for a control unit but no finder method is defined.

Fields inherited from class org.osgi.service.cu.ControlUnitException
ILLEGAL_ACTION_ARGUMENTS_ERROR, NO_SUCH_ACTION_ERROR, NO_SUCH_STATE_VARIABLE_ERROR, UNDETERMINED_ERROR

Constructor Summary

ControlUnitAdminException(int errorCode) Constructs a new control unit exception with the given error code.
ControlUnitAdminException(java.lang.String message, int errorCode) Constructs a new control unit exception with the given message and error code.
ControlUnitAdminException(java.lang.String message, java.lang.Throwable exception)

Constructs a new undetermined error exception with the given message and exception.
ControlUnitAdminException(java.lang.Throwable exception) Constructs a new undetermined error control unit exception with the given nested exception.

Methods inherited from class org.osgi.service.cu.ControlUnitException
getNestedException, getErrorCode, printStackTrace, printStackTrace, printStackTrace, toString

Methods inherited from class java.lang.Throwable
fillInStackTrace, getLocalizedMessage, getMessage

Methods inherited from class java.lang.Object
Clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

NO_SUCH_CONTROL_UNIT_ERROR

public static final int **NO_SUCH_CONTROL_UNIT_ERROR**

This error code means that the user tried to perform an operation over non-existent control unit.

NO_SUCH_CONSTUCTOR_ERROR

public static final int **NO_SUCH_CONSTUCTOR_ERROR**

This error code means that the user attempted to create a control unit, but it's factory doesn't provide the requested constructor method.

NO_SUCH_FINDER_ERROR

public static final int **NO_SUCH_FINDER_ERROR**

This error code means that the user tried to perform a search, but the there is not finder method that matches the given finder ID.

CREATION_NOT_SUPPORTED_ERROR

public static final int **CREATION_NOT_SUPPORTED_ERROR**

This error code means that someone has tried to create a control unit dynamically, but it's factory doesn't provide any constructor methods.

DESTRUCTION_NOT_SUPPORTED_ERROR

public static final int **DESTRUCTION_NOT_SUPPORTED_ERROR**

This error code means that the user has tried to destroy a control unit which factory doesn't define a destructor method.

SEARCHING_NOT_SUPPORTED_ERROR

public static final int **SEARCHING_NOT_SUPPORTED_ERROR**

This error code means that the user has tried to search for a control unit but no finder method is defined.
See Also:
Constant Field Values

NO_SUCH_CONTROL_UNIT_TYPE_ERROR

public static final int **NO_SUCH_CONTROL_UNIT_TYPE_ERROR**

This error code means that the user tried to perform an operation over non-existent control unit type.
See Also:
Constant Field Values

Constructor Detail

ControlUnitAdminException

public **ControlUnitAdminException**(int errorCode)

Constructs a new control unit exception with the given error code.
Parameters:
errorCode - the error code

ControlUnitAdminException

```
public ControlUnitAdminException(java.lang.String message,
                                 int errorCode)
```

Constructs a new control unit exception with the given message and error code.
Parameters:
message - the detail message
errorCode - the error code

4.6.10 ControlUnitAdminException

public **ControlUnitAdminException**(java.lang.Throwable exception)

Constructs a new control unit exception with the given nested exception.
The error code of the constructed exception will be ControlUnitException.UNDETERMINED_ERROR.
The nested exception may be retrieved by the ControlUnitException.getNestedException() method.
Parameters:
exception - the nested exception

4.6.11 ControlUnitAdminException

public **ControlUnitAdminException**(java.lang.String message, java.lang.Throwable exception)

Constructs a new control unit exception with the given message and exception.
The error code of the constructed exception will be ControlUnitException.UNDETERMINED_ERROR.
The nested exception may be retrieved by the ControlUnitException.getNestedException() method.
Parameters:
message - detail message
exception - the nested exception

4.6.12 Package org.osgi.service.cu.admin.spi

4.6.12.1 Interface ManagedControlUnit

All Superinterfaces:

ControlUnit

---

public interface **ManagedControlUnit**
extends ControlUnit
This interface must be registered as an OSGi service in order to make a single resource manageable through the control unit abstraction.
The ManagedControlUnit services should not be used directly by the applications. Instead applications access them as ControlUnit instances obtained via the ControlUnitAdmin service. The ControlUnitAdmin service is responsible for tracking all ManagedControlUnit services registered in the service registry of the framework and to notify registered ControlUnitAdminListeners when a new ManagedControlUnit appears or an existing one is unregistered. To be properly handled by the Control Unit Admin service the ManagedControlUnit service must be registered with the following required properties:

- Property ControlUnitConstants.TYPE with value of type String specifying the type of the control unit instance.
- Property ControlUnitConstants.ID with value of type String specifying the identifier of the control unit instance.

Optionally the registration properties may contain properties ControlUnitConstants.PARENT_ID and ControlUnitConstants.PARENT_TYPE, both with values of type String specifying the parent control unit in the control unit hierarchy.
Note that the control units exported through ManagedControlUnit may specify only one control unit as a parent, while control units registered by a ControlUnitFactory may have more than one parent.
ManagedControlUnits may dynamically change their parent by modifying PARENT_TYPE and/or PARENT_ID in their service registration properties.
Control Units that support versioning of their type should additionally register with the property ControlUnitConstants.VERSION with value of type String.

Method Summary

void	**setControlUnitCallback**(CUAdminCallback adminCallback) Supplies the Control Unit admin callback interface to the implementation of the ManagedControlUnit service.

Methods inherited from interface org.osgi.service.cu.ControlUnit
getId, getType, invokeAction, queryStateVariable

Method Detail

setControlUnitCallback

public void **setControlUnitCallback**(CUAdminCallback adminCallback)

Supplies the Control Unit Admin callback interface to the implementation of the ManagedControlUnit service.
This method is invoked by the Control Unit Admin service with a non- null argument after registration of the ManagedControlUnit service or after startup of the Control Unit Admin for already registered control units.
It is supposed that the Managed Control Unit will assign this reference to an instance variable and use it later to notify the Control Unit Admin for changes in the state variables of the control unit.
The method is invoked with a null argument during unregistration of the ManagedControlUnit service or when the Control Unit Admin is stopped.

Parameters:
adminCallback - reference to the control unit callback interface or null if previously set reference is not longer valid.

4.6.12.2 Interface CUAdminCallback

All Superinterfaces:
ControlUnitAdminListener, HierarchyListener, StateVariableListener

---

public interface **CUAdminCallback**
extends ControlUnitAdminListener, StateVariableListener, HierarchyListener
Represents the interface of the ControlUnitAdmin service provided to the implementations of the ManagedControlUnit and ControlUnitFactory. ManagedControlUnit and ControlUnitFactory instances use the methods of this interface to notify the ControlUnitAdmin service for changes of the state variables. Control unit factories also use this interface to notify the ControlUnitAdmin service for appearance and disappearance of control unit instances maintained by the factory.
See Also:
ManagedControlUnit.setControlUnitCallback(CUAdminCallback), ControlUnitFactory.setControlUnitCallback(CUAdminCallback)

Fields inherited from interface org.osgi.service.cu.admin.ControlUnitAdminListener
CONTROL_UNIT_ADDED, CONTROL_UNIT_REMOVED, CONTROL_UNIT_TYPE_ADDED, CONTROL_UNIT_TYPE_REMOVED

Fields inherited from interface org.osgi.service.cu.admin.HierarchyListener
ATTACHED, DETACHED

Methods inherited from interface org.osgi.service.cu.admin.ControlUnitAdminListener
controlUnitEvent

Methods inherited from interface org.osgi.service.cu.StateVariableListener
stateVariableChanged

Methods inherited from interface org.osgi.service.cu.admin.HierarchyListener
hierarchyChanged

4.6.12.3 Interface ControlUnitFactory

---

public interface **ControlUnitFactory**
This interface must be registered as an OSGi service in order to make more than one resource of the same type manageable through the control unit abstraction. The ControlUnitFactory services should not be used directly by the applications. Instead applications access ControlUnitAdmin service, which delegates the requests to the appropriate ControlUnitFactory or ManagedControlUnit service. ControlUnitFactory objects are suitable for representing variable number of resources with similar characteristics. An advantage is that it is not necessary to permanently hold control unit instance for every resource, because the corresponding wrapper may be created on demand.
ControlUnitAdmin service is responsible for tracking all ControlUnitFactory services registered in the service registry of the framework.

One ControlUnitFactory may be responsible for providing ControlUnit instances of exactly one type. It is not allowed to have more than one factory for the same type and it is not allowed to have both ControlUnitFactory and ManagedControlUnit services for the same control unit type.
To be properly handled by the ControlUnitAdmin, the ControlUnitFactory service must be registered with property ControlUnitConstants.TYPE with value of type String specifying the type of the control unit instances provided by this factory. Optionally the registration properties may contain property ControlUnitConstants.PARENT_TYPE with value of type String or String[] specifying the type(s) of parent control units in the control unit hierarchy.
Factories which support versioning of their control units' type should additionally register with the property ControlUnitConstants.VERSION with value of type String.

Method Summary

java.lang.String	**createControlUnit**(java.lang.String constructorID, java.lang.Object arguments) Explicitly creates a control unit and returns the ID of the newly created control unit.
void	**destroyControlUnit**(java.lang.String controlUnitID) Explicitly removes the control unit instance with the given ID.
java.lang.String[]	**findControlUnits**(java.lang.String finderID, java.lang.Object arguments) Returns the IDs of the control units satisfying the finder method and the supplied argument(s).
ControlUnit	**getControlUnit**(java.lang.String controlUnitID) Returns the ControlUnit object identified by the given ID.
java.lang.String[]	**getControlUnits**(java.lang.String parentControlUnitType, java.lang.String parentControlUnitID) Returns the IDs of the control unit instances, which are children of the control unit with the given type and ID.
java.lang.String[]	**getParents**(java.lang.String childControlUnitID, java.lang.String parentControlUnitType) Returns the IDs of the parents of the given control unit specified by its ID.
java.lang.Object	**invokeAction**(java.lang.String controlUnitID, java.lang.String actionID, java.lang.Object arguments) Executes the specified action over the control unit with specified ID.
java.lang.String[]	**listControlUnits**() Returns the IDs of all control units currently provided by this factory.
java.lang.Object	**queryStateVariable**(java.lang.String controlUnitID, java.lang.String stateVariableID) Queries the control unit with the specified ID for the value of the specified state variable.
void	**setControlUnitCallback**(CUAdminCallback adminCallback) Supplies the CUAdminCallback interface to the implementation of the ControlUnitFactory service.

Method Detail

setControlUnitCallback

public void **setControlUnitCallback**(CUAdminCallback adminCallback)

Supplies the CUAdminCallback callback interface to the implementation of the ControlUnitFactory service.
This method is invoked by the ControlUnitAdmin service with a non-null argument after registration of the ControlUnitFactory service or after start-up of the ControlUnitAdmin for already registered factories.

It is supposed that the ControlUnitFactory will assign this reference to an instance variable and use it later to notify the ControlUnitAdmin for creation or removal of control unit instances, attaching/detaching of control units to/from given parent and for changes in state variables.
The method is invoked with a null argument during un-registration of the ControlUnitFactory service or when the ControlUnitAdmin is stopped.
Parameters:
adminCallback - reference to the control unit callback interface or null if previously set reference is not longer valid.

---

getControlUnit

public ControlUnit **getControlUnit**(java.lang.String controlUnitID)
Returns the ControlUnit object identified by the given ID. If there is no such control unit maintained by this factory, null is returned.
Parameters:
controlUnitID – The ID of the requested control unit
Returns:
The ControlUnit object with the given ID

---

getControlUnits

public java.lang.String[] **getControlUnits**(java.lang.String parentControlUnitType,
java.lang.String parentControlUnitID)
Returns the IDs of the control unit instances, which are children of the control unit with the given type and ID.
Supplying null as arguments to this method results in returning only those control units provided by this factory, which have no parent, omitting all other provided control units.
Parameters:
parentControlUnitType – The type of the parent control unit
parentControlUnitID – The ID of the parent control unit
Returns:
The sub-control units of the specified control unit

---

listControlUnits

public java.lang.String[] **listControlUnits**()
Returns the IDs of all control units currently provided by this factory.
Returns:
The control unit IDs provided by this factory or null it doesn't provide any control unit.

---

findControlUnits

public java.lang.String[] **findControlUnits**(java.lang.String finderID,
java.lang.Object arguments)
throws ControlUnitException
Returns the IDs of the control units satisfying the finder method and the supplied argument(s). The ControlUnitFactory may optionally define in its metadata one or more methods for filtering control units. Those methods are called finders and are specified in the metadata definition for the particular type of control units. A finder is defined as a special class of action whose identifier starts with "$find.". Every finder method may have different number and/or type of arguments, specified in the metadata and implemented by the corresponding ControlUnitFactory.
If there is no control unit that satisfies the finder condition the method returns null.

Parameters:
finderID – The ID of the finder method. Must start with "$find.".
arguments – The finder argument(s). If the argument is only one this is the argument itself. If the arguments are more than one, the value must be an Object array and arguments are retrieved from it
Returns:
the sub-control units of the specified control unit.
Throws:
ControlUnitException - if the search operation cannot be performed due to an error. ControlUnitException.getErrorCode() and ControlUnitException.getNestedException() methods can be used to determine the actual cause.
ControlUnitAdminException - if searching is not supported by the factory or there is no finder with the given finderID.

---

getParents

public java.lang.String[] **getParents**(java.lang.String childControlUnitID,
java.lang.String parentControlUnitType) throws ControlUnitAdminException

Returns the IDs of the parents of the given control unit specified by its ID.
This method returns only IDs of the control units for the specified parent type
Parameters:
childControlUnitID –The ID of the child control unit
parentControlUnitType – The type of the returned parent control units
Returns:
The IDs of parent control units or null, if the given control unit has no parents of the specified type
Throws:
ControlUnitAdminException - if there is no such child control unit provided in the framework

---

queryStateVariable

public java.lang.Object **queryStateVariable**(java.lang.String controlUnitID,
java.lang.String stateVariableID)
throws ControlUnitException

Queries the control unit with the specified ID for the value of the specified state variable.
Parameters:
controlUnitID – The ID of the control unit provided by this factory
stateVariableID – The ID of the variable
Returns:
The value of the variable
Throws:
ControlUnitException - if the state variable's value cannot be retrieved for some reason. ControlUnitException.getErrorCode() and ControlUnitException.getNestedException() methods can be used to determine the actual cause.
ControlUnitAdminException - if a control unit with the given control unit ID does not exist.

---

invokeAction

public java.lang.Object **invokeAction**(java.lang.String controlUnitID,
java.lang.String actionID,
java.lang.Object arguments)
throws ControlUnitException

Executes the specified action over a control unit with the specified ID.

Parameters:
controlUnitID – The ID of the control unit
actionID – The ID of the action
arguments – The input argument(s). If the argument is only one this is the argument itself. If the arguments are more than one, the value must be an Object[] and arguments are retrieved from that array.
Returns:
The output argument(s) or null if the action does not return value
Throws:
ControlUnitException - if an error prevents the execution of the action.
ControlUnitException.getErrorCode() and ControlUnitException.getNestedException() methods can be used to determine the actual cause.
ControlUnitAdminException - if a control unit with the given control unit ID does not exist.

---

createControlUnit

public java.lang.String **createControlUnit**(java.lang.String constructorID,
java.lang.Object arguments)
throws ControlUnitException

Explicitly creates a control unit and returns the ID of the newly created control unit. The ControlUnitFactory may optionally define in its metadata one or more methods for creating new control units. These methods are called constructors and are specified in the metadata definition for the particular type of control units. A constructor is defined as a special class of action whose identifier starts with "$create.". Every constructor method may have different number and/or types of arguments, specified in the metadata and implemented by the corresponding ControlUnitFactory.
Parameters:
constructorID – The ID of the constructor. Must start with "$create.".
arguments – The 'constructor' argument(s). If the argument is only one this is the argument itself. If the arguments are more than one, the value must be an Object array and arguments are retrieved from it.
Returns:
The ID of the newly created control unit.
Throws:
ControlUnitException - if the control unit cannot be created for some reason.
ControlUnitException.getErrorCode() and ControlUnitException.getNestedException() methods can be used to determine the actual cause.
ControlUnitAdminException - if creation is not supported by the factory or there is no constructor with the given constructor ID.

---

destroyControlUnit

public void **destroyControlUnit**(java.lang.String controlUnitID)
throws ControlUnitException

Explicitly removes the control unit instance with the given ID. Some type of control units may not support explicit removing of the resources represented by the corresponding control units. In that case this method throws ControlUnitAdminException. Support for explicit destroying of control units is specified in the control unit metadata by the presence of an action with ID "$destroy".
Parameters:
controlUnitID – The control unit ID.
Throws:
ControlUnitException - if the control unit cannot be destroyed for some reason.
ControlUnitException.getErrorCode() and ControlUnitException.getNestedException() methods can be used to determine the actual cause.
ControlUnitAdminException - if destruction of control units is not supported by the factory.

5 Drawbacks and Issues

#	Issues	Comments
1	Do we need conditional constraints for actions?	For a unified user interface it might be useful to know what particular action is available when a given state variable satisfies some conditions. In this case, the user interface can properly disable invalid (for current state) actions. We can add this functionality in the properties of the `ExtendedAttributeDefinitions`, but if we may also need to specify the format, how these constraints are given
2	Flood of Events	If a device has a state variable that is a timer, it is non-sense to send event change for every millisecond. Can we prevent devices from generating thousand of events in a very short period?
3	Security	Do we need to provide means to limit access only for certain state variables and actions: • The management services are allowed to access all actions • End-user services, like general control unit browsers or customized user-interface bundle are disallowed to change the network address (setAddress actions) and upload a new firmware (updateFirmware action).
4	Signaling action invocation errors	The proposed interfaces don't define what exception is thrown if action fails. Obviously there are two runtime exceptions that are possible and we don't need to define them in the throws clause: • `IllegalStateException` – when execution of action is not allowed, considering the current state of device • `IllegalArgumentException` - when some of the parameters are of invalid types. However, there are situations that the action invocation must start network operation. If that operation fails what kind of exception must be thrown. Maybe we should define an `ActionInvocationException` from which we

		might
5	Implementation of listener notification might be too complex	The listeners are registered with LDAP filter as property. To match that filter a Dictionary object must be created and filled for each control unit and matched against every registered listener.
6	Do we need serialization	If we are using control units to synchronize them with remote, management platform do we need to define that `ControlUnit` implements `Serializable`?
7	Multiple ways of control unit notification	Currently, when control unit is not created from factory it is registered as OSGi Service. The user of that control unit might track OSGi registry for changes or may register `ControlUnitListener`. Isn't this flexibility a little bit confusing?

6 Considered Alternatives

6.1 Java Beans

Java Beans are proven and world-spread alternative. The beans will vary in the functionality they support, but the typical unifying features that distinguish a Java Bean are:

- Support for “introspection” so that a builder tool can analyze how a bean works
- Support for “customization” so that when using an application builder a user can customize the appearance and behavior of a bean.
- Support for “events” as a simple communication metaphor than can be used to connect up beans.
- Support for “properties”, both for customization and for programmatic use.
- Support for persistence, so that a bean can be customized in an application builder and then have its customized state saved away and reloaded later.

A bean is not required to inherit from any particular base class or interface. Visible beans must inherit from java.awt.Component so that they can be added to visual containers, but invisible beans aren’t required to do this.

Therefore, we can also use Beans instead of Control Units.

6.1.1 Advantages

- The Beans architecture is proven, stable and works.
- As the beans specification relies on build-in Java Reflection mechanism, there is no need for Meta typing.
- There are not direct dependences on any Java packages.

6.1.2 Drawbacks

- The beans are accessed always through reflection. Reflection is fast if methods are cached but in dynamic environment such as OSGi, that caching might cause a memory leaks – e.g. the package was uninstalled but a `PropertyDescriptor` object is still kept in the cache.
- Beans introspection gives information only about accepted Java types. It doesn’t give information if the parameter must be within given range, or incremented by step, or accepts only some predefined values.
- Even if there is visual property editor of the bean, it is useless in OSGi because the application must mutually import (in the manifest file) the package of the property editor to instantiate it. This is possible only with latest framework implementations that support dynamic import of packages.

- The properties of the bean are directly set. This situation is rarely happening in the real world.

6.2 Diagnostics RFC

Control Units and the Diagnostics RFC are presenting the same technology but in a different way.

Excluding the API difference, there are some other minor differences:

- Control Units are using optional Meta typing to reduce the usual overhead created by the textual descriptions and reuses the OSGi Meta Typing API.
- The Status class is removed as it is not required in the Control Units API and the Diagnosable RFC didn't present a reasonable usage of that class.
- Uses Object instead of `Object[]` for input and output parameters. This will eliminate the need to create a new `Object[]` on every action execution, even if the parameter is only one.
- As description is option, it is considered harmful to leave the methods that are listing the attributes and actions. That list in control unit is obtained from the meta-typing instead of the control unit itself.

6.3 OSGi UPnP API

In general, both API – Control Unit and OSGi UPnP are providing similar technologies for accessing devices. Here is a brief comparison of these technologies:

- Control Unit API is smaller and more compact.
- UPnP leverages IP and XML technologies, which also might be used by Control Unit implementations, but it's not necessary.
- The Meta typing is optional for Control Units and for UPnP it's part of the protocol.
- UPnP Supports `Date` and `TimeZone` attribute types.

6.4 Wire Admin API

See RFC 77 – Diagnostic in chapter 6.1

6.5 SyncML DM

6.5.1 Objectives

Though an OSGi gateway can reside alone in the home environment or the vehicle, the most common situation involves a gateway operator that performs some management functions.

The gateway operator manages a single repository of services and controls a number of gateways. If a gateway operation is critical for the deployed target it is very important that gateway operator performs some diagnostic tests over all gateway installations. So a question raises – how the operator should perform that diagnostics and how system health information is transferred between the management server and the gateway.

In order to provide the most flexible mechanism we must fulfill at least these common requirements:

- The diagnostics framework must be flexible enough to handle various types of diagnostics states
- If repair is possible it should allow to perform some operation to restore the proper operation or at least to minimize the problems
- As there could be various diagnostics states and devices the framework should be self-describing and provides meaningful information about the available diagnostics parameters and their types
- Operator must be always informed about the current state of each device
- To stay informed, the data is synchronized from the gateway to the operator, whenever the gateway stays online
- The synchronization mechanism must use common standards (if possible) to be adopted easily not only from our management products but from other, maybe competitive or supplementary platforms.

The answer to the first thee question could be the Control Units. However Control Units are never mend to be synchronized remotely or fulfill the latest requirement.

SyncML DM, in other hand is the answer to the latest three questions but it is XML based, much more complex and thus – not suitable to represent device states in the framework. But it well adopted by the wireless community and is de-facto standard for remote devices synchronization and management.

The obvious solution is to use the SyncML DM protocol to transfer the states of the Control Units and to invoke the provided action.

6.5.2 Device Representation

SyncML DM doesn't have means for a 'device'. Instead, the most important element is the 'node'. The nodes are organized into a tree with one static root node named '.'.

This tree forms a core component of the SyncML specification named a 'device description framework'. It is similar to control unit in functionality:

- To provide a self-describing unit
- To allow reading and changing of certain parameters (nodes)
- To execute some action upon a node

There are two types of nodes:

- Leaf Nodes – that can accept values
- Interior Nodes – that doesn't have any values but have some sub nodes which can be either Leaf or Interior

The specification defines that a Management Object could be the whole tree or only a node. This could be quite useful as you can provide global operations on the tree and local – on a certain node.

There are some common (Runtime) elements of the Node that are required for interoperability:

- ACL – access control list to limit the number of domain that can perform certain operation on a node or sub-tree.
- Format – specifies the type of the property, only byte[], boolean, String and int are supported. This is quite limiting compared to the types supported by the Control Unit.
- Name – the name of the node that must be unique to the nodes at the same level
- Size – specifies the size of the property (I cannot figure it out why this is needed)
- Title – an user-friendly description of the node
- TStamp – an optional parameter that specifies the latest time when the property was changed.
- Type – a content transfer type (MIME)

Additionally, each vendor may define other node properties.

6.5.3 Mapping between SyncML DM & Control Unit

6.5.3.1 Control Unit Tree

If control units were organized in a tree-like manned, the tree would look like:

```
- Management root (that stays on operator platform)
  - Gateway root (this is a sub-node when resides on management platform or the root node on a local gateway)
    - CU
      - Parent CU1
      - Parent CU2
   - CU3
```

6.5.3.2 SyncML Tree

```
- ./ (that stays on management platform)
  - Gateway root (this is a sub-node when resides on management platform or the root node [./] on a local gateway)
    - Parent CU (interior node)
      - CU1 (interior node – metadata copied from ControlUnitMetaData)
        - StateVar1 (leaf node – has value)
        - StateVar2
        - StateVar3
```

- Action1 (leaf node that supports Exec command, metadata available from `ControlUnitMetaData`)

- CU2

  - StateVar1

  - StateVar2

  - StateVar3

  - Action1

It is obvious that in this case the Control Unit is not the deepest node in the tree, but each of its state variables and actions.

The actions are also leaf nodes and might have value – the return value of the action. However, their specialty is the support of the ‘Exec’ action. They also support a list of input arguments and are not limited to only one.

6.5.4 SyncML DM Restrictions

As seen from the examples above there are some restrictions that should be applied to Control Units in order to be successfully mapped to SyncML DM protocol. These restrictions are:

- The control unit should support less Data types – a byte[], String and int
- To successfully map an action to SyncML it should have one or none output arguments.
- As the Type property is required for the leaf nodes, the framework that does the mapping must have knowledge of the data type of the corresponding state variable. Therefore only Control Units that has a CU Adapter providing their metadata are supported. Control Units without metadata cannot be mapped to SyncML DM.

7 Security Considerations

The Control Unit functionality should be available only to well-known set of bundles because execution of an action might have effect on the whole system.

The bundles, that are intended to listen for State Variables or changes in the Control Unit Tree are required to have `ServicePermission[REGISTER,StateVariableListener|ControlUnitListener|HierachyListener]`.

The other trusted bundles must have `ServicePermission[GET,ControlUnit|ControlUnitAdmin]` permissions assigned.

There is no security limitation for those that would like to register an ControlUnit or ControlUnitFactory implementations.

8 Document Support

8.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. OMA-SyncML-DMTND-V1_1_2-20030612-C.pdf

[4]. OMA-SyncML-DMRepPro-V1_1_2-20030612-C.pdf

[5]. http://www-306.ibm.com/software/wireless/wme/features.html

[6]. Technical Whitepaper: Listeners Considered Harmful: The "Whiteboard" Pattern

8.2 Author’s Address

Name	Valentin Valchev
Company	ProSyst Software AG
Address	D-50858 Cologne, Germany. Dürener Strasse 405
Voice	+359 2 952 35 81 (107)
e-mail	v_valchev@prosyst.bg

8.3 Acronyms and Abbreviations

8.4 End of Document