Pegasus Enhancement Proposal (PEP)

PEP#: 348

PEP Type: Functional

Title: The CMPI infrastructure using SCMO.

Status: Approved

Version History:

Version	Date	Author	Change Description
0.1	07 July 2009	Thilo Boehm	Initial draft
0.2	10 July 2009	Thilo Boehm	After Review Arch Call 07/09/09
0.3	30 July 2009	Thilo Boehm	Work in several comments.
0.4	26 August 2009	Marek Szermutzky	Add XmlWriter design
0.5	15. October 2009	Thilo Boehm	Add SCMOClass Cache.
0.6	05. November 2009	Marek Szermutzky	Add Data Transport and Transformation
1.0	19. November 2009	Thilo Boehm	Updates for ballot
1.1	12. January 2010	Thilo Boehm	Updates from ballot: - Rework pictures (remove figure of round robin class cache) - Add findings. Updates from implementation: - SCMOClassCache: The OOP Agent communication with the CIM server is using pipes.

Abstract:

In PEP341 the concept of SCMO (Single Chunk Memory Objects) was introduced. This PEP defines the implementation the usage of SCMO within the CMPI provider manager and processing of SCMO instances in the CIM Server.

Definition of the Problem

One of the performance gluttons of the CMPI Provider Manager and CMPI providers is the object oriented structure of the internal data representation and the semantic of the internal interface.
The semantic of the internal interface is a pure C++ programming language style and cannot be changed. The reason is that it is closely interwoven with the external interface, defined by PEP#344, and can not easily be split such that the external interface is not broken. CMPI is a C language style interface with a limited functionality on instances. One difference for instances the semantic logic used to set a property value on an instance object.

To further optimize the CMPI performance of the CIM Server, the CMPI implementation has to use a different implementation of instances

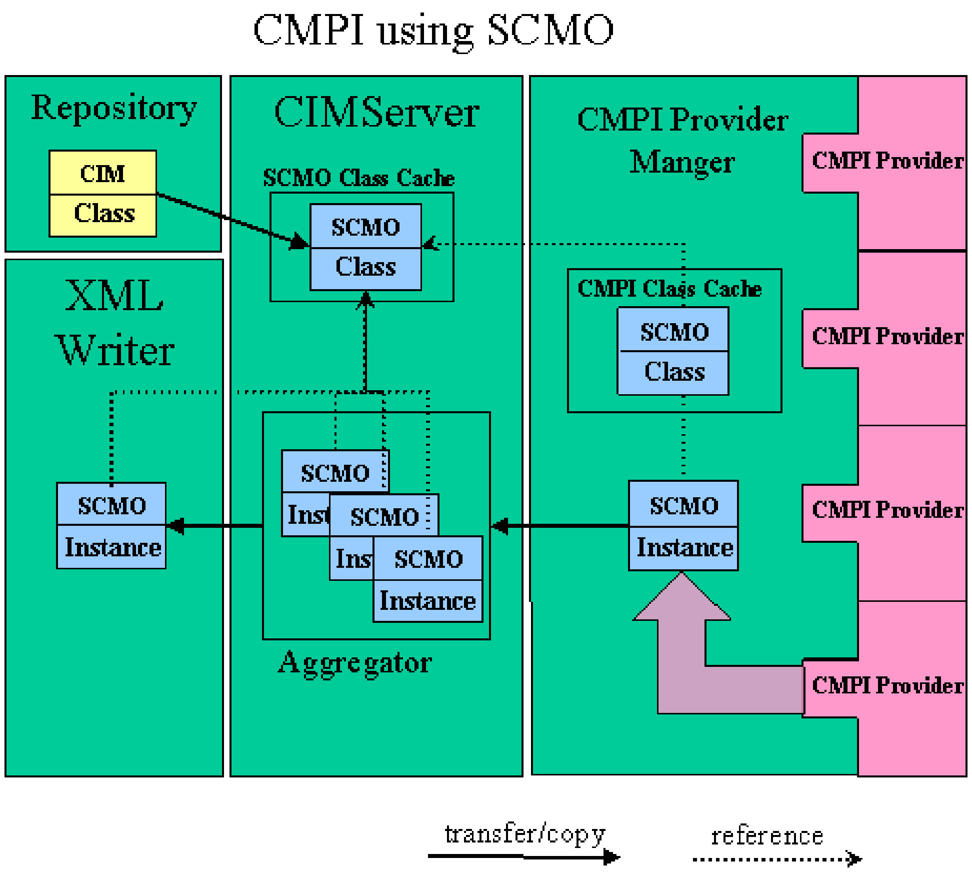
Proposed Solution

For CMPI only the usage of CIMClasses and CIMInstances will be replaced by SCMO objects. At the CMPI Provider Manager the CMPI interface implementation will be working with SCMO objects.
The handling thought the server and the end points (like the XMLWriter) has to be enabled to handle SCMO objects. This will be an alternate way in addition to the existing implementation. For example: The CIMInstanceResponseData class will be utilized to transport the SCMO objects through the server.
New end points (like the XMLWriter) have to be designed and to be implemented to avoid conversion of SCMO objects.

To be able to convert CIM objects into SCMO objects a new CIM Server wide class cache is needed for SCMO classes. The reason for not enabling the existing repository class cache to handle SCMO Classes is, that SCMO Classes also needed to be available in the provider agent and the repository is only available with in the main CIM server process.

Within the CMPI Provider Manager the existing CMPI Class Cache is changed to cascade over the new SCMO classes cache.
The current CMPI class cache handling/algorithm will **not** be changed. In the current implementation a class is retrieved only once from the repository into the CMPI Class cache when the provider has a demand for that class. e.g to create an instance. There is no logic in place to check for an update of a class in the repository. So there will be no

update in the CMPI Class cache if a class is changed in the Repository. The lifecycle of the cache is bound to the CMPI Broker for one provider. That means the cache is destroyed only when the provider is unloaded.



The SCMO Class Cache

Hard requirements for the class cache:

1. The class definitions must be available in the CIM server and OOP agent.
2. The size must be limited
3. The implementation must be thread save with minimal serialization
4. Design is optimized for the CMPI/SCMO scenario.
5. Hold the currently used classes in cache.
6. The look up of the classes must be fast.
7. Class name and name space name are keys.

Soft requirements for the class cache:

1. The cache size is defined by the number of classes to be stored in the cache
2. The number should match that of the repository class cache.
3. The number is defined at compile time.
4. Must be suppressible to save memory for e.g. embedded systems.
5. Keep memory usage low.
6. Avoid duplicate caching.
7. Class modifications are picked up by the cache for consistency

Design

The cache is organized as an indexed array used in round robin.

The SCMO Class cache is implemented as a singleton. The class cache is instantiated at global scope in the CIM server and OOPAgent and initialized with a callback function for retrieving CIM Classes from the repository. The retrieved classes are converted to SCMOClasses and stored in the cache.

```
    typedef CIMClass (*SCMOClassCacheCallbackPtr)(
            const CIMNamespaceName& nameSpace,
            const CIMName& className);
```

In the CIM server the callback function is directly using the repository.

Retrieving SCMOClasse in the OOP agent.

To be able to retrieve SCMOClasses in the OOP agent the already established pipe connection is used. The ratrional to use the pipe connection is, during the CIM server start up indication subscriptions are setup and for CMPI providers SCMOClasses are needed but the communication with the CIMOMHandle, using the CIMClient, is disabled.

There for two new messages are introduced:

1. *ProvAgtGetScmoClassRequestMessage* (CIMNamespaceName nameSpace_, CIMName className)
2. *ProvAgtGetScmoClassResponseMessage* (SCMOClass scmoClass)

The following describes the logical flow between OOP agent and CIM server to retrieve a SCMOClass:

1. The OOP agent call back function ProviderAgent::_scmoClassCache_GetClass() sends a *ProvAgtGetScmoClassRequestMessage()* to the CIM server over the pipe and wait for the posting of the _scmoClassDelivered semaphore. If the semaphore is not posted until PEGASUS_DEFAULT_CLIENT_TIMEOUT_MILLISECONDS are reached, the call back is not successful and an empty SCMOClass is returned.

2. The CIM server reads the *ProvAgtGetScmoClassRequestMessage()* in *ProviderAgentContainer::_processResponses()*, requests the SCMOClass from the CIM server SCMOClassCache, and sends the *ProvAgtGetScmoClassResponseMessage()* to the OOP agent using the existing pipe.

3. The OOP agent receives the *ProvAgtGetScmoClassResponseMessage()* in *ProviderAgent::_readAndProcessRequest(),* saves the SCMOClass in *_transferSCMOClass* and posts the *scmoClassDelivered* semaphore.

4. The *_scmoClassCache_getClass()* function reads the SCMClass from *_transferSCMOClass* and delivers it to the caller.

Expiration rules

The if the cache is full, the oldest element is released. If the cache is not full all cache entries are remaining in the cache.

If a CIMClass has been removed form the repository, it will also be removed from the cache.

If a CIMClass has been modified in the repository, the whole cache will be cleared.

SCMBClassCacheEntry

```
    struct SCMBClassCacheEntry
    {
        // Spin-lock to serialize the access to the entry
        AtomicInt lock;
        // The key to identify the entry
        Uint64    key;
        // Pointer to the cached SCMOClass
        SCMOClass* data;
    };
```

The lock

This lock indicates that a thread is using the entry read-only. This lock is used to protect read operations from write operations when an entry needs to be updated.

A cache entry lock is implemented via a spin-lock over an atomic int. The lock is obtained when the atomic lock counter increment results in a lock count of 1. Otherwise the lock count is decremented and incremented again until we end up at 1.

The reason to use a spin-lock is the assumption that the lock is held only while executing very few instructions before the lock is released again. To search for an entry only the key is compared and if equal, the class name and name space have to be compared.

The key

An entry is identified by the key. A key consists of:

- the length of the class name,
- the last and the first character of the UTF8 representation of the class name,
- the length of the name space name,
- and the first and the last character of the UTF8 representation of the name space name.

These values are accumulated to a single Uint64:

```
Uint64   key  =  (Uint64(classNameLen) << 48 ) |
                 (Uint64(className[0]) << 40 ) |
                 (Uint64(className[classNameLen-1]) << 32 ) |
                 (Uint64(nameSpaceNameLen) << 16 ) |
                 (Uint64(nameSpaceName[0]) << 8 ) |
                 Uint64(nameSpaceName[nameSpaceNameLen-1]);
```

The data

The data contains a pointer to a SCMOClass. SCMOClasses are reference counted. So as long as a SCMOClass is referenced by the cache it will not be removed from memory. If a SCMOClass is removed from the cache, only the reference count is decreased. As long as a SCMOInstance or the CMPIClassCache are referencing a class it resides in memory until the last reference is destroyed.

SCMOClassCache class

The SCMO Class Cache is an array of SCMBClassCacheEntries with size:

```
#define PEGASUS_SCMO_CLASS_CACHE_SIZE   32
```

One SCMOClass is about 30 KByte in size. With a size of 32 the cache requires about 1MByte of memory.
For suppressing the cache PEGASUS_CLASS_CACHE_SIZE is set to 0. The caching itself is then disabled and compiled out but the functionality to retrieve a SCMOClass is still available. The request for a SCMOClass is then simply passed through with out caching.

To accelerate consecutive retrievals of the same class from the class cache, the cache remembers the position of the last successfull retrieval and always starts searching at that position. This optimization is based on the assumption that processing CIM requests which yield multiple instances will cause 'number-of-instances' consecutive look-up's for the same class in the cache.

```
    class PEGASUS_COMMON_LINKAGE SCMOClassCache
    {

        /**
         * This function returns the SCMOClass for the given class name and
         * name space.
         * @param ndName The UTF8 encoded name space. '\0' terminated
         * @param nsNameLan The strlen of ndName ( without '\0')
         * @param className The UTF8 encoded class name. '\0' terminated
         * @param nsNameLan The strlen of className ( without '\0')
         * @return The SCMOClass. If the class was not found, an empty
         *         SCMOClass is returned.  This can be checked by using the
         *         SCMOClass.isEmpty() method.
         **/
        SCMOClass getSCMOClass(
            const char* nsName,
            Uint32 nsNameLen,
            const char* className,
            Uint32 classNameLen);

        /**
```

```
         * Removes the named SCMOClass from the cache.
         * @param cimNameSpace The name space name of the SCMOClass to
remove.
         * @param cimClassName The class name of the SCMOClass to remove.
         **/
        void removeSCMOClass(CIMNamespaceName cimNameSpace,CIMName
cimClassName);

        /**
         * Clears the whole cache.
         * This should be only done at modification of a class.
         * This may invalidate subclass definitions in the cache.
         * Since class modification is relatively rare, we just flush the
entire
         * cache rather than specifically evicting subclass definitions.
         **/
        void clear();

        /**
         * Returns the pointer to an instance of SCMOClassCache.
         */
        static SCMOClassCache* getInstance();

        /**
         * Set the call back function for the SCMOClass to retrieve
CIMClasses.
         * @param clb The static call back function.
         */
        void setCallBack(SCMOClassCacheCallbackPtr clb)
        {
           _resolveCallBack = clb;
        }

        static void destroy();


    private:

        // Singleton instance pointer
        static SCMOClassCache* _theInstance;

        // The call back function pointer to get CIMClass's
        SCMOClassCacheCallbackPtr _resolveCallBack;

        // The cache array
        SCMBClassCacheEntry _theCache[PEGASUS_SCMO_CLASS_CACHE_SIZE];

        // Lock to prevent parallel modifications of the cache.
        ReadWriteSem _modifyCacheLock;

        // Last successful read index.
        Uint32 _lastSuccessIndex;

        // Last successful written cache index.
        Uint32 _lastWrittenIndex;

        // Counter of used cache entries.
        Uint32 _fillingLevel;

        // Indicator for destruction of the cache.
        Boolean _dying;

    };
```

Class cache Use cases

The following use case are written in pseudo code. These cases only describe the straight forward algorithm.

How to find a SCMOClass

```
calculate search key from input;

start index = _lastSuccessIndex % PEGASUS_SCMO_CLASS_CACHE_SIZE ;

next index = start index;

for all cache entries:
{
        obtain the lock for the next index entry in the cache;

        ? is Key of entry equal to search key ?

        Yes --> ? is class name and name space of entry equal with input ?
                Yes -->
                        _lastSuccessIndex = next index;
                        save SCMOClass pointer;
                        unlock of the entry;
                        return SCMOClass pointer;
        unlock of the entry;
        get next entry index;
}
```

How to add a SCMOClass

```
obtain _modifyCacheLock;

To determine whether the class was already added while waiting for the _modifyCacheLock, search the cache
again.

        Note:
        In this scenario is it not required to obtain further locks,
        because no modification can occur while we hold the _modifyCacheLock;
        Other threads can continue to search and find entries in the cache.


for all cache entries:
{
        ? is Key of entry equal to search key ?

        Yes --> ? is class name and name space of entry equal with input ?
                Yes -->
                        _lastSuccessIndex = next index;
                        unlock _modifyCacheLock;
                        return SCMOClass pointer;

        get next entry index;
}

Use call back function to get the CIM class;

? Not Found ?
        unlock _modifyCacheLock;

        return NULL;

Transform CIM class into SCMOClass;

The index for the entry to be written is the next index to _lastWritten ( _lastWritten + 1 %
PEGASUS_SCMO_CLASS_CAHE_SIZE)

get entry lock;
```

```
update entry with key and SCMOClass;

release entry lock;

update _lastSuccessIndex = next index;

Update _lastWritten = next index;

release _modifyChacheLock;
```

The SCMO Data model

The focus of this PEP is on implementing a fast delivery instances from CMPI providers.
To be interoperable within OpenPegasus (e.g. C++ provider up-calls) converting functions will be provided to convert SCMO to C++ objects and vice versa.
The current C++ object model is not impacted by this implementation.

Basics

Data objects in SCMO are residing in a single chunk memory block. The SCMOClass contains all static data, data which does not vary between instances, (like qualifier, class origin etc.) and the SCMOInstance the dynamic data (like property values).

The size of a SCMOInstance is small due to the constant data is stored only once in a SCMOClass. Many instances are connected to the same class.

The design for SCMO is optimized to set data once, and not for data modification/removal.

The single chunk memory block of a SCMO object consists of a header section and a dynamic section.

To distinguish between the SCMO class object and the layout of the single chunk memory block, the suffix of the structures names is SCMB.

Structures used to build the main header sections use the suffix **`_Header`** (e.g. SCMBObjectPath_Header). The nature of these header structures is, that the size is known at compile time.

All other structures are components to manage the dynamic section of the SCMO object.

SCMOClass and SCMOInstance are reference counted to make them easier usable. This also prevents for unnecessary copies of the object, memory leaks, and addressing failures due to already released memory.

Addressing

The data in SCMOClass and SCMOInstance is stored in a single chunk memory block.
A data object within the single chunk memory block is described by a Uint64 byte start index relative to the pointer to the single chunk memory block and it's content length.
All addressing is done relative to the start of the single chunk memory block. This implies that for all valid references the start index is greater than 0. A SCMBDataPtr NULL pointer is defined by start index set to 0.
The rationale for this addressing is

- easy reallocation. If the allocated single chunk memory block is not large enough to hold all data, more memory has to be reallocated. The reallocated memory might not be at the same address as it was before but a relative pointer remains valid.
- easy transferable. If the single chunk memory block is transferred between processes, a relative pointer remains valid.
- Relative pointers are using Uint64 values to be prepared for 64Bit environments.

This the definition of a relative pointer for objects stored in the single chunk memory. The memory is handled as array of chars.

```
struct SCMBDataPtr
{
    // start index of the data area
    Uint64      start;
    // size of data area
    Uint64      size;
};
```

Memory management

To be able to use common SCMO functionality for managing a single chunk memory block, a SCMO common control structure is needed.
This structure must be located at the beginning of the main header of the SCMO objects to be addressable without knowing the type of the SCMO object.

```
struct SCMBMgmt_Header
{
    // The magic number for a SCMB memory object
    Uint32          magic;
    // Total size of the SCMB memory block( # bytes )
    Uint64          totalSize;
    // The # of bytes available in the dynamic area of SCMB memory
block.
    Uint64          freeBytes;
    // Index to the start of the free space in this SCMB memory block.
    Uint64          startOfFreeSpace;
    // Number of external references in this instance.
    Uint32          numberExtRef;
    // Size of external reference index array;
    Uint32          sizeExtRefIndexArray;
    // Relative pointer to the external reference Array.
    SCMBDataPtr     extRefIndexArray;
};
```

External references are objects allocated outside of the single memory block.
For a SCMOInstance these are objects like embedded instances, objects. references, all implemented using SCMOInstances.
To be able to access the external references of a SCMOInstances, an array of relative start pointers is managed.
If an external references is added/set, the relative start pointer of this external reference is added to the array.
If the SCMOInstance is deleted, cloned,or binary transferred the relative index is used to straight address the external references and managed them.

The array is only allocated, if the a SCMOInstance contains external references.

```
#define PEGASUS_SIZE_REFERENCE_INDEX_ARRAY 8
```

If the number of references of the SCMOInstance exceeds this number, the array is growing by PEGASUS_SIZE_REFERENCE_INDEX_ARRAY.

The reference to the related SCMOClass has to be managed manually.

Data mapping

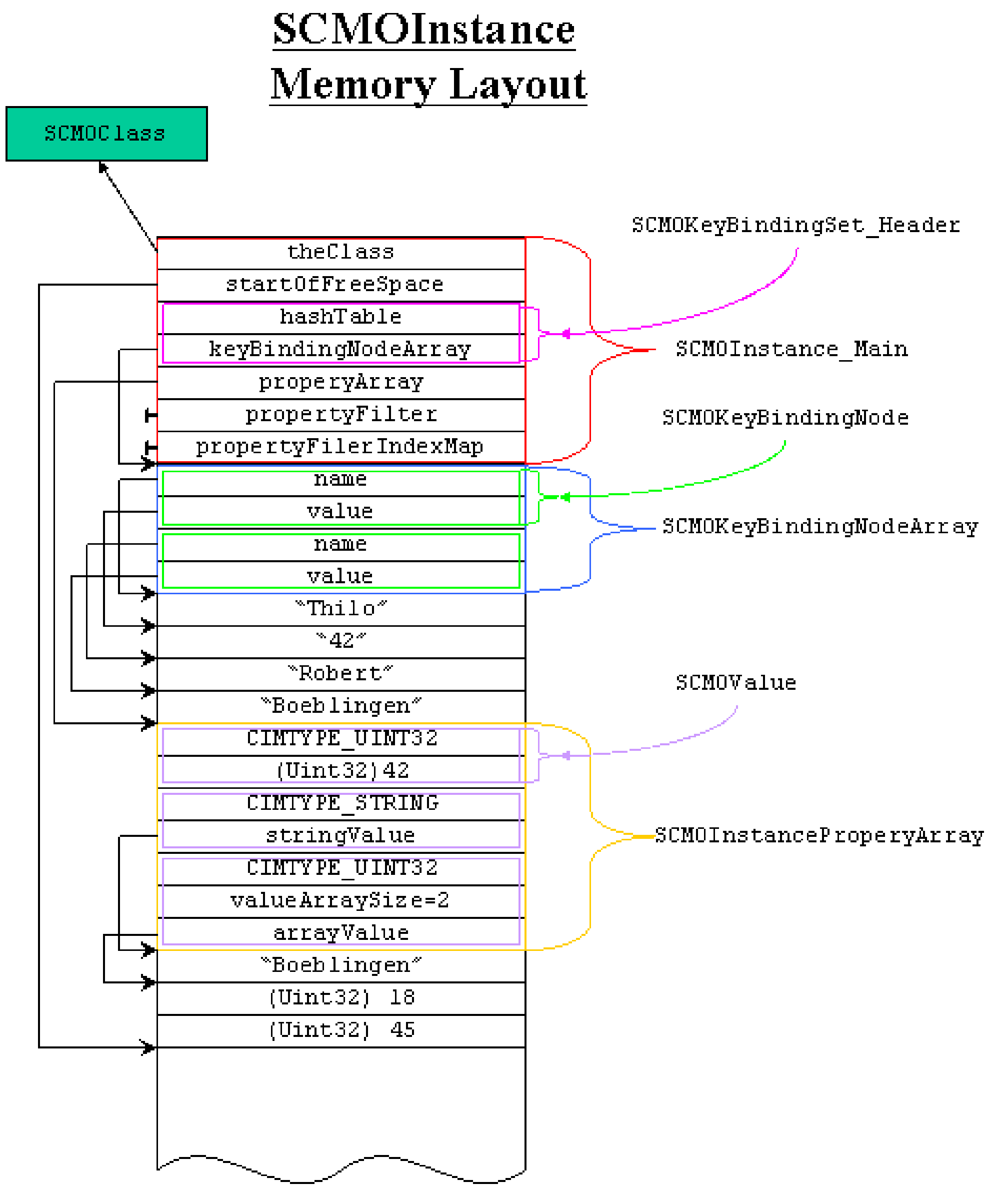
In SCMB structures the primitive types of <Pegasus/Common/Config.h> are used.

Open Pegasus Strings are stored in character strings (char*) with UTF-8 encoding.

Memory Layout

To visualize the general memory layout, an instance with three properties is printed here. Two of of the properties are key properties.
This picture is a general view of a SCMB instance memory layout. Some structure members are not itemized to keep the picture simple.
The whole structures are described in more detail later in this PEP.

The whole block is a single chunk memory block. The arrows are relative pointers within the block. With one exception: theClass is a external pointer to the related SCMOClass memory block.



Error handling

SCMO mainly uses return values to report the result of a method.
Only at disaster conditions like out of memory an exception is throw.

```
// The enum of return values for SCMO functions.
enum SCMO_RC
{
    SCMO_OK = 0,
    SCMO_NULL_VALUE,
    SCMO_NOT_FOUND,
    SCMO_INDEX_OUT_OF_BOUND,
    SCMO_NOT_SAME_ORIGIN,
    SCMO_INVALID_PARAMETER,
    SCMO_TYPE_MISSMATCH,
    SCMO_WRONG_TYPE,
    SCMO_NOT_AN_ARRAY,
    SCMO_IS_AN_ARRAY
};
```

SCMO Ordered Set

To manage properties and key properties the algorithm of Pegasus/Common/OrderedSet.h has been adapted to SCMO.
The reason for using an ordered set is that properties have to be accessible by index and by name.
The hash code algorithm is the same as for Pegasus/Common/OrderedSet.h .

In contrast to the Pegasus/Common/OrderedSet.h implementation, a node in the array is addressed by an index instead by a pointer.

The `hashTable[]` is used for name based lookups based on the name tags.
The index of the `hashTable[]` is calculated doing a remainder operator with the name tag and hash size (NameTag % 64)

The `hashTable[]` contains the index+1 of the node in the `nodeArray[]`. The reason for shifting one up is to define 0 as an invalid index entry in the `hashTable[]`

The `nodeArray[]` consists of nodes containing the property or a relative pointer to the property and link to the next node containing a property with the same hash.

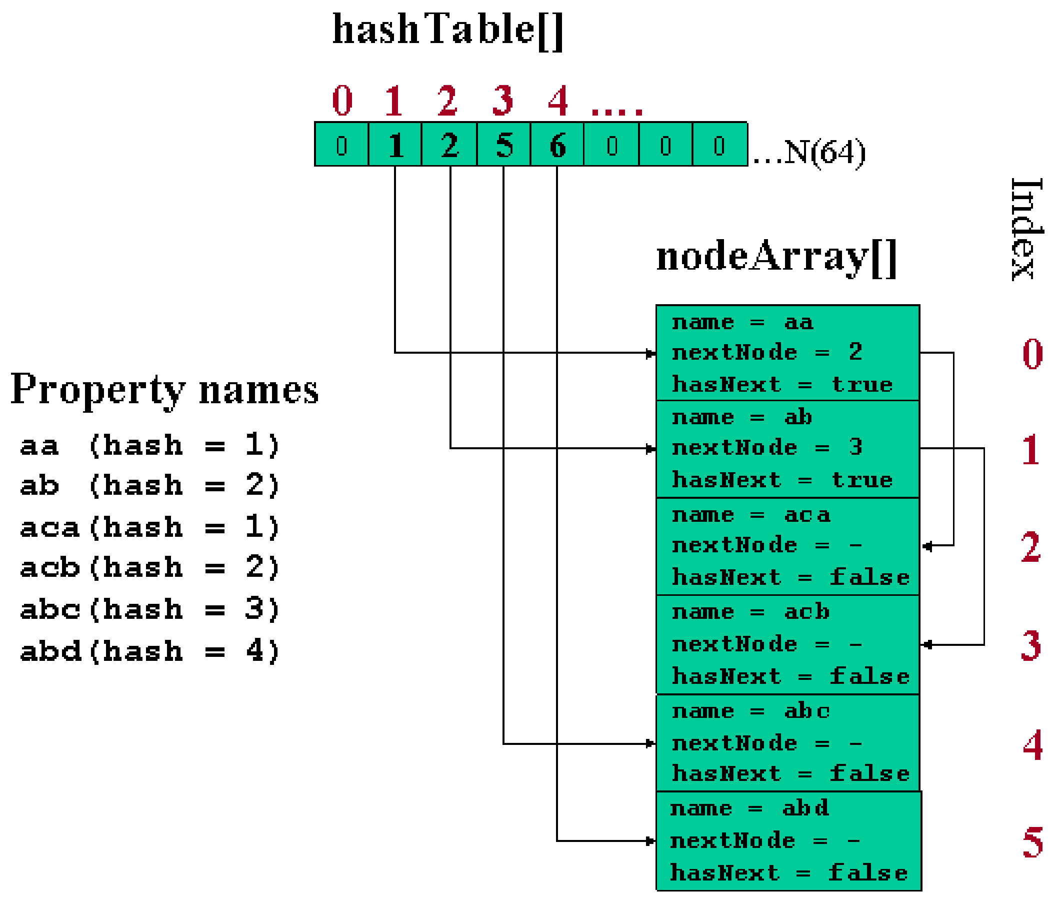
To decide, if the property was found, the names in the linked node chain have to be compared with the search property name.

If the access to the properties are done directly by index, the look up is only done in the `nodeArray[]`.

Example:

The picture below shows an example of an SCMO ordered set. There are six property names with a hash calculated on the first and last letter of the property name.

To find the property with the name **aca**, the hash has to be calculate. In this case the hash is 1. In the `hashTable[]` at index 1 the value 1 is found. 1-1= 0 is the index of the property node in the `nodeArray[]`. The member **`Next`** at the node indicates that there is a next property with the same hash, so the names have to be compared. But the property node belongs to the property with the name **aa**. So the **`Next`** node (2) has to be looked at. At index 2 of the `nodeArray[]` the property node for the property name **aca** is found.


hashTable[]
0 1 2 3 4 ....
0 1 2 5 6 0 0 0 ...N(64)
nodeArray[]
Index
Property names
aa (hash = 1)
ab (hash = 2)
aca (hash = 1)
acb (hash = 2)
abc (hash = 3)
abd (hash = 4)
name = aa nextNode = 2 hasNext = true
name = ab nextNode = 3 hasNext = true
name = aca nextNode = - hasNext = false
name = acb nextNode = - hasNext = false
name = abc nextNode = - hasNext = false
name = abd nextNode = - hasNext = false
0 1 2 3 4 5

CMPIObjectPath implementation using SCMO

CMPI implements CMPIObjectPath using the SCMOInstance as the data container.
In CMPI an object path can be created providing a class name and name space.
At creation of an object path the CMPI layer retrieves/creates the corresponding SCMOClass and creates a SCMOInstance from it.
If the SCMOClass is not found, the SCMOInstance is for the object path still created but marked as compromised to indicate that it was created for a stale class.

The rationale to allow for the creation of compromised SCMOInstances is, that the CMPI interface allows to create an object path for unknow classes (=classes which do not exist in the schema in the repository).

The SCMOInstance is linked with the SCMOClass. If a key property is set/added on the CMPI object path object, the key property is checked with the SCMOClass and if it is part of this class it is added to the SCMOInstance key property array. Otherwise the key property is added to a chain of user defined key properties.

When a compromised CMPIObjectPath is used to create a CMPIInstance, the class referenced in the object path must exist in the repository. So this will generally fail unless the classname and/or namespace have been updated after creation of the CMPIObjectPath to match an existing class in the repository.
While it is possible to add key properties to a CMPIObjectPath, which are not defined in the respective CIM class to satisfy the CMPI specification, these stale key properties will be stripped of when creating a CMPIInstance from this CMPIObjectPath.

In the previous implementation (2.9 and before), any key binding could be added but invalid key bindings were later removed by the normalizer or when returning an instance.
With this design the semantic is not changed, only the removal of invalid keys occurs earlier in the processing.

SCMBClass

The SCMBClass contains all static data which is valid across instances. A SCMBClass can be created from a CIMClass. This assumes that the CIMClass contains all Properties, Qualifiers and KeyBindings of the requested class, including propagated items.

```
struct SCMBClass_Main
{
    // The SCMB management header
    SCMBMgmt_Header     header;
    // The reference counter for this class
    AtomicInt       refCount;
    // Object flags
    struct{
      unsigned isEmpty :1;
    }flags;

    // SuperClassName
    SCMBDataPtr     superClassName;
    // Relative pointer to classname
    SCMBDataPtr     className;
    // Relative pointer to namespace
    SCMBDataPtr     nameSpace;
    // The key properties of this class are identified
    // by a SCMBKeyPropertyMask
    SCMBDataPtr     keyPropertyMask;
    // A list of index to the key properties in the property set node
array.
    SCMBDataPtr     keyIndexList;
    // Keybinding orderd set
    SCMBKeyBindingSet_Header keyBindingSet;
    // A set containing the class properties.
    SCMBClassPropertySet_Header    propertySet;
    // Relative pointer to SCMBQualifierArray
    Uint32          numberOfQualifiers;
    SCMBDataPtr     qualifierArray;
};
```

The SCMBClass flag `flags.isEmpty` indicates if a SCMBClass consists only of a class and/or name space name.
This can be the case if a interim SCMOClass has to be constructed.
If such class is used to create a SCMOInstance, the instance is flagged as compromised.

The indexes of key properties are stored in an array of Uint32 at the `keyIndexList` .
The size of the array is the number of properties in the class.

Also the bit mask `keyPropertyMask` of the key properties is build at reading the CIMClass to have a fast look-up if a property is a key property.
The mask is organized as Uint64 array.
The array size of the key property mask is (numberOfProperties / 64).
The index of a key property in the property set is equal to the index with in the key property mask.

> If a bit is set (1) then it is a key property.
> If a bit is not set (0) then it is not a key property.

```
typedef Uint64      SCMBKeyPropertyMask[];
```

`keyIndexList` and `keyPropertyMask` are uses to construct the property filter of the SCMBInstance.

Class Properties

The class properties are organized as SCMO ordered set.

```
#define PEGASUS_PROPERTY_SCMB_HASHSIZE 64

struct SCMBClassPropertySet_Header
{
    // Number of property nodes in the nodeArray
    Uint32          numberOfProperties;
    // Used for name based lookups based on the name tags.
    // A tag is generated by entangling the bit values of a first
    // and last letter of a CIMName.
```

```
        // PEGASUS_PROPERTY_SCMO_HASHSIZE is the hash size.
        // The index of the hashTable is calculated doing a remainder
    operator
        // with the name tag and hash size PEGASUS_PROPERTY_SCMB_HASHSIZE
        // (NameTag % PEGASUS_PROPERTY_SCMO_HASHSIZE)
        // The hashTable contains the index of the SCMBClassPropertyNode
        // in the nodeArray.
        Uint32  hashTable[PEGASUS_PROPERTY_SCMB_HASHSIZE];
        // Relative pointer to the ClassPropertyNodeArray;
        SCMBDataPtr     nodeArray;
    };

    struct SCMBClassPropertyNode
    {
        // Is there a next node in the hash chain?
        Sint32          hasNext;
        // Array index of next property in hash chain.
        Uint32          nextNode;
        // The class property
        SCMBClassProperty theProperty;
    };
```

The flags of the class property indicate:

1. If a property was propagated. (propagated=1)
2. Is a key property. (isKey=1)

```
    struct SCMBClassProperty
    {
        // Relative pointer to property name
        SCMBDataPtr     name;
        Uint32          nameHashTag;
        // Flags
        struct{
            unsigned propagated:1;
            unsigned isKey:1;
        } flags;
        // Relative pointer to the origin class name
        SCMBDataPtr     originClassName;
        // Relative pointer to the reference class name
        SCMBDataPtr     refClassName;
        // Contains the default value if specified
        SCMBValue       defaultValue;
        // Number of qualifiers in the array
        Uint32          numberOfQualifiers;
        // Relative pointer to SCMBQualifierArray
        SCMBDataPtr     qualifierArray;
    };
```

Class Key Bindings

The SCMO class contains the ordered set for addressing the key bindings by name.
The class is storing the static data of the key bindings.
SCMO key bindings are stored with the CIMType of the property instead the CIMKeyBinding type

```
    #define PEGASUS_KEYBINDIG_SCMB_HASHSIZE 32

    struct SCMBKeyBindingNode
    {
        // Is there a next node in the hash chain?
        Sint32          hasNext;
        // Array index of next property in hash chain.
        Uint32          nextNode;
        // Relativer pointer to the key property name.
        SCMBDataPtr     name;
        Uint32          nameHashTag;
        // The type of the key binding.
        CIMType         type;
    };

    struct SCMBKeyBindingSet_Header
    {

        // Number of keybindings in the keyBindingNodeArray.
        Uint32          number;
        // Used for name based lookups based on the name tags
        // A tag is generated by entangling the bit values of a first
        // and last letter of a CIMName.
        // PEGASUS_KEYBINDIG_SCMB_HASHSIZE is the hash size.
        // The index of the hashTable is calculated doing a remainder
    operator
        // with the name tag and hash size PEGASUS_KEYBINDIG_SCMB_HASHSIZE
        // (NameTag % PEGASUS_KEYBINDIG_SCMB_HASHSIZE)
        // The hashTable contains the index of the SCMBKeyBindingNode
        // in the keyBindingNodeArray.
        Uint32          hashTable[PEGASUS_KEYBINDIG_SCMB_HASHSIZE];
        // Relative pointer to an array of SCMBKeyBindingNode.
        SCMBDataPtr     nodeArray;
    };
```

Class Qualifiers

Qualifiers for the class and properties are stored in a fix linear array. The size is determined while traversing the CIMClass.
This structure represents a single qualifier in SCMB. In different to the CIMQualifier the qualifier names are stored as an enumeration of the DMTF qualifier names. This avoids the comparison of char represented qualifiers to find well known qualifier in the CIM Server processing. An exception is an user defined qualifier. The value QUALNAME_USERDEFINED(=0) of this enumeration indicates an user defined qualifier and the qualifier name must be stored additionally.

```
    struct SCMBQualifier
    {
        //Boolean flag
        Sint32              propagated;
        QualifierNameEnum   name;
        // The same value as CIMFlavor.
        Uint32              flavor;
        // if name == QUALNAME_USERDEFINED
        // the relative pointer to the user defined name
        SCMBDataPtr     userDefName;
        // Qualifier Value
        SCMBValue       value;

    };

    typedef SCMBQualifier SCMBQualifierArray[];
```

Qualifiers Names

In SCMB the qualifiers defined by the DMTF (standardized and optional) names are mapped to an enumeration and the corresponding display name can be found in a `const char*` array. The rationale for this design decision is, that almost all class, instance, method, and property definitions are using the DMTF defined qualifiers, and user defined qualifiers are very rare. If at all used, the number is very small (1 to 3).

For the base set of the qualifier names, the DMTF CIM Infrastructure DSP0004 Version 2.5.0a is used.
The definitions provided here are not complete. They are cut to keep the PEP readable.

```
    enum QualifierNameEnum
    {
        QUALNAME_USERDEFINED=0,
        QUALNAME_ABSTRACT,
        ....
        QUALNAME_WEEK,
        QUALNAME_WRITE
    };

    static StrLit _qualifierNameStrLit[] =
    {
        STRLIT(""),
        STRLIT("ABSTRACT"),
        ....
        STRLIT("WEAK"),
        STRLIT("WRITE")
    };
```

SCMBInstance

The SCMBInstance contains the instance specific data and a reference to the corresponding SCMBClass.

If a value is not present, SCMBDataPtr is set to NULL and the value of the class has to be used.

The instance flags are used to indicate if the instance properties are including:

1. propagated properties (deepInheritance=true)
2. qualifiers (includeQualifiers=true)
3. class origins (includeClassOrigin=true)
4. if a filter is applied to the instance (isFiltered=true)
5. if the instance is only the bag to hold a SCMOClass(isClassOnly=true)
6. if instance class and/or name space name was modified.(isCompromised=true)

```
    struct SCMBInstance_Main
    {
        // The SCMB management header
        SCMBMgmt_Header     header;
        // The reference counter for this instance
        AtomicInt           refCount;
        // A absolute pointer/reference to the class SCMB for this instance
        SCMOClass*          theClass;
        // Instance flags
        struct{
          unsigned includeQualifiers  :1;
          unsigned includeClassOrigin :1;
          unsigned isFiltered:1;
          unsigned isClassOnly:1;
          unsigned isCompromised:1;
        }flags;

        // Number of user defined key bindings
        Uint32          numberUserKeyBindings;
        // Relative pointer  SCMBUserKeyBindingElement
        SCMBDataPtr     userKeyBindingElement;
        // Relative pointers to the name space name and class name.
        // Will be initialized by with the values of the linked SCMOClass.
        // If it was overwritten, the new value is stored in the
    SCMOInstance
        // and the the flag isCompromised is set to true.
        SCMBDataPtr     instNameSpace;
        SCMBDataPtr     instClassName;
        // Relative pointer to hostname
        SCMBDataPtr     hostName;
        // Number of key bindings of the instance
        Uint32          numberKeyBindings;
        // Relative pointer to SCMBInstanceKeyBindingArray
        SCMBDataPtr     keyBindingArray;
        // Relative pointer to SCMBPropertyFilter
        SCMBDataPtr     propertyFilter;
        // Relative pointer to SCMBPropertyFilterIndexMap
        SCMBDataPtr     propertyFilterIndexMap;
        // Number of properties of the class.
        Uint32          numberProperties;
        // Number of filter properties of the instance.
        Uint32          filterProperties;
        // Relative pointer to SCMBInstancePropertyArray
        SCMBDataPtr     propertyArray;
    };
```

Instance key bindings

The key binding values are organized in a simple array of `SCMBKeyBindingValue`.

```
    struct SCMBKeyBindingValue
    {
        // Boolean flag, if the key binding was set by the provider.
        Sint32       isSet;
        // The value of the key binding
        SCMBUnion    data;
    };
```

The index of an instance key binding is the same as the class. If a key binding has to be find by name, the class key binding ordered set has to be used to find the right index.

```
    typedef SCMBKeyBindingValue SCMBInstanceKeyBindingArray[];
```

If a key binding to be set is not part of the SCMOClass, the key binding is added to a single linked list of `SCMBUserKeyBindingElement`:

```
    struct SCMBUserKeyBindingElement
    {
        // If not 0, a relative pointer to the next element.
        SCMBDataPtr   nextElement;
        // The cim type
        CIMType       type;
        // Relativer pointer to the key property name.
        SCMBDataPtr   name;
        // The value.
        SCMBKeyBindingValue value;
    };
```

Instance properties

The instance properties are organized in a simple array. The index of an instance property is the same as the class property.
If a property is looked up by name, the class property ordered set is used to find the according index.
The properties of an instance contain only values set by the provider.
If an instance property does not contain a value the default value of the class is used.

```
    typedef SCMBValue   SCMBInstancePropertyArray[];
```

Instance property filter

The property filter is implemented as a bit mask. The index of a property in the instance property array matches the index of the property filter.

If a bit is set (1) then a property is eligible to be set and to be returned.
If a bit is not set (0), the property is filtered out and cannot be set and is not eligible to be returned.

The array size of the property filter is (numberOfProperties / 64)

```
    typedef Uint64      SCMBPropertyFilter[];
```

If a filter is applied to an instance (flag.isFiltered=1) the `SCMBPropertyFilterIndexMap` array is used to ensure that one can iterate straight from 0 to `filterProperties`.

```
    typedef Uint32      SCMBPropertyFilterIndexMap[];
```

The filter is initialized with the values of the SCMBClass `keyIndexList` and `keyPropertyMask` .
That means, if a property is a key property, it cannot be filtered out.

SCMBValue

The value handling in SCMO is similar to the values used in CIMValue. The CIMType definitions are used.

```
    struct SCMBValue
    {
        // The CIMType of the value
        CIMType         valueType;

        struct{
            // If the value not set
            unsigned isNull:1;
            // If value is a type array
            unsigned isArray:1;
            // If value is set by the provider( valid for SCMOInstance )
            unsigned isSet:1;
        } flags;

        // The number of elements if the value is a type array.
        Uint32          valueArraySize;

        SCMBUnion       value;
    };
```

The flag `isSet` is needed to identify the value was set by the provider or it was initially created by the SCMOInstance.

SCMBDateTime

The structure of CIMDateTimeRep has been used:

1. because no transformation has to take place at storing a CIMDateTime data object
2. It can be stored directly in the SCMBUnion. It has the same size.

```
    typedef CIMDateTimeRep SCMBDateTime;
```

SCMBUnion

This SCMBUnion is used to represent the values of properties and qualifiers.
Simple CIMTypes, not larger then 16 bytes, are stored directly in the union. If the CIMType is an Array, String or Reference type the union contains a relative pointer to this value with in the SCMO memory block.

EmbeddedObjects and EmbeddedInstances are not moved into the SCMO memory block. For these data types, the SCMBUnion contains an external reference.

```
    union SCMBUnion
    {
        struct
        {
            union
            {
                Boolean  bin;
                Uint8    u8;
                Sint8    s8;
                Uint16   u16;
                Sint16   s16;
                Uint32   u32;
                Sint32   s32;
                Uint64   u64;
                Sint64   s64;
                Real32   r32;
                Real64   r64;
                Uint16   c16;
            }val;
            // SCMBUnion used in array values.
            // This indicates if the single array member is a Null value.
            // The type of size 64 is used to fill up the whole union.
            Uint64 hasValue;
        }simple;

        SCMBDataPtr arrayValue;
        SCMBDataPtr stringValue;
        SCMBDateTime dateTimeValue;
        // Used for embedded referecnes, instances, and objects
        // as an external references to SCMO_Instances.
        SCMOInstance* extRefPtr;

        // This structure is used to handle an absolute char*
        // including the length ( without traling '\0')
        struct
        {
            Uint64 length;
            char*  pchar;
        }extString;
    };
```

Access Methods

The access methods to the data structures above are described as C++ class definitions.
Only complex methods are described here.
The primitive getter and setter methods are not part of this document and are added as needed.
Internal helper functions are also not described here

SCMOInstance class

```
    class PEGASUS_COMMON_LINKAGE SCMOInstance
    {
    public:

        /**
         * Creating a SCMOInstance using a SCMOClass.
         * @param baseClass A SCMOClass.
         */
        SCMOInstance(SCMOClass& baseClass);

        /**
         * Copy constructor for the SCMO instance, used to implement
    refcounting.
         * @param theSCMOClass The instance for which to create a copy
         * @return
         */
        SCMOInstance(const SCMOInstance& theSCMOInstance );

        /**
         * Assignment operator for the SCMO instance,
         * @param theSCMOClass The right hand value
         **/
        SCMOInstance& operator=(const SCMOInstance& theSCMOInstance);

        /**
         * Destructor is decrementing the refcount. If refcount is zero, the
         * singele chunk memory object is deallocated.
         */
        ~SCMOInstance()

        /**
         * Builds a SCMOInstance based on this SCMOClass.
         * The method arguments determine whether qualifiers are included,
         * the class origin attributes are included,
         * and which properties are included in the new instance.
         * @param baseClass The SCMOClass of this instance.
         * @param includeQualifiers A Boolean indicating whether qualifiers
    in
         * the class definition (and its properties) are to be added to the
         * instance.  The TOINSTANCE flavor is ignored.
         * @param includeClassOrigin A Boolean indicating whether
    ClassOrigin
         * attributes are to be added to the instance.
         * @param propertyList Is an NULL terminated array of char* to
    property
         * names defining the properties that are included in the created
    instance.
         * If the propertyList is NULL, all properties are included to the
    instance.
         * If the propertyList is empty, no properties are added.
         *
         * Note that this function does NOT generate an error if a property
    name
         * is supplied that is NOT in the class;
         * it simply does not add that property to the instance.
         *
         */
        SCMOInstance(
            SCMOClass& baseClass,
            Boolean includeQualifiers,
            Boolean includeClassOrigin,
            const char** propertyList);

        /**
         * Builds a SCMOInstance from the given SCMOClass and copies all
         * CIMInstance data into the new SCMOInstance.
         * @param baseClass The SCMOClass of this instance.
         * @param cimInstance A CIMInstace of the same class.
         * @exception Exception if class name does not match.
         * @exception Exception if a property is not part of class
    definition.
         * @exception Exception if a property does not match the class
    definition.
         */
        SCMOInstance(SCMOClass& baseClass, const CIMInstance& cimInstance);

        /**
         * Builds a SCMOInstance from the given SCMOClass and copies all
         * CIMObjectPath data into the new SCMOInstance.
         * @param baseClass The SCMOClass of this instance.
         * @param cimInstance A CIMObjectpath of the same class.
         * @exception Exception if class name does not match.
         */
        SCMOInstance(SCMOClass& baseClass, const CIMObjectPath& cimObj);

        /**
         * Builds a SCMOInstance from the given CIMInstance copying all
    data.
         * The SCMOClass is retrieved from SCMOClassCache using
         * the class and name space of the CIMInstance.
         * If the SCMOClass was not found, an empty SCMOInstance will be
    returned
         * and the resulting SCMOInstance is compromized.
         * If the CIMInstance does not contain a name space, the optional
    fall back
         * name space is used.
         * @param cimInstance A CIMInstace with class name and name space.
         * @param altNameSpace An alternative name space name.
         * @exception Exception if a property is not part of class
    definition.
         * @exception Exception if a property does not match the class
    definition.
         */
        SCMOInstance(
            const CIMInstance& cimInstance,
            const char* altNameSpace=0,
            Uint64 altNSLen=0);

        /**
         * Builds a SCMOInstance from the given CIMObjectPath copying all
    data.
         * The SCMOClass is retrieved from SCMOClassCache using
         * the class and name space of the CIMObjectPath.
         * If the SCMOClass was not found, an empty SCMOInstance will be
    returned
         * and the resulting SCMOInstance is compromized.
         * If the CIMObjectPath does not contain a name space,
         * the optional fall back name space is used.
         * @param cimObj A CIMObjectpath with name space and name
         * @param altNameSpace An alternative name space name.
         * @
         */
        SCMOInstance(
            const CIMObjectPath& cimObj,
            const char* altNameSpace=0,
            Uint64 altNSLen=0);

        /**
         * Builds a SCMOInstance from the given CIMObject copying all data.
         * The SCMOClass is retrieved from SCMOClassCache using
         * the class and name space of the CIMObject.
         * If the SCMOClass was not found, an empty SCMOInstance will be
    returned
         * and the resulting SCMOInstance is compromized.
         * If the CIMInstance does not contain a name space, the optional
    fall back
         * name space is used.
         * @param cimInstance A CIMInstace with class name and name space.
         * @param altNameSpace An alternative name space name.
         * @exception Exception if a property is not part of class
    definition.
         * @exception Exception if a property does not match the class
    definition.
         */
        SCMOInstance(
            const CIMObject& cimObject,
            const char* altNameSpace=0,
            Uint64 altNSLen=0);



        /**
         * Converts the SCMOInstance into a CIMInstance.
         * It is a deep copy of the SCMOInstance into the CIMInstance.
         * @param cimInstance An empty CIMInstance.
         */
        void getCIMInstance(CIMInstance& cimInstance) const;

        /**
         * Makes a deep copy of the instance.
         * This creates a new copy of the instance.
         * @return A new copy of the SCMOInstance object.
         */
        SCMOInstance clone() const;

        /**
         * Returns the number of properties of the instance.
         * @param Number of properties
         */
        Uint32 getPropertyCount() const;

        /**
         * Gets the property name, type, and value addressed by a positional
    index.
         * The property name and value has to be copied by the caller !
         * @param pos The positional index of the property
         * @param pname Returns the property name as '\0' terminated string.
         *              Has to be copied by caller.
         *              It is set to NULL if rc != SCMO_OK.
         * @param pvalue Returns a pointer to the value of property.
         *               The value is stored in a SCMBUnion
         *                and has to be copied by the caller !
         *               It returns NULL if rc != SCMO_OK.
         *
         *               If the value is an array, the
         *               value array is stored in continuous memory.
         *               e.g. (SCMBUnion*)value[0 to size-1]
         *
         *               If the value is type of CIMTYPE_STRING,
         *               the string is referenced by the structure
         *               SCMBUnion.extString:
         *                       pchar contains the absolut pointer to the
    string
         *                       length contains the size of the string
         *                              without trailing '\0'.
         *               Only for strings the caller has to free pvalue !
         * @param type Returns the CIMType of the property
         *             It is invalid if rc == SCMO_INDEX_OUT_OF_BOUND.
         * @param isArray Returns if the value is an array.
         *             It is invalid if rc == SCMO_INDEX_OUT_OF_BOUND.
         * @param size Returns the size of the array.
         *             If it is not an array, 0 is returned.
         *             It is invalid if rc == SCMO_INDEX_OUT_OF_BOUND.
         *
         * @return     SCMO_OK
         *             SCMO_NULL_VALUE : The value is a null value.
         *             SCMO_INDEX_OUT_OF_BOUND : Given index not found
         *
```

```
         */
        SCMO_RC getPropertyAt(
            Uint32 pos,
            const char** pname,
            CIMType& type,
            const SCMBUnion** pvalue,
            Boolean& isArray,
            Uint32& size ) const;

        /**
         * Gets the type and value of the named property.
         * The value has to be copied by the caller !
         * @param name The property name
         * @param pvalue Returns a pointer to the value of property.
         *                 The value is stored in a SCMBUnion
         *                  and has to be copied by the caller !
         *                 It returns NULL if rc != SCMO_OK.
         *
         *                 If the value is an array, the
         *                 value array is stored in continuous memory.
         *                 e.g. (SCMBUnion*)value[0 to size-1]
         *
         *                 If the value is type of CIMTYPE_STRING,
         *                 the string is referenced by the structure
         *                 SCMBUnion.extString:
         *                         pchar contains the absolut pointer to the
    string
         *                         length contains the size of the string
         *                                without trailing '\0'.
         *                 Only for strings the caller has to free pvalue !
         * @param type Returns the CIMType of the property
         *               It is invalid if rc == SCMO_NOT_FOUND.
         * @param isArray Returns if the value is an array.
         *               It is invalid if rc == SCMO_NOT_FOUND.
         * @param size Returns the size of the array.
         *               If it is not an array, 0 is returned.
         *               It is invalid if rc == SCMO_NOT_FOUND.
         *
         * @return     SCMO_OK
         *               SCMO_NULL_VALUE : The value is a null value.
         *               SCMO_NOT_FOUND : Given property name not found.
         */
        SCMO_RC getProperty(
            const char* name,
            CIMType& type,
            const SCMBUnion** pvalue,
            Boolean& isArray,
            Uint32& size ) const;

        /**
         * Set/replace a property in the instance.
         * If the class origin is specified, it is honored at identifying
         * the property within the instance.
         * Note: Only properties which are already part of the
    instance/class can
         * be set/replaced.
         * @param name The name of the property to be set.
         * @param type The CIMType of the property
         * @param value A pointer to the value to be set at the named
    property.
         *                The value has to be in a SCMBUnion.
         *           The value is copied into the instance
         *           If the value == NULL, a null value is assumed.
         *                If the value is an array, the value array has to be
```

```
     *               stored in continuous memory.
     *               e.g. (SCMBUnion*)value[0 to size-1]
     *
     *               To store an array of size 0, The value pointer has
to
     *               not NULL ( value != NULL ) but the size has to be 0
     *               (size == 0).
     *
     *               If the value is type of CIMTYPE_STRING,
     *               the string is referenced by the structure
     *               SCMBUnion.extString:
     *                        pchar contains the absolut pointer to the
string
     *                        length contains the size of the string
     *                               without trailing '\0'.
     * @param isArray Indicate that the value is an array. Default
false.
     * @param size Returns the size of the array. If not an array this
     *         this parameter is ignorer. Default 0.
     * @param origin The class originality of the property.
     *                If NULL, then it is ignorred. Default NULL.
     * @return     SCMO_OK
     *              SCMO_NOT_SAME_ORIGIN : The property name was found,
but
     *                                   the origin was not the same.
     *              SCMO_NOT_FOUND : Given property name not found.
     *              SCMO_WRONG_TYPE : Named property has the wrong type.
     *              SCMO_NOT_AN_ARRAY : Named property is not an array.
     *              SCMO_IS_AN_ARRAY  : Named property is an array.
     */
    SCMO_RC setPropertyWithOrigin(
        const char* name,
        CIMType type,
        const SCMBUnion* value,
        Boolean isArray=false,
        Uint32 size = 0,
        const char* origin = NULL);

    /**
     * Rebuild of the key bindings from the property values
     * if no or incomplete key properties are set on the instance.
     * @exception NoSuchProperty
     */
    void buildKeyBindingsFromProperties();

    /**
     * Set/replace a property filter on an instance.
     * The filter is a white list of property names.
     * A property part of the list can be accessed by name or index and
     * is eligible to be returned to requester.
     * Key properties can not be filtered. They are always a part of the
     * instance. If a key property is not part of the property list,
     * it will not be filtered out.
     * @param propertyList Is an NULL terminated array of char* to
     * property names
     */
    void setPropertyFilter(const char **propertyList);

    /**
     * Gets the hash index for the named property. Filtering is ignored.
     * @param theName The property name
     * @param pos Returns the hash index.
     * @return     SCMO_OK
     *              SCMO_INVALID_PARAMETER: name was a NULL pointer.
     *              SCMO_NOT_FOUND : Given property name not found.
     */
    SCMO_RC getPropertyNodeIndex(const char* name, Uint32& pos) const;

    /**
     * Set/replace a property in the instance at node index.
     * Note: If node is filtered, the property is not set but the return
value
     * is still SCMO_OK.
     * @param index The node index.
     * @param type The CIMType of the property
     * @param pInVal A pointer to the value to be set at the named
property.
     *                The value has to be in a SCMBUnion.
     *          The value is copied into the instance
     *                If the value == NULL, a null value is assumed.
     *                If the value is an array, the value array has to be
     *                stored in continuous memory.
     *                e.g. (SCMBUnion*)value[0 to size-1]
     *
     *               To store an array of size 0, The value pointer has
to
     *                not NULL ( value != NULL ) but the size has to be 0
     *                 (size == 0).
     *
     *                If the value is type of CIMTYPE_STRING,
     *                the string is referenced by the structure
     *                SCMBUnion.extString:
     *                         pchar contains the absolut pointer to the
string
     *                        length contains the size of the string
     *                                 without trailing '\0'.
     * @param isArray Indicate that the value is an array. Default
false.
     * @param size The size of the array. If not an array this
     *          this parameter is ignorer. Default 0.
     * @return     SCMO_OK
     *              SCMO_INDEX_OUT_OF_BOUND : Given index not found
     *              SCMO_WRONG_TYPE : The property at given node index
     *                                has the wrong type.
     *              SCMO_NOT_AN_ARRAY : The property at given node index
     *                                  is not an array.
     *              SCMO_IS_AN_ARRAY  : The property at given node index
     *                                  is an array.
     */
    SCMO_RC setPropertyWithNodeIndex(
        Uint32 node,
        CIMType type,
        const SCMBUnion* pInVal,
        Boolean isArray=false,
        Uint32 size = 0);

    /**
     * Set/replace the named key binding using binary data
     * @param name The key binding name.
     * @param type The type as CIMType.
     * @param keyvalue A pointer to the binary key value.
     *          The value is copied into the instance
     *          If the value == NULL, a null value is assumed.
     * @param keyvalue A pointer to the value to be set at the key
binding,
     *                The keyvalue has to be in a SCMBUnion.
     *                The keyvalue is copied into the instance.
     *                If the keyvalue == NULL, a null value is assumed.
     *
     *               If the keyvalue is type of CIMTYPE_STRING,
     *               the string is referenced by the structure
     *               SCMBUnion.extString:
     *                        pchar contains the absolut pointer to the
string
     *                       length contains the size of the string
     *                                without trailing '\0'.
     * @return     SCMO_OK
     *             SCMO_INVALID_PARAMETER : Given name or pvalue
     *                                      is a NULL pointer.
     *             SCMO_TYPE_MISSMATCH : Given type does not
     *                                   match to key binding type
     *             SCMO_NOT_FOUND : Given property name not found.
     */
    SCMO_RC setKeyBinding(
        const char* name,
        CIMType type,
        const SCMBUnion* keyvalue);

    /**
     * Set/replace the key binding at node
     * @param node The node index of the key.
     * @param type The type as CIMType.
     * @param keyvalue A pointer to the value to be set at the key
binding,
     *               The keyvalue has to be in a SCMBUnion.
     *               The keyvalue is copied into the instance.
     *               If the keyvalue == NULL, a null value is assumed.
     *
     *               If the keyvalue is type of CIMTYPE_STRING,
     *               the string is referenced by the structure
     *               SCMBUnion.extString:
     *                        pchar contains the absolut pointer to the
string
     *                       length contains the size of the string
     *                                without trailing '\0'.
     * @return     SCMO_OK
     *             SCMO_INVALID_PARAMETER : Given pvalue is a NULL
pointer.
     *             SCMO_TYPE_MISSMATCH : Given type does not
     *                                   match to key binding type
     *             SCMO_INDEX_OUT_OF_BOUND : Given index is our of
range.
     */
    SCMO_RC setKeyBindingAt(
        Uint32 node,
        CIMType type,
        const SCMBUnion* keyvalue);

    /**
     * Clears all key bindings in an instance.
     * Warning: External references are freed but only the internal
     * control structures are resetted. No memory is freed and at
setting
     * new key bindings the instance will grow in memory usage.
     **/
    void clearKeyBindings();

    /**
     * Gets the key binding count.
     * @return the number of key bindings set.
     */
    Uint32 getKeyBindingCount() const;
    /**
     * Get the indexed key binding.
     * @parm idx The key bining index
     * @parm pname Returns the name.
     *             Has to be copied by caller.
     *             It is invalid if rc == SCMO_INDEX_OUT_OF_BOUND.
     * @param type Returns the type as CIMType.
     *             It is invalid if rc == SCMO_INDEX_OUT_OF_BOUND.
     * @param keyvalue A pointer to the binary key value.
     *             Has to be copied by caller.
     *             It is only valid if rc == SCMO_OK.
     * @return     SCMO_OK
     *             SCMO_NULL_VALUE : The key binding is not set.
     *             SCMO_INDEX_OUT_OF_BOUND : Given index not found
     *
     */
    SCMO_RC getKeyBindingAt(
        Uint32 idx,
        const char** pname,
        CIMType& type,
        const SCMBUnion** keyvalue) const;

    /**
     * Get the named key binding.
     * @parm name The name of the key binding.
     * @param type Returns the type as CIMType.
     *             It is invalid if rc == SCMO_INDEX_OUT_OF_BOUND.
     * @param keyvalue Returns a pointer to the value of keybinding.
     *               The value is stored in a SCMBUnion
     *                and has to be copied by the caller !
     *               It returns NULL if rc != SCMO_OK.
     *
     *               If the value is type of CIMTYPE_STRING,
     *               the string is referenced by the structure
     *               SCMBUnion.extString:
     *                       pchar contains the absolut pointer to the
string
     *                       length contains the size of the string
     *                              without trailing '\0'.
     *               Only for strings the caller has to free pvalue !
     * @param keyvalue A pointer to the binary key value.
     *             Has to be copied by caller.
     *             It is only valid if rc == SCMO_OK.
     * @return     SCMO_OK
     *             SCMO_NULL_VALUE : The key binding is not set.
     *             SCMO_NOT_FOUND : Given property name not found.
     */
    SCMO_RC getKeyBinding(
        const char* name,
        CIMType& ptype,
        const SCMBUnion** keyvalue) const;


    /**
     * Determines whether the object has been initialized.
     * @return True if the object has not been initialized, false
otherwise.
     */
    Boolean isUninitialized( ) const {return (inst.base == NULL); };

private:

    /**
```

```
         * A SCMOInstance can only be created by a SCMOClass
         */
        SCMOInstance();

        void _initSCMOInstance(Boolean inclQual,Boolean inclOrigin);
        union{
            // To access the instance main structure
            SCMBInstance_Main *hdr;
            // To access the memory management header
            SCMBMgmt_Header     *mem;
            // Generic access pointer
            char *base;
        }inst;

        friend class SCMOClass;
    };
```

The method setPropertyFilter() accepting a property list and a key property list is not implemented by the SCMOInstance. The rational for this design decision is, that a key property is not filtered out by the implementation of the setPropertyFiler() method and therefore the key list can be ignorred.

At converting a CIMInstance to a SCMOInstance, it is assumed that the CIMInstance has no filter applied on. The rational for this is, that the request contains the property filter list and the for CIMInstances the filter is applied at the end point (like XMLWriter). All properties of the CIMInstance are set on SCMOInstance and no filer is applied on the SCMOInstance.

SCMOClass class

```
    class PEGASUS_COMMON_LINKAGE SCMOClass
    {
    public:

        /**
         * Constructs a SCMOClass out of a CIMClass.
         * @param theCIMClass The source the SCMOClass is constructed off.
         * @param nameSpaceName The namespace for the class, optional.
         * @return
         */
        SCMOClass(const CIMClass& theCIMClass, const char* altNameSpace=0 );

        /**
         * Copy constructor for the SCMO class, used to implement
    refcounting.
         * @param theSCMOClass The class for which to create a copy
         * @return
         */
        SCMOClass(const SCMOClass& theSCMOClass );

        /**
         * Assignment operator for the SCMO class,
         * @param theSCMOClass The right hand value
         **/
        SCMOClass& operator=(const SCMOClass& theSCMOClass)

        /**
         * Destructor is decrementing the refcount. If refcount is zero, the
         * singele chunk memory object is deallocated.
         */
        ~SCMOClass();

        /**
         * Converts the SCMOClass into a CIMClass.
         * It is a deep copy of the SCMOClass into the CIMClass.
         * @param cimClass An empty CIMClass.
         */
        void getCIMClass(CIMClass& cimClass) const;

        /**
         * Gets the key property names as a string array
         * @return An Array of String objects containing the names of the
    key
         * properties.
         */
        void getKeyNamesAsString(Array<String>& keyNames) const;

        /**
         * Determines whether the object has been initialized.
         * @return True if the object has not been initialized, false
    otherwise.
         */
        Boolean isUninitialized( ) const {return (cls.base == NULL); };

    private:

        /**
         * Constructs an uninitialized SCMOClass object.
         */
        SCMOClass();

        union{
            // To access the class main structure
            SCMBClass_Main *hdr;
            // To access the memory management header
            SCMBMgmt_Header  *mem;
            // Generic access pointer
            char *base;
        }cls;

    };
```

Usage of SCMO for the CMPIProviderManager implementation

To implement SCMO for CMPI basically the implementation of CMPIInstance and CMPIObjectPath have to be reworked to make use of the new SCMO data structures described above. Both CMPI types are implemented by the SCMOInstance class. Other CMPI data types are only redefined where necessary to accelerate the implementation of SCMO Instances.
This affects the CMPIBrokerEnc, CMPIInstance and CMPIObjectPath function tables (only listing functions that have significant changes):

CMPIBrokerEncFT

- function newObjectPath:
  Retrieves the SCMOClass object that matches the given namespace and classname from the CMPI Class Cache and derives a SCMOInstance object from this SCMOClass. The SCMOInstance is returned as CMPIObjectPath* encapsulated in a CMPIObject.
  Creation of stale CMPIObjectPaths for a non-existent or undefined namespace/classname combination is prohibited to prevent inconsistent CMPIObjectPaths and CMPIInstances.

- function newInstance:
  Creates new SCMOInstance object pointing to the same SCMB data as the SCMOInstance representing the passed in CMPIObjectPath. For this a special copy constructor of SCMOInstance is The new SCMOInstance is returned as CMPIInstance* encapsulated in a CMPIObject.

CMPIInstanceFT

- function release:

Removes the SCMOInstance from the thread context and deletes it. This decreases the refcount of the associated SCMB object and eventually deletes the SCMB when the refcount becomes zero.

- function clone:
  Returns a 1:1 copy from the SCMOInstance, including a new copy of the associated SCMB data object with a refcount of 1.
  The new SCMOInstance is returned as CMPIInstance* encapsulated in a CMPIObject.

- function getProperty/getPropertyAt:
  Returns a copy of the named property in the form of a memory managed SCMOProperty encapsulated as CMPIData

- function setProperty/setPropertyWithOrigin:
  Updates the named property with the given value on the instance. The CMPIValue is used unchanged over the existing implementation except for the complex types of CMPIInstance and CMPIObjectPath or arrays of these types.

- function getObjectPath:
  Simply returns a pointer to this SCMOInstance encapsulated as CMPIObject, casted to CMPIObjectPath.

- function setPropertyFilter:
  Sets the propertyFilter on the SCMOInstance using the setPropertyFilter() method. The key properties argument is ignored, since the key information is already obtained from the SCMOClass.

- function setObjectPath:
  Updates the ObjectPath part of the SCMOInstance with the classname, namespace and keys from the given CMPIObjectPath.
  If the namespace and classname do not match the current values defined for the CMPIInstance, the function will fail with because it would otherwise leave the CMPIInstance in an inconsistent state.

CMPIObjectPathFT

- function release:
  Removes the SCMOInstance behind the CMPIObjectPath from the thread context and deletes it. This decreases the refcount of the associated SCMB object and eventually deletes the SCMB when the refcount becomes zero.

- function clone:
  Returns a 1:1 copy from the SCMOInstance, including a new copy of the associated SCMB data object with a refcount of 1.
  The new SCMOInstance is returned as CMPIObjectPath* encapsulated in a CMPIObject.

- function setNameSpace/setClassName:
  Updates the namespace/classname combination on the CMPIObjectPath (SCMOInstance). To prevent inconsistencies, changed new values will result in a lookup of the new SCMOClass and a validation of the SCMOInstance. Invalid namespace/classname combinations (class definition not available in the repository) will result in an error.

- function addKey:
  Updates the value for the key in the CMPIObjectPath (SCMOInstance) if the key name and type are valid for the class for which the CMPIObjectPath was created.

CMPI Class Cache updates

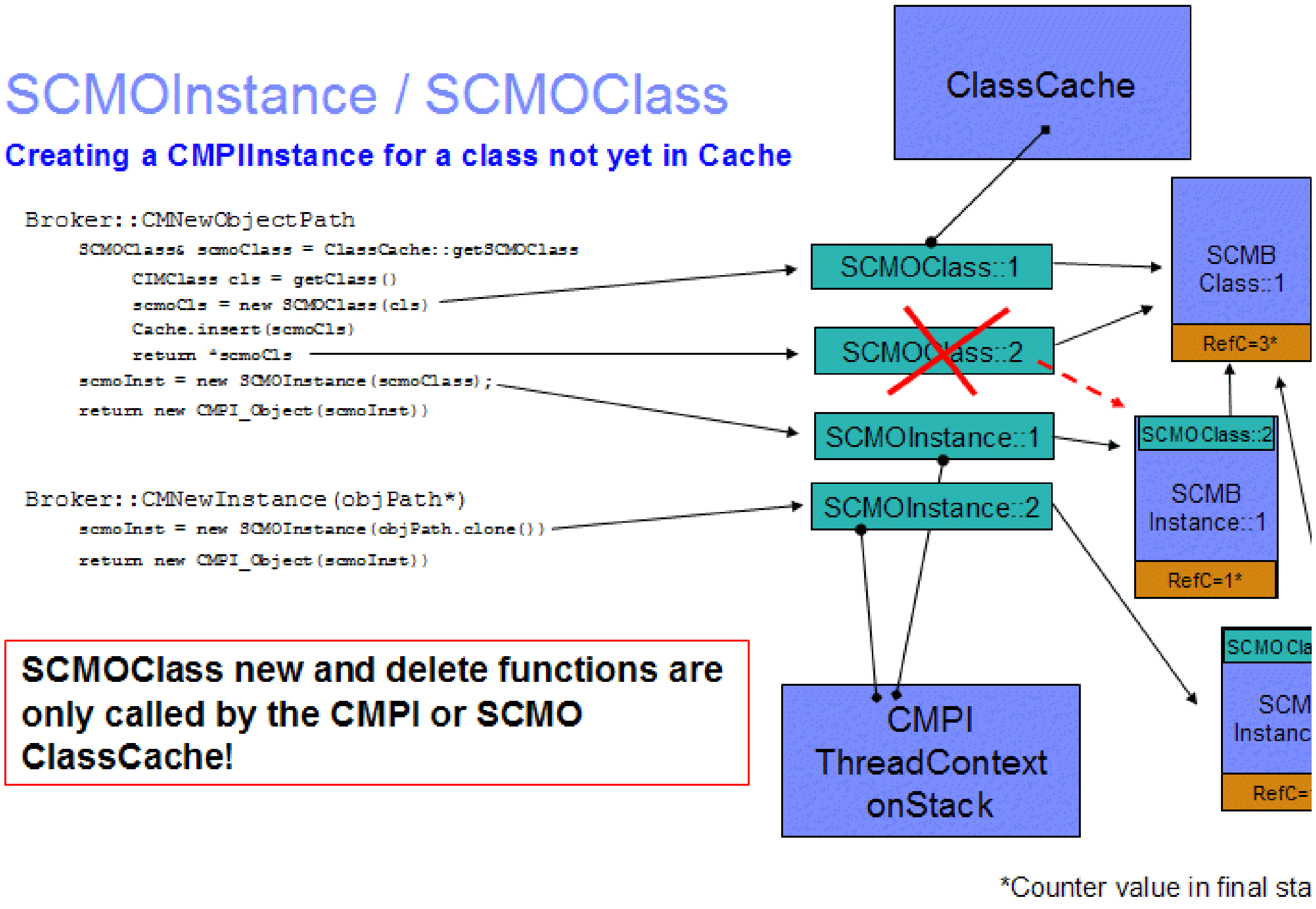
The CMPI class cache is changed to cache SCMOClass objects rather than CIMClass objects.
For this a new method getSCMOClass is added which caches SCMO Classes instead of CIM Classes. It returns a reference to an SCMOClass from the class cache.
All SCMOInstances hold references to the SCMOClass stored in the class cache. The SCMOClass object itself is ref counted, so when the cache is updated with a new copy of a SCMOClass, the SCMOInstances still refer to their original copy, which will be deleted when the last SCMOInstance that references this SCMOClass object is deleted.
The key for SCMO classes is still namespace+classname.
When no entry exists method getSCMOClass () tries to obtain the SCMOClass from the SCMOClassCache, stores the original copy in the cache (refcount++) and returns a new copy to the caller (refcount++).

SCMO and memory management

The SCMO objects created via any of the mbEncNewXXX functions in CMPI_BrokerEnc are memory managed like in the previous implementation by using the CMPIObject wrapper class, which puts an anchor on the thread context for releasing the memory when going out of scope.

For this new CMPIObject constructors for SCMOInstance will be implemented for CMPIInstance and CMPIObjectPath.

Example of SCMOClass and SCMOInstance allocation



SCMOInstance / SCMOClass

Creating a CMPIInstance for a class not yet in Cache

```
Broker::CMNewObjectPath
    SCMOClass& scmoClass = ClassCache::getSCMOClass
        CIMClass cls = getClass()
        scmoCls = new SCMOClass(cls)
        Cache.insert(scmoCls)
        return *scmoCls
    scmoInst = new SCMOInstance(scmoClass);
    return new CMPI_Object(scmoInst))

Broker::CMNewInstance(objPath*)
    scmoInst = new SCMOInstance(objPath.clone())
    return new CMPI_Object(scmoInst))
```

SCMOClass new and delete functions are only called by the CMPI or SCMO ClassCache!

CMPIProviderManager updates for SCMO

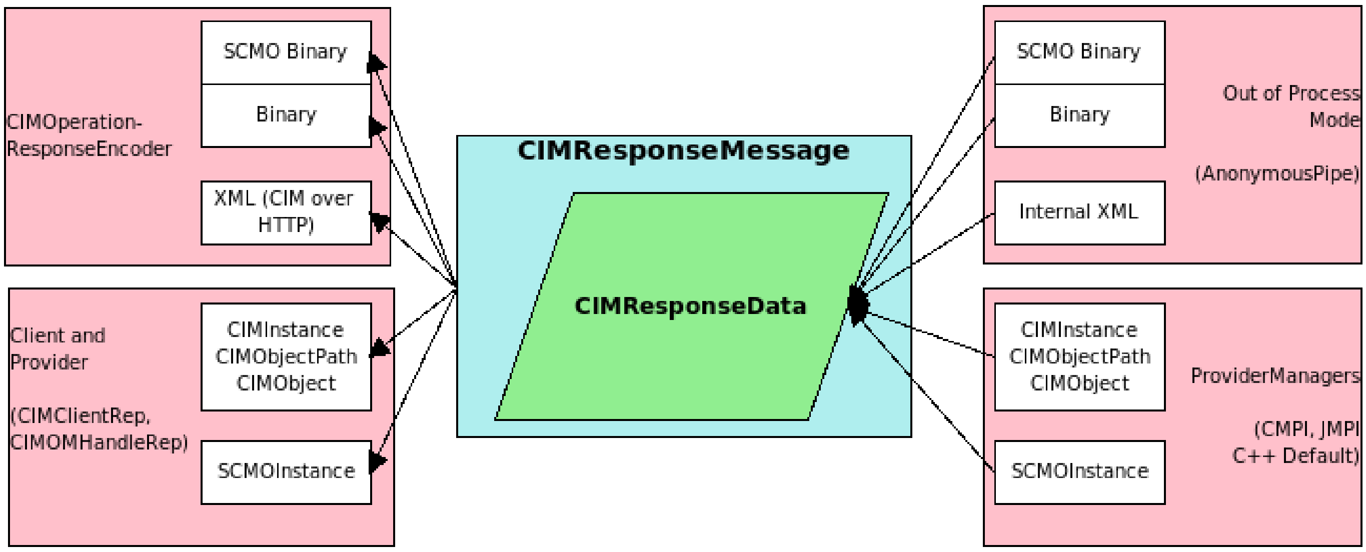
Due to the new implementation of CMPIObjectPath and CMPIInstance, the CMPI Provider Manager needs to convert objects of type CIMObjectPath and CIMInstance into objects of type SCMOInstance when passing them to a CMPI provider. This affects most of the CMPIxxxMIFT functions as well as the CMPIBrokerFT functions which return object paths or instances or enumerations of those.
For this according constructors of SCMOInstance are provided which can do this conversion and the CMPIProviderManager will use these functions to convert CIMxxx type objects into SCMOInstance objects before passing them to a CMPI Provider.
The provider upcalls implemented by the CMPIBroker will use the CIMResponseData class to convert incomming CIMxxx objects into SCMOInstances where necessary. (see also Upcall for CMPI provider manager and providers)

Data transport and transformation of SCMO objects through the CIM Server

With the introduction of the new SCMO data representation it is necessary to transport and convert the new representation into all other possible data representations. Today OpenPegasus does support the representation of data internally as Binary (OOP optimization), InternalXml (used to avoid transformation in a simple Out-of-Process) and C++ default objects. All those formats are used primarily in responses, stored in CIMResponseMessages. With the introduction of the binary representation for some of the CIM operations (GetInstance, EnumerateInstances, ExecQuery, Associators) the classes CIMInstanceResponseData, CIMInstancesResponseData, CIMObjectsResponseData have been introduced to carry the data representations for their respective response messages.
On further extension of the binary representation to other CIM operations the number of the CIMxxxResponseData classes would have further grown with some considerable code replication necessary.
To avoid that code replication and to keep complexity simple in other code parts using response data the single class CIMResponseData is used.

CIMResponseData

This class serves the purpose to not only hold all known representations of response data, but also is capable to transform each format into another one. For efficency, data is only transformed on request. To allow binary or xml generation of output send to a client without a transformation of the SCMO format to C++ default or vice versa, CIMResponseData also holds the two functions encodeXmlResponse() and encodeBinaryResponse().
The following picture shows the different formats and transformations that are supported by CIMResponseData.



The class CIMResponseData can store all four different data representations: CIM (aka C++ default), SCMO, binary and InternalXML at the same time. The bit flags field _encoding holds the following enumeration values: enum ResponseDataEncoding {RESP_ENC_CIM = 1, RESP_ENC_BINARY = 2, RESP_ENC_XML = 4, RESP_ENC_SCMO = 8}; SCMO Binary and Binary are both represented as binary encoding, an explanation to the "How" will follow later on.

The different provider managers can create SCMO or C++ default objects and insert them into CIMResponseData, just as well can binary or internal Xml be received from an Out of Process agent and placed into CIMResponseData. On the consumer side, either the CIMOperationResponseEncoder can retrieve a binary or Xml representation of the response objects by using the aforementioned functions encodeXmlResponse() and encodeBinaryResponse(), or Clients and Provider can consume what's in CIMResponseData as C++ default objects or SCMO objects(instances).

The capability of CIMResponseData to transform every data representation into each other allows a much simpler approach for transporting data through the CIM server, because it relieves the consumer from needing knowledge about the actual data representation.

As the type of data in a CIMResponseData object is specifc to a CIM request type, CIMResponseData also stores the expected _dataType using the following enumeration:
enum ResponseDataContent { RESP_INSTNAMES = 1, RESP_INSTANCES = 2, RESP_INSTANCE = 3, RESP_OBJECTS = 4, RESP_OBJECTPATHS =5 };
The _dataType is used to protect CIMResponseData from accepting data types invalid for the CIM response.

Transport of response data through the CIM server In-Process

The following picture shows the for this design relevant parts of the data flow for the simplest possible case, an In-Process running CIM Server and providers not making upcalls. On the next pictures the flow will be extended by upcalls and the Out-of-Process mode.

No upcalls

1. First a request is received from an user on a client.
2. After decoding and authorizing the request, the generated CIMRequestMessage which holds C++ object data is sent to the CIMRequestDispatcher
3. The CIMOperationRequestDispatcher reads the CIMRequestMessage, determines the responsible providers and provider managers and dispatches the request message.
4. The request message is handed by the ProviderManagers to their responsible providers.
5. The providers deliver instances, objects, instance names or object paths by using the SimpleResponseHandlers which are extended to also hold SCMOInstances.
6. Using the transfer function of the OperationResponseHandlers a provider initiates that the C++ default objects and SCMOInstances are taken from the SimpleResponseHandlers and placed in the response message in CIMResponseData.
7. The operation aggregator takes the response messages and combines them into a single response. As CIMResponseData can hold all four data representations at once, it is not necessary to do a transformation of the data from one format into another.
8. The CIMOperationResponseEncoder uses the functions CIMResponseData::encodeBinaryResponse() and CIMResponseData::encodeXmlResponse() to get the clients requested encoding of the response data.

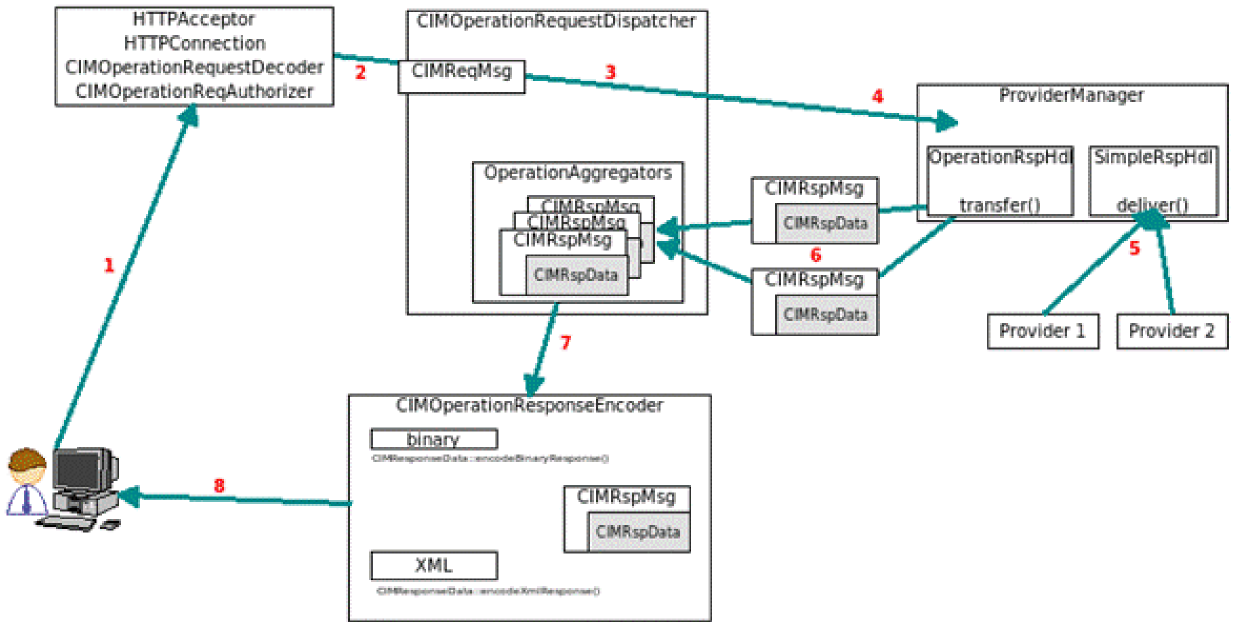

HTTPAcceptor
HTTPConnection
CIMOperationRequestDecoder
CIMOperationReqAuthorizer
CIMOperationRequestDispatcher
CIMReqMsg
OperationAggregators
CIMRspMsg
CIMRspData
ProviderManager
OperationRspHdl
transfer()
SimpleRspHdl
deliver()
Provider 1
Provider 2
CIMOperationResponseEncoder
binary
CIMResponseData::encodeBinaryResponse()
XML
CIMResponseData::encodeXmlResponse()
1
2
3
4
5
6
7
8

Figure 1 with no Upcall

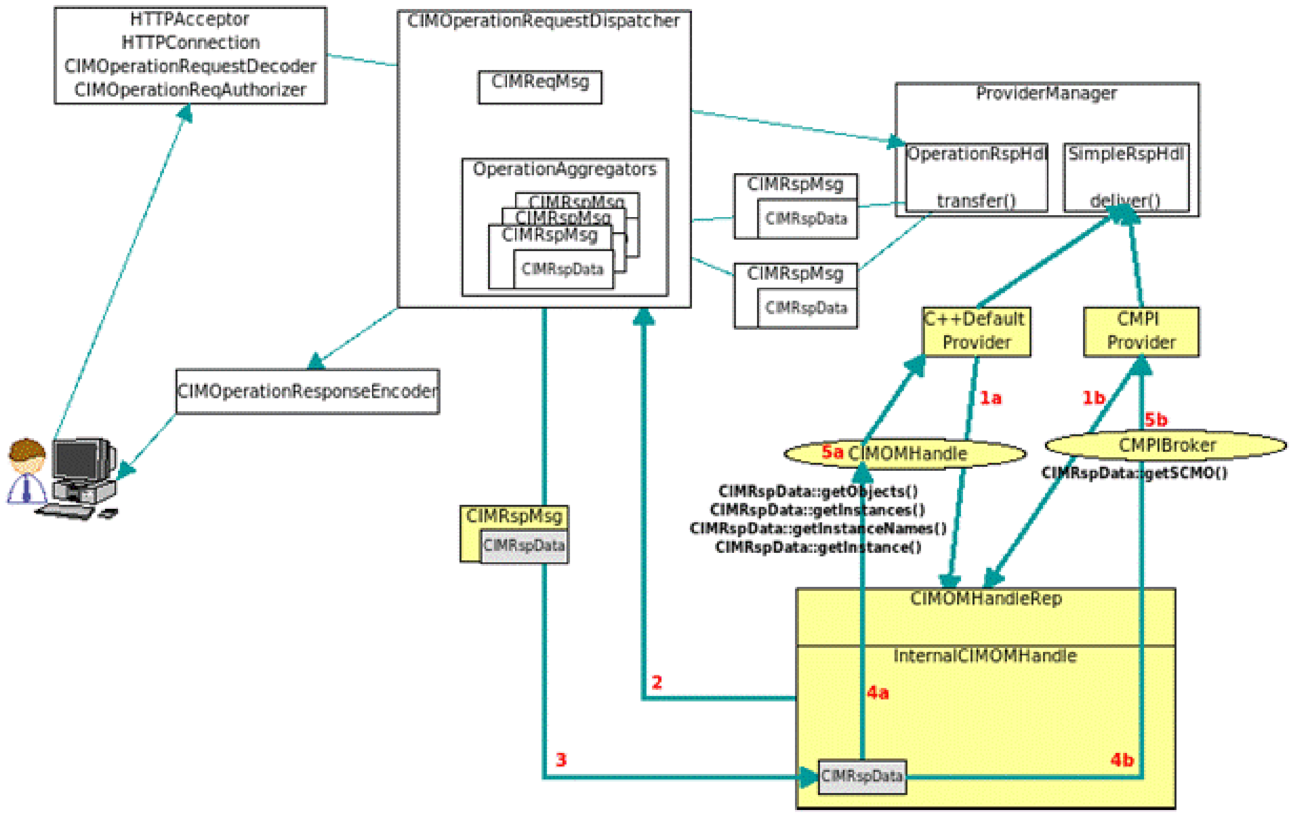


Figure 2 with Upcall

An upcall

To allow for an easier understanding the same picutre as used for the upcall-less scenario before is used here, extended by what's needed for an upcall. As the handling is slightly different between the CMPI provider manager and providers and the C++ Default provider manager and providers, the flow is split in step 1a-5a and 1b-5b.

Upcall for C++ default provider manager and providers

1a. The provider does make an upcall through the unchanged CIMOMHandle interface as defined in CIMOMHandle.h against the CIMOMHandleRep, which in the In-Process case is represented by a InternalCIMOMHandle.
2. The InternalCIMOMHandle does process the request just as before by putting a request message on the queue where it is taken off from by the CIMOperationRequestDispatcher.
3. The CIMOperationRequestDispatcher has processed the request and returns the ResponseMessage which contains a CIMResponseData object (that can hold SCMO instances and C++ default objects).
4a. CIMOMHandleRep which used InternalCIMOMHandle to take the CIMResponseData object from the ResponseMessage returns the CIMResponseData object to its caller class CIMOMHandle.
5a. CIMOMHandle then calls the respective getXXX() function [getObjects(), getInstances(), getInstanceNames(), getInstance()] to retrieve the C++ default representation as required by the provider. On returning C++ default objects CIMResponseData might do an opaque data transformation which does not have an effect to the provider.

Upcall for CMPI provider manager and providers

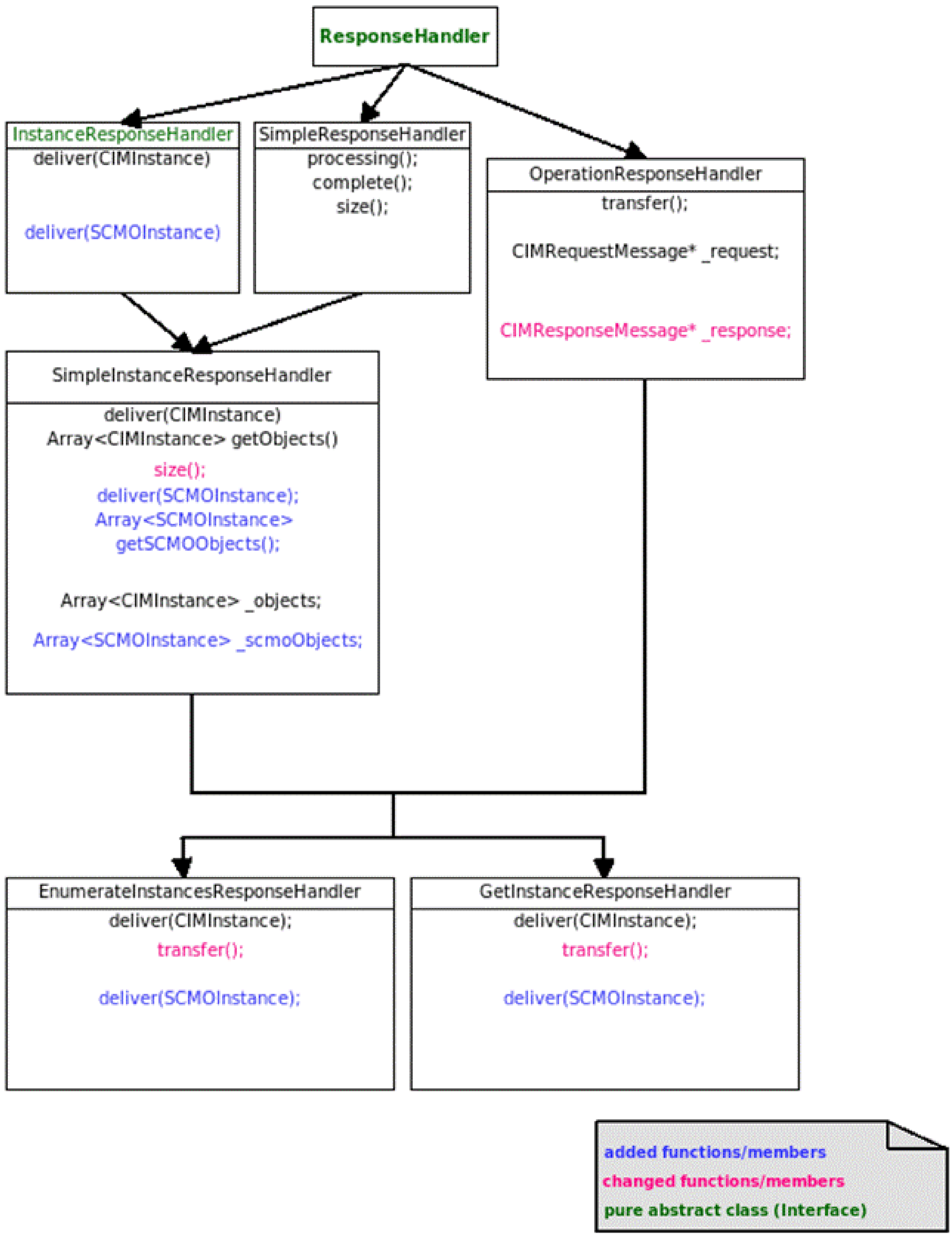
1a. The provider does make an upcall through the CMPIBroker interface directly against the CIMOMHandleRep, which in the In-Process case is represented by a InternalCIMOMHandle. The CIMOMHandle interface is circumvented and not used anymore by CMPIBroker.
2. The InternalCIMOMHandle does process the request just as before by putting a request message on the queue where it is taken off from by the CIMOperationRequestDispatcher.
3. The CIMOperationRequestDispatcher has processed the request and returns the ResponseMessage which contains a CIMResponseData object (that can hold SCMO instances and C++ default objects).
4a. CIMOMHandleRep which used InternalCIMOMHandle to take the CIMResponseData object from the ResponseMessage returns the CIMResponseData object to its caller class CIMOMHandle.
5a. CMPIBroker then calls the getSCMO() function to get the SCMOInstances and provides the CMPI layer around it required by the CMPI provider. On returning SCMO instances, CIMResponseData might do an opaque data transformation which does not have an effect to the provider.

ResponseHandlers

The major aspects are that the CIMOMHandle interface is not changed and the CIMOMHandleReps now return CIMResponseData, which then can be transformed into the target format by either the CIMOMHandle or the CMPIBroker.

How ProviderManagers and provider handle the data using the ResponseHandlers in step 4 and 5 shall be explained in more detail here.

Starting with the OperationResponseHandlers, response data is stored in class CIMResponseData and can be consumed in whatever format necessary. To allow CMPI providers to deliver the response data in as SCMO instances, the SimpleResponseHandlers are extended to hold an Array<SCMOInstance>.
The following picture shows the necessary changes and modifications to the ResponseHandlers setup using the example of the getInstance and enumerateInstance CIM requests.

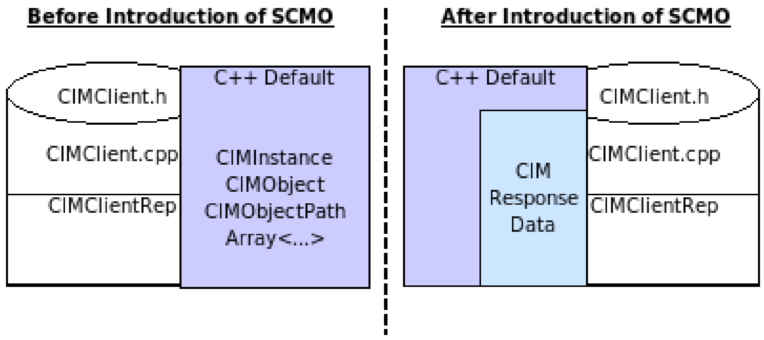


The abstract class ResponseHandler is not changed.
The class OperationResponseHandler does not really change, but the CIMResponseMessage now holds the CIMResponseData object.
The interface class InstanceResponseHandler (pure virtual) is extended by a deliver function to deliver a SCMOInstance.

The implementation class SimpleInstanceResponseHandler now also implements the deliver of a SCMOInstance by storing the SCMO instances in an Array<SCMOInstance> as well as it allows to retrieve SCMO data by function getSCMOObjects(). That function is used by the InternalCIMOMHandle for provider upcalls, but more to that later.
Of course the size() needs to be changed too, as a SimpleInstanceResponseHandler now can hold C++ default objects as well as SCMO instances.

The specific OperationResponseHandlers (EnumerateInstancesResponseHandler, GetInstanceResponseHandler) which also inherit from SimpleInstanceResponseHandler have the primary task to take data from the specific SimpleResponseHandlers and place it on the response message. For that the function transfer() is used, which is changed to place the data onto the CIMResponseData object that resides on the response messages.
Transport of response data through the CIM server Out of Process
In the Out-of-Process communication case the AnonymousPipe is used to return the response of the called provider and the ClientCIMOMHandle is used for upcalls.
The ClientCIMOMHandle is changed to use the CIMClientRep (CIMClient representation) directly, that way the CIMClient interface does not need to be changed.

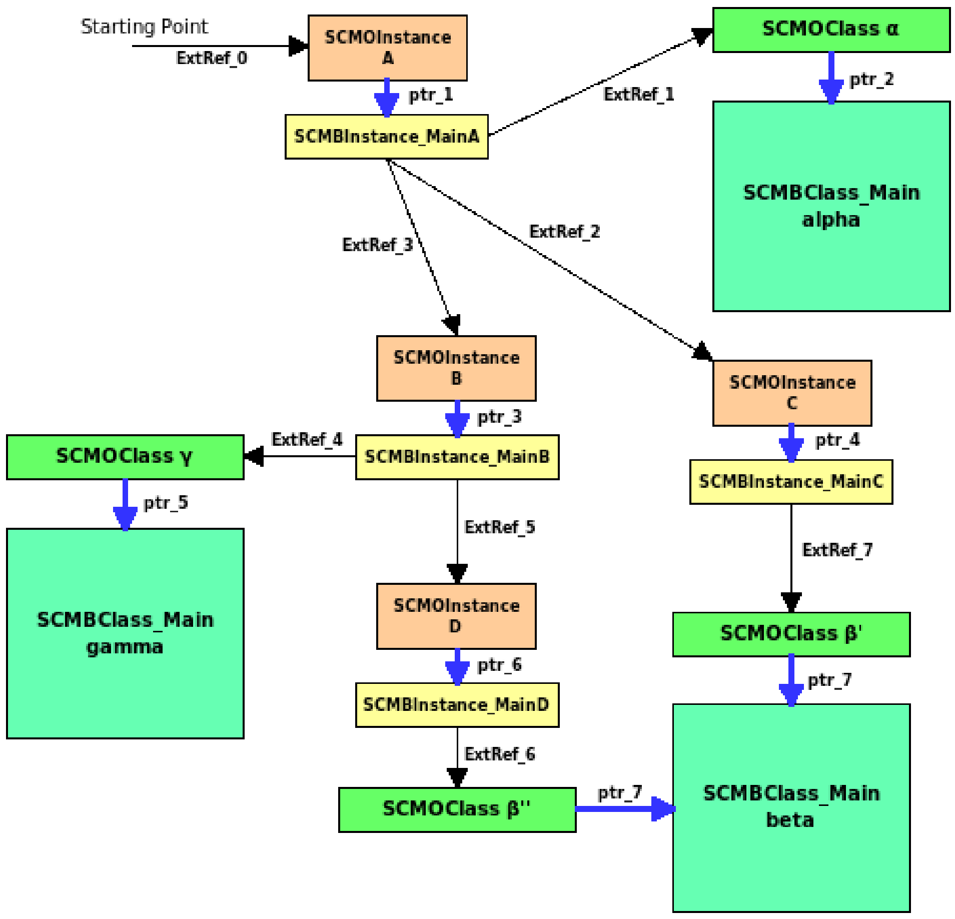
The following picture shall describe the change to the CIMClient a little more detailed:



On the CIMClient interface no change is necessary, CIMClient.cpp is extended to use the getXXX() functions to retrieve C++ default objects from the CIMResponseData object returned by CIMClientRep.
The difference in the Out of Process case is that data is stored in CIMResponseData() by the clients class CIMOperationResponseDecoder, which places the retrieved binary or Xml in CIMResponseData. Only at request, by either a client, the CIMOMHandle or the CMPIBroker, does the retrieved data get transferred back into SCMO instances or C++ default objects.

Overview of binary encoding for SCMO objects

For the Out of Process mode, as well as local clients, the CIM Server was extended to communicate using a binary protocol. The SCMO implementation supports this feature too. The central function that triggers the encoding is encodeBinaryResponse() in class CIMResponseData.
Encoding SCMO objects the same way to binary as done for C++ default objects would be unnecessary inefficent, as SCMO data is stored in memory chunks and can be dumped in a single step. Therefor the protocol is extended to write a marker about the kind of binary object (SCMO or CIM) right before writing an object(CIMInstance,CIMObjectPath,SCMOInstance etc.).
As SCMOInstances consist of SCMOInstance, SCMBInstance_Main (the instances memory block), SCMOClass and SCMBClass_Main (the class memory block) some marshalling has to be done. Also, SCMOInstance can contain embedded instances or instance path properties which are realised as external references to a SCMOInstance.
To explain the marshalling algorithmic used, the following picture shall serve as an example of how a complex SCMOInstance might look like and which components need to marshalled.



In this example the complex SCMOInstance A shall be encoded as binary. The corresponding class is stored in SCMOClass alpha.
Further two embedded instances/objectpath properties reside in SCMOInstance A, namely SCMOInstance B and SCMOInstance C.
SCMOInstance C uses SCMOClass beta prime. SCMOInstance B uses SCMOClass gamma and also has another embedded instance property with SCMOInstance D stored. SCMOInstance D holds SCMOClass beta prime prime.
SCMOClass beta prime and SCMOClass beta prime prime point to the same memory block (SCMBClass_Main beta).

Encoding

The encoder runs along an array of SCMOInstances and recursively traverses through each SCMOInstance external reference.
For each SCMOInstance it records the referenced class and external references to other SCMOInstances in a set of tables, which are later used by the receiver to restore these references.
For this pupose the following three tables are created:

1. **ClassTable**
   An array of pointers to the SCMClass_Main objects.
   One entry for each class that is referenced by all instances to be encoded.
   The order in which the SCMOClasses appear in this table is the same order as encoded.
2. **InstanceToClassResolutionTable**
   A two column table that connects SCMOInstances to SCMOClasses.
   Column 1 contains the SCMBInstance_Main pointer for an instance, and column 2 contains the index for the ClassTable at which the corresponding SCMBClass_Main pointer is stored.
   This table contains one row for each SCMOInstance (including referenced SCMOInstances) to be encoded.
   The order in which the SCMOInstances appear in this table is the order in which they are encoded.
3. **InstanceResolutionTable**
   A two column table that connects SCMOInstances to their referenced SCMOInstances.
   Column 1 contains the SCMBInstance_Main pointer for a referenced instance, and column 2 contains the index for the InstanceToClassResolutionTable at which the corresponding SCMBInstance_Main pointer is stored.
   This table contains one row for each referenced SCMOInstance to be encoded.
   The order in which the SCMOInstances appear in this table is the order in which they were found while recursively traversing the tree of SCMOInstances to be encoded.
   SCMOInstance references are restored in the same order as they are listed in this table.

In the above example the tables would look as follows:

ClassTable:

SCMBClass_Main*
beta
gamma
alpha

InstanceToClassResolutionTable:

SCMBInstance_Main*	Class Index#
D	0
B	1
C	0
A	2

InstanceResolutionTable:

SCMBInstance_Main*	Instance Index#
D	0
B	1
C	2

When the encoder has built up these tables, it serializes the array of SCMOInstances in the following order:

1. Number of SCMOClasses following (Uint32)
2. The SCMOClass memory blocks listed in the ClassTable in the exact same order
3. Number of SCMOInstances following (Uint32)
4. The contents of the InstanceToClassResolutionTable
5. Total number of external references (Uint32)
6. The contents of the InstanceResolutionTable
7. The SCMOInstance memory blocks listed in the InstanceToClassResolutionTable in the exact same order

Decoding

The decoder reinstantiates the Classes in memory and rebuilds the class table, where it now stores the SCMBClass_Main pointers for the newly instantiated class memory blocks in the same order as they were 'decoded'.
It then reinstantiates the InstanceToClassResolutionTable and InstanceResolutionTable from the input buffer followed by the Instances, where it replaces the SCMBInstance_Main pointers in the InstanceToClassResolutionTable with the pointers to the new memory location of the reinstantiated instances.
For each instantiated instance the SCMOClass pointer is updated using the InstanceToClassResolutionTable:index into the class table.
The pointers to the external references are restored the same way, using the InstanceResolutionTable.
Decoding relies on the fact that the decoding of instances occurs in the same order as the instances were added to the resolution tables, so the n-th decoded instance matches entry #n in the InstanceToClassResolutionTable and the n-th decoded referenced instance matches entry#n in the InstanceResolutionTable.

Whenever a reference to a Class or Instance is resolved by the decoder, it wraps the SCMBClass_Main or SCMBInstance_Main with a new SCMOClass or SCMOInstance object, which simply point to the SCMB pointer and increase the refcount for the wrapped SCMB.

XmlWriter updates and changes for efficent Xml generation from SCMO objects

The new class SCMOXmlWriter inheriting from XmlGenerator will be used to provide the functions necessary to generate CIM-XML from SCMO objects. Duplication of functions will be avoided as much as possible, at the same time the existing usage of XmlWriter will remain the same.
The following major functions from XmlWriter will be reimplemented in the SCMOXmlWriter using SCMO objects as input, so that the class CIMResponseData, which does generate the Xml from response objects(see fct's encodeXmlResponse), can make use of them:

- appendInstanceNameElement()
- appendInstanceElement()
- appendValueNamedInstanceElement()
- appendValueObjectWithPathElement()

Dependent functions using SCMO objects and data structures are made available as necessary and required by the beforementioned functions.

The implementation of SCMOXmlWriter will make heavy use of the fact that all values, property names etc. are not only stored in the SCMO objects, but with their length known. This offers a performance advantage over the original XmlWriter implementation as the length of strings written to the Buffer does not have to be calculated (avoid many calls of strlen). For quick access to the data the SCMOXmlWriter will be friend to the SCMO classes (SCMOInstance, SCMOClass).

Rationale

Using a single chunk of memory to host CIM classes and CIM instances(SCMO objects), the allocation/deallocation overhead is minimized.
At first SCMO is introduced for CMPI only to limit the amount of code changes and the C++ interface is not touched. This is a good proof point for the SCMO data model and from there the usage of SCMO can be expanded.

Schedule

OpenPegasus 2.10

Discussion

1) This implementation of SCMO will not allow to add properties to an instance, which are not part of the class definition.

2) The SCMOClass does not contain the definition of Methods. The target scenario (Provider --> delivery of instances) for this SCMO does not need the implementation.

3) In the CMPI select expression the CIMInstance is used to host the CQL expression. This use case is not considered in this design because CIMInstance is used as a helper tohost the expresseion. This may be a candidate for a First Class Object.

4)What is the impact of this solution on

- CMPI C++ provider interface
- Remote CMPI
- Indication Processing

Answer: The changes to the underlying implementation of the CMPI layer do NOT have any impact on providers using the CMPI interface today. Due to this, neither Remote CMPI, nor the C++ CMPI extension require any change as they are today implemented using the CMPI interface only.

Open Points

Future Items

1) Why don't we apply the same model to C++ interfaces so that it can create these objects? Do we have general set of conversion functions to move from one model to another (ex. scmo model of object to C++ model, etc.)? While this is inefficient, I would assume that it will be required since object will move freely around between providers, the persistent store, and sinks like the response and indication output mechanisms.

2) A new class CIMRequestData used by the request messages could help improve the performance further by avoiding the generation of C++ default objects which would need to be transformed to the SCMO format for CMPI providers.

3) Adding full support for all CIM operations to make use of CIMResponseData. In that case, the CIMResponseData could be moved from the leave response message classes to CIMResponseMessage.

4) Propose by Mike Brasher on Arch.Team call: Use a blocked model to reserve memory for the single chunk memory objects to avoid usage of realloc(). Could chain those blocks together.

5) Propose by Mike Brasher on Arch.Team call: Have the CMPI layer use pointers into the SCMO objects instead of having its own copies.Good doable in combination with 4) as reallocations (and thus invalidating those pointer) will disappear.

Copyright (c) 2009 IBM Corp.

Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in
the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of
the Software, and to permit persons to whom the Software is furnished to do so,
subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Template last modified: **February 17 th 2009** by **Martin Kirk**
Template version: **1. 15**