Report of the  
ISO C++Language PSM for DDS Finalization Task Force 1.0  
to the  
OMG Platform Technical Committee  
GG MM YYYY

Document Number: ptc/2011-??-??

Task Force Chair: Angelo Corsaro (PrismTech)

Specification

Revised specification (clean): ptc/2011-??-??

Revised specification (change-bar): ptc/2011-??-??

Accompanying documents

Inventory: ptc/2011-??-?? Non-normative

dds-psm-cxx.tar.gz: ptc/2011-??-?? Normative

Template: omg/09-06-01

Issue List

Issue 15965: XML-Based QoS Policy Settings (DDS-PSM-Cxx/DDS-PSM-Java) (dds-psm-cxx-ftf) 2

Issue 15967: factory methods on the "parents" (e.g. create\_topic, create\_data\_writer, etc.) (dds-psm-cxx-ftf) 2

Issue 16261: Union/array/bounded types lacking (dds-psm-cxx-ftf) 3

Issue 16269: Inconsistencies related to use of const& (dds-psm-cxx-ftf) 3

Issue 16308: Dividing a scalar in Duration and Time classes (dds-psm-cxx-ftf) 3

Issue 16338: Compilation errors on Visual Studio 2008/2010 (dds-psm-cxx-ftf) 4

Issue 16339: Improving usability of Reference<DELEGATE> class (dds-psm-cxx-ftf) 5

Issue 16340: Fixing bugs and improving usability of the InstanceHandle<D> class (dds-psm-cxx-ftf) 5

Issue 16354: Inheritance via dominance warning on Visual Studio (dds-psm-cxx-ftf) 6

Issue 16374: Use traits for topic/datareader/datawriter (dds-psm-cxx-ftf) 6

Issue 16401: Portable exception-safety guarantees for DDS C++ PSM (dds-psm-cxx-ftf) 7

Issue 16402: Exception safety guarantees for the DataReader API (dds-psm-cxx-ftf) 8

Issue 16403: General Exception Safety Considerations (dds-psm-cxx-ftf) 9

Issue 16404: Improving usability of EntityQoS API (dds-psm-cxx-ftf) 10

Issue 16405: Supporting automatic conversion from value types to delegate types (dds-psm-cxx-ftf) 10

Issue 16411: Make parameter passing same for native/type parameters (dds-psm-cxx-ftf) 11

# Issue 15965: XML-Based QoS Policy Settings (DDS-PSM-Cxx/DDS-PSM-Java) (dds-psm-cxx-ftf)

**Source:**  PrismTech (Dr. Angelo Corsaro, PhD., [angelo.corsaro(at)prismtech.com](mailto:%20angelo.corsaro(at)prismtech.com))

**Nature:**  Revision

**Severity:**  Critical

**Summary.** The newly introduced XML Based Policy configuration adds new methods in the core DDS entities that allow to fetch QoS from XML filers. This solution is not ideal since if generalized, e.g. QoS configuration from an URI, JSON stream, etc., would lead to an explosion of the core DDS API. The suggestion is to remove the added methods from the core API and use instead a Builder pattern (of some form). A sketch of the suggested change is provided below:

PolicyBuilder builder = PolicyBuilder::load("XMLBuilder");

TopicQos tqos = builder.topic\_qos(file\_name, profile\_name);

Notice that the suggested approach allows to easily extend the supported format for QoS representation w/o any impact on the core DDS API and overall facilitate the support for multiple approaches.

**Revised Text:**

**Actions taken:**

**Discussion:** This issue has a twin-issue on the dds-psm-java. The two issues need to be addressed similarly to ensure consistencies across PSM – modulo language-specific idioms.

**Proposed Resolution:** Defer

**Resolution:**  Under Discussion

# Issue 15967: factory methods on the "parents" (e.g. create\_topic, create\_data\_writer, etc.) (dds-psm-cxx-ftf)

**Source:**  PrismTech (Dr. Angelo Corsaro, PhD., [angelo.corsaro(at)prismtech.com](mailto:%20angelo.corsaro(at)prismtech.com))  
**Nature:**  Revision

**Severity:** Critical

**Summary.**The changes introduced in the final submission to support factory methods on the "parents" (e.g. create\_topic, create\_data\_writer, etc.) has introduced a few issues. a. Reference types are not created in a uniform way. In essence, DomainParticipant, Subscriber, Publisher, DataReader and DataWriter types are instantiated using a different syntax than that used by WaitSets or Conditions. This is unfortunate as it reduces the consistency of the API. b. To eliminate the circular dependencies induced by the factory methods the API has to implement a few "creative" solutions that impact clarity as well as robustness. All of a sudden it is relatively hard to find where is defined what and also the include systems is very fragile, e.g. one can break the API very easily by mistakenly changing one include order. The changes suggested to the submission is to equip each Reference type with a factory method called "create". This not only allows to remove circular references, but it also allows for an organization of the API that is more robust and easier to follow. Furthermore, it ensures that all references are created in an uniform manner.

**Resolution:** The dds-psm-cxx API has been updated to remove the circular dependencies introduced by factory methods. Along with removing circular dependencies the API has been refactored to ensure that no forward includes are needed thus maintaining local, and thus clear and visible, the set of dependencies for each file.

**Revised Text:**

**Actions taken:**

**Discussion:**

**Proposed Disposition:** Resolved

**Disposition:** Under Discussion

# Issue 16261: Union/array/bounded types lacking (dds-psm-cxx-ftf)

**Source:**  Remedy IT (Mr. Johnny Willemsen, [jwillemsen(at)remedy.nl](mailto:%20jwillemsen(at)remedy.nl))

**Nature:**  Clarification

**Severity:**  Minor

**Summary.** The specification should describe the mapping for Union, Array, and the behavior for bounded strings (what happens if we go beyond the bound)

**Resolution:**

**Revised Text:**

**Actions taken:**

**Discussion:**

# Issue 16269: Inconsistencies related to use of const& (dds-psm-cxx-ftf)

**Source:**  Remedy IT (Mr. Johnny Willemsen, [jwillemsen(at)remedy.nl](mailto:%20jwillemsen(at)remedy.nl))

**Nature:**  Clarification

**Severity:**  Minor

**Summary.** There is consistency issue in the spec with respect to the use of const & versus returning copies.

**Resolution:** The API has some inconsistencies on the use of const& vs. object copies. Many of these have been ruled out but the final clean-up will be done before closing the FTF. As such my proposition is to keep this issue open until the very end to ensure that we do a consistency check on the whole API.

**Revised Text:**

**Actions taken:**

**Discussion:**

**Proposed Disposition:** Defer

**Disposition:** Under Discussion

# Issue 16308: Dividing a scalar in Duration and Time classes (dds-psm-cxx-ftf)

**Source:**

**Nature:**  Revision

**Severity:**  Minor

**Summary.** Duration and Time have some arithmetic operators that have the form:

const Duration operator /(uint32\_t lhs, const Duration& rhs);

const Time operator /(uint32\_t lhs, const Time& rhs);

In the above, the intent of dividing a scalar by Duration/Time is not clear. Duration/N is conceivable but not N/Duration.

**Proposed solution:**

Remove the following free functions. const Duration operator /(uint32\_t lhs, const Duration& rhs); const Time operator /(uint32\_t lhs, const Time& rhs);

**Resolution:**

**Revised Text:**

**Actions taken:** Removed methods that allowed a number to be divided by a duration. Concerning the Time class the method was not there since it made no sense to multiply an absolute time for a constant.

**Discussion:**

**Proposed Disposition:** Resolved

**Disposition:** Under Discussion

# Issue 16338: Compilation errors on Visual Studio 2008/2010 (dds-psm-cxx-ftf)

**Source:**  Real-Time Innovations (Mr. Sumant Tambe, [sumant(at)rti.com](mailto:%20sumant(at)rti.com))

**Nature:**  Revision

**Severity:**  Critical

**Summary.** The sources obtained from dds-psm-cxx.googlecode.com do not compile on Visual Studio compilers without significant efforts.

There are multiple issues most related to type conversion.

1. All the classes that inherit from dds::core::Value<D> seem to be missing a copy-ctor. For example, QosPolicyCount in src\hpp\tdds\core\policy\QosPolicyCount.hpp. Without a copy-ctor, VS2010 issues a “cannot convert from ...” error. Most of these classes are found in the following files: src\hpp\tdds\core\InstanceHandle.hpp src\hpp\tdds\core\policy\CorePolicy.hpp src\hpp\tdds\core\policy\QosPolicyCount.hpp src\hpp\tdds\core\qos\EntityQos.hpp Proposed Solution: Add a copy constructor to all the classes that inherit from dds::core::Value<D> as follows: QosPolicyCount(const QosPolicyCount& src) : dds::core::Value<DELEGATE>(src.delegate()) { }
2. The exception classes in dds-psm-cxx\src\hpp\dds\core\Exception.hpp do not need copy-ctor because there is nothing to copy and the base classes don’t have copy constructors either. Proposed resolution: Remove the declarations of copy constructors in dds-psm-cxx\src\hpp\dds\core\Exception.hpp
3. The private constructor of SampleRejectedStatus in dds-psm-cxx-read-only\src\hpp\dds\core\status\State.hpp needs a typecast to avoid compilation errors on Visual studio versions of STL. The following constructor can’t be called due to ambiguous overloads of bistset<N> constructors. private: SampleRejectedState(uint32\_t s) : MaskType(s) { } Proposed solution: Change the call to the base constructor to include an explicit static\_cast to int. private: SampleRejectedState(uint32\_t s) : MaskType(static\_cast<int>(s)) { }

**Resolution:** The Visual Studio C++ 2010 compiler was raising a series of errors and warning that were not caught by GCC relating to some lacking ctors and conversion operators in some template classes. All the errors raised by Visual Studio C++ 2010 have been addressed as verifiable on the latest version of the dds-psm-cxx source code available at https://github.com/kydos/dds-psm-cxx

**Proposed Disposition:** Resolved

**Disposition:** Under Discussion

# Issue 16339: Improving usability of Reference<DELEGATE> class (dds-psm-cxx-ftf)

**Source:**  Real-Time Innovations (Mr. Sumant Tambe, [sumant(at)rti.com](mailto:%20sumant(at)rti.com))

**Nature:**  Revision

**Severity:**  Minor

**Summary.**

1. Safety of Reference<D> class should be improved by adding "explicit" keyword in the following constructors:

template <typename D> Reference(const Reference<D>& ref);

template <typename R> Reference(const R& that);

Reference(const DELEGATE\_REF\_T& ref);

1. With just member operator== and operator!= functions, Reference<D> can't be used in expressions like if(dds::null == r) { ... }

Proposed solution: Add the following free functions in dds::core in Reference.hpp.

template <class D> bool operator == (dds::null\_type,

const Reference<D> & r)

{return r.is\_nil();}

template <class D> bool operator != (dds::null\_type,

const Reference<D> & r)

{ return !r.is\_nil(); }

**Resolution:**

**Revised Text:**

**Actions taken:**

**Discussion:**

# Issue 16340: Fixing bugs and improving usability of the InstanceHandle<D> class (dds-psm-cxx-ftf)

**Source:** Real-Time Innovations (Mr. Sumant Tambe, [sumant(at)rti.com](mailto:%20sumant(at)rti.com))

**Nature:** Revision

**Severity:** Significant

**Summary.** The InstanceHandle class in dds-psm-cxx\src\hpp\tdds\core\instancehandle.hpp appears to be incomplete in several ways

1. A missing constructor InstanceHandle(const DELEGATE & d) : dds::core::Value<DELEGATE>(d) {} There is no way to construct an instance handle except a null one!
2. A missing copy-constructor InstanceHandle(const InstanceHandle& src) : dds::core::Value<DELEGATE>(src.delegate()) { }
3. Typos: a missing return and needs a dot instead of an arrow. InstanceHandle& operator=(const dds::core::null\_type& src) { return this->delegate().operator=(src); }
4. Missing comparison operators to allow comparisons like if(dds::null == instance\_handle\_object) Currently it supports other way round. The proposed solution is to add two overloaded operators in tdds::core namespace.

template <class D> bool operator==(dds::core::null\_type,

InstanceHandle<D> const &ih)

{ return ih.is\_nil(); }

template <class D> bool operator!=(dds::core::null\_type,

InstanceHandle<D> const &ih)

{ return !ih.is\_nil(); }

1. Finally, the InstanceHandle<D> class and in general the classes that support comparison with dds::null will benefit from supporting a generic and succinct syntax of the form: if(instance\_handle\_object). Proposed Solution: An idiomatic way of implenting it is the safe-bool idiom, which has been used widely in standard and boost smart pointer classes, such as std::auto\_ptr, boost::shared\_ptr. Here is a self-sufficient file that shows one way of implementing the safe bool idiom for the instance handle class: http://cpptruths.googlecode.com/svn/trunk/cpp/instance\_handle.cpp Other possible implementation based on the discussions on the boost mailing list is available here: http://codepaste.net/c83uuj

**Resolution:**

**Revised Text:**

**Actions taken:**

**Discussion:**

# Issue 16354: Inheritance via dominance warning on Visual Studio (dds-psm-cxx-ftf)

**Source:** Real-Time Innovations (Mr. Sumant Tambe, [sumant(at)rti.com](mailto:%20sumant(at)rti.com))

**Nature:** Enhancement

**Severity:** Minor

**Summary.** Visual Studio spits out many #C4250 warnings on the diamond hierarchy of listeners: for instance, DomainParticipantListener.hpp(48): warning C4250: 'dds::domain::NoOpDomainParticipantListener' : inherits 'dds::pub::NoOpPublisherListener::dds::pub::NoOpPublisherListener::on\_offered\_deadline\_missed' via dominance Proposed solution: Add empty bodies for all the inherited virtual functions in all the NoOp\*Listener classes. Particularly, NoOpDomainParticipantListener, NoOpPublisherListener, and NoOpSubscriberListener. Alternatively, #pragma warning( disable : 4250 ) could be used but its purpose will be less clear even with documentation: "Prevents via dominance warning."

**Resolution:**

**Revised Text:**

**Actions taken:**

# 

# Issue 16374: Use traits for topic/datareader/datawriter (dds-psm-cxx-ftf)

**Source:** Remedy IT (Mr. Johnny Willemsen, [jwillemsen(at)remedy.nl](mailto:%20jwillemsen(at)remedy.nl))

**Nature:** Enhancement

**Severity:** Significant

**Summary.** The proposed PSM takes the DDS PSM and now gives the users templates instead of new classes as with the IDL to C++ mapping. For example RadarTrackDataWriter becomes DataWriter<RadarTrack>. To my idea this this is syntactical sugar and I still see how the DDS implementation implement their support. I would like to propose a different way. As user I just have RadarTrack, that is coming from my user domain, so why now create a set of traits that can be used by the end user. Than he doesn't see anything special from DDS, not whether it is a template or a class. So he writes RadarTrack::data\_writer\_type dw = pub.create\_datawriter() RadarTrack::topic\_type tp = dp.create\_topic(). The DDS implementation can than do anything behind data\_writer\_type, the only thing the user has to know are the traits and the methods that are possible to be used.

**Resolution:**  The issue is rejected since the proposed change is tying the mapping of the type to generated code as opposed to use external traits. The DDS-PSM-Cxx currently uses traits specialization to devise the right Delegate type for a Topic<Foo>, DataWriter<Foo> etc. The trait specialization is generated by the code generator, yet it is not part of the Foo type thus making it easier to control the

**Revised Text:**

**Actions taken:**

**Proposed Disposition:** Reject

**Disposition:** Under Discussion

# Issue 16401: Portable exception-safety guarantees for DDS C++ PSM (dds-psm-cxx-ftf)

**Source:** Real-Time Innovations (Mr. Sumant Tambe, [sumant(at)rti.com](mailto:%20sumant(at)rti.com))

**Nature:** Clarification

**Severity:** Critical

**Summary.** The users of the DDS C++ PSM can benefit significantly if the specification standardizes exception-safety guarantees of the API. Exception safety specification for normative classes would support portable safety guarantees irrespective of vendor implementation and extensions. This would facilitate frictionless transition from one vendor to another in user code. Exception safety is about how programs, libraries behave when an exceptional condition arises. It is often much more than just raising an exception. For instance, constructor or copy-ctor of std::string may run out of memory resulting into std::bad\_alloc exception. However, in practice, APIs must specify guarantees about the internal state of the data structures after any exception propagates out. Exception-safety guarantees for C++ abstractions come in four categories: strong, basic, nothrow, and neutral. For instance, the push\_back member function of std::vector, std::list guarantees that the container remains unchanged (strong exception safety) in case of any exception. In general, programmers expect strong exception safety guarantees but it is often expensive to do so. RTI is proposing refined specifications for normative DDS C++ PSM API to make it easier to use even in the face of exceptions.

1. Strong guarantees for value types: All the constructors and copy-assignment operators of normative classes that inherit from Value<D> and the Value<D> template itself should provide strong guarantees. A simple way to achieve that is to ensure that Value<D> template provides strong guarantees and then implement the derived classes in terms of value<D>. Not all value types are POD. Some of them use std::string and may throw. Consider, for instance, SubscriptionBuiltinTopicData. Value<D> delegates the assignment to its delegate, SubscriptionBuiltinTopicDataImpl. This impl class has non-pod members such as std::string and does not define its own assignment operator. Assignment of one string (topic\_name) may succeed but the assignment of second string (type\_name) may fail with a bad\_alloc exception. This may result into inconsistent left-hand-side SubscriptionBuiltinTopicData object when DataWriter:: matched\_subscription\_data is called. In general, classes that have more than one non-pod types should have an assignment operator that is strongly safe. Instead of handpicking such classes, Value<D should be made strong exception-safe so that irrespective of vendor-specific extensions, types that users will deal with the most, will be safe to use. Proposed solution: Implement the copy-assignment operator of Value<D> using copy-and-swap technique.
2. At least basic guarantees for reference types: Not all implementations may be able to provide strong guarantees for reference types such as DataWriter<Foo>, DataReader<Foo>. Therefore, constructors and copy-assignment operators must provide at least basic guarantees. While Reference<D> itself should provide strong guarantees; just that is insufficient to extend the guarantees to all its derived classes. The normative reference types should provide strong guarantees whenever it is not prohibitively expensive to do so.

**Resolution:**

**Revised Text:**

**Actions taken:**

# Issue 16402: Exception safety guarantees for the DataReader API (dds-psm-cxx-ftf)

**Source:** Real-Time Innovations (Mr. Sumant Tambe, [sumant(at)rti.com](mailto:%20sumant(at)rti.com))

**Nature:** Clarification

**Severity:** Critical

**Summary.** The DataReader API must provide an exception-safe way to retrieve samples and must specify guarantees when exceptions are thrown. The exception safety of the DataReader API is analyzed with respect to user level state and middleware state. Read and take are both logically non-const operations with respect to the m/w because both of them have side effects such as changes in the sample\_state and instance\_state. There could be exceptions while delivering samples from m/w space to user space on the boundary of the read/take function calls. Depending upon the implementation there may or may not be a way to roll-back the changes in sample\_state and instance\_state. However, it is critical to have a way to not lose the samples even in case of exceptions. In general, it is difficult to provide strong guarantees with respect to m/w for datareader api.

1. Strong and no throw guarantees for LoanedSamples: The constructor of LoanedSamples should provide strong guarantees. All the remaining operations (specifically copy-ctor and copy-assignment operator) should provide no-throw guarantees. It is important for LoanedSample to not throw exception during copy because then the samples would be lost. Further, read/take api should minimize dynamic memory allocation and locks on the critical path for performance reasons. Therefore LoanedSamples can’t use reference semantics either. (Reference<D> uses shared\_ptr, which uses dynamic memory allocation and mutex to construct and protect the reference count.) These strict guarantees for LoanedSamples appear to be essential to support at least minimally exception safe read/take api that does no lose samples in case of exceptions. It is also the intent of LoanedSamples<T> to return the resources to m/w as soon as it goes out of scope. Proposed Solution: Use std::auto\_ptr semantics in LoanedSamples. A key property of std::auto\_ptr is that it can be safely returned while moving the resources out of an inner scope to an outer. No exceptions will be thrown. LoanedSamples<T> should be designed as such to avoid exceptions on copy. There is an idiomatic way of implementing such a class: The Move-constructor Idiom. Consequences: No std::vector<LoanedSamples<T>> would be possible. However the anticipated use-case of LoandedSamples is that the users would iterate over the range of samples and copy them and return the loan. For those users who really want to create vectors of loaned buffers, they would take ownership of the buffer from the LoanedSamples and use std::vector<shared\_ptr<T> > to automate memory management. Use of custom deleters feature of shared\_ptr may facilitate returning of the loan.
2. LoanedSamples take(): Basic exception safe. The take function is loaning a buffer from the m/w. The implementations may or may not provide a way to ‘untake' samples. So this function cannot provide strong exception safety with respect to m/w. Therefore, guarantees of LoanedSample above are applicable here. This would guarantee that the samples that are removed from the history cache are not destoyed before the user gets a chance to access them.
3. LoanedSamples read(): Basic exception safe. An attempt can be made to reread the samples. Above guarantees of LoanedSamples are applicable here
4. // --- Forward Iterators: --- // void read(SamplesFWIterator, InfoFWIterator, size\_t max\_samples): Basic exception safe void take(SamplesFWIterator, InfoFWIterator, size\_t max\_samples): Basic exception safe This API probably needs to specify guarantees with respect to two different things:
   1. With respect to state changes in the user supplied range
   2. With respect to m/w state Changes to user-supplied range ==== If Nth copy throws there is no way to recover earlier N-1 objects that were already copied successfully. So in general only basic guarantee is provided. If the first object in the range throws, nothing changes and strong safety is provided. Further the API must provide a way to return # of objects were actually copied. API should specify a precondition: iterators must be from a range that is initialized and contain valid max\_samples objects. In other words, the range cannot contain any uninitialized object. This prohibits uses such as std::vector<T> v; v.reserve(max); dr.read(v.begin(), v.end(), max); Changes to m/w state ==== The read() function has side effects like changes in the view\_state and instance\_state so it's really not a const function. The implementations may or may not provide a way to ‘unread' samples. So with respect to these status bits there is no way to provide strong exception guarantee so only basic guarantee is provided. Proposed Solution: In case of an exception (e.g., std::bad\_alloc) during these function calls, construct an internal LoanedSamples<T> object and wrap it in a ‘ReadTakeException’ object and throw it. ReadTakeException class must provide an API to retrieve the LoanedSamples. Nothrow guarantees of LoanedSamples are important here again. This solution provides at least one way to retrieve the samples that are read/take from the history cache. Such a wrapper may also need to wrap the original exception that was raised to support exception neutrality.
5. // --- Back-Inserting Iterators: --- // read(SamplesBIIterator, InfoBIIterator, size\_t max\_samples): Basic exception safe take(SamplesBIIterator, InfoBIIterator, size\_t max\_samples): Basic exception safe In addition to the exceptions that forward iterator versions can throw, these functions can throw when vector::push\_back throws. I think the same rules are applicable to these functions.

**Resolution:**

**Revised Text:**

**Actions taken:**

# Issue 16403: General Exception Safety Considerations (dds-psm-cxx-ftf)

**Source:** Real-Time Innovations (Mr. Sumant Tambe, [sumant(at)rti.com](mailto:%20sumant(at)rti.com))

**Nature:** Clarification

**Severity:** Critical

**Summary.**

1. The normative implementations shown in the wrapper classes should be at strong exception safe (or at least basic safe). For instance, implementation of DataReader::create\_readcondition as in dds-psm-cxx\src\hpp\dds\sub\DataReader.hpp is a shared\_ptr usage anti-pattern. Proposed Solution: Create separate shared\_ptr<T> objects for every new operation. Pass the shared\_ptr<T> to the constructor of ReadCondition.
2. All setters and getters (including overloaded operators) of EntityQos should be strong exception safe. Proposed Solution: Make copy-assignment operator of every qos type in normative namespaces strong exception safe.

**Resolution:**

**Revised Text:**

**Actions taken:**

# Issue 16404: Improving usability of EntityQoS API (dds-psm-cxx-ftf)

**Source:** Real-Time Innovations (Mr. Sumant Tambe, [sumant(at)rti.com](mailto:%20sumant(at)rti.com))

**Nature:** Clarification

**Severity:** Minor

**Summary.** Supporting method chaining for setting qos parameters would improve readability of the API.

For instance,

DataWriterQos dwqos;

dw >> dwqos;

Dwqos.policy<History>().kind(KEEP\_ALL).depth(200); Dwqos.policy<ResourceLimits>().max\_samples(p).max\_instances(q).max\_samples\_per\_instance(r);

dw << dwqos;

Currently, it needs different function calls for every qos parameter. Proposed Solution: Change all the normative core qos policy classes (e.g., History, ResourceLimits) to return a reference to itself from the setters. Currently they return nothing.

**Resolution:**

**Revised Text:**

**Actions taken:**

**Proposed Disposition:** Defer

**Disposition:** Under Discussion

# Issue 16405: Supporting automatic conversion from value types to delegate types (dds-psm-cxx-ftf)

**Source:** Real-Time Innovations (Mr. Sumant Tambe, [sumant(at)rti.com](mailto:%20sumant(at)rti.com))

**Nature:** Enhancement

**Severity:** Significant

**Summary.** Supporting some vendor-specific extension api is significantly easier using automatic conversion from value types to the delegate types. The situation arises when an extension method defined in the idds namespace takes a parameter defined only in the idds namespace. The parameter type may also be used as a delegate elsewhere. While the programmers can use the -> operator to invoke the extension method, the parameter need must be obtained using the delegate() method of the value type. It would be quite seamless to support automatic convertibility from value types to the delegate.

**Proposed Solution**: Define two additional generic conversion operators in

dds::core::Value<D> template.

operator D & () { return d\_; } operator const D & () const { return d\_; } ;

**Resolution:**  It is recommended that this issue is rejected as it would make it transparent for the user to call vendor specific extensions thus defeating one of the key purpose of the new API which is portability. The DDS-PSM-Cxx uses a syntactical market, the “->” operator to access extensions and providing an automatic conversion would provide uniform access via the “.” operator to proprietary operations.

**Revised Text:**

**Actions taken:**

**Proposed Disposition:** Defer

**Disposition:** Under Discussion

# Issue 16411: Make parameter passing same for native/type parameters (dds-psm-cxx-ftf)

**Source:** Remedy IT (Mr. Johnny Willemsen, [jwillemsen(at)remedy.nl](mailto:%20jwillemsen(at)remedy.nl))

**Nature:** Enhancement

**Severity:** Minor

**Summary.** We would like to propose to make the mapping for native and type parameters the same. So const T& for in and T/const T& for return. This makes the mapping easier and more consistent. For example the DDS InstanceHandle\_t is defined as native, some vendors define it as long, some as a struct which leads to different code being generated when the end user has a local interface with a DDS InstanceHandle\_t as argument. Also when the end user changes a typedef from a native to a struct the argument passing doesn't change from T to const T&. This doesn't impact performance in anyway, it just makes the mapping the same for all types

**Resolution:**  The current specification strikes for consistency and coherency with existing idiomatic practices on the target programming language. The current type mapping is consistent and coherent with C++ idioms and differently from what raised on this issue does not create amy problem with the InstanceHandle type as this is indeed a full blown class (more specifically a Value type).

**Revised Text:**

**Actions taken:**

**Proposed Disposition:** Reject

**Disposition:** Under Discussion

# Issue 16562: Typos in Value.hpp, Exception.hpp

**Source:** Real-Time Innovations (Mr. Sumant Tambe, [sumant(at)rti.com](mailto:%20sumant(at)rti.com))

**Nature:** Enhancement

**Severity:** Minor

**Summary.**

- hpp\dds\core\Value.hpp around line 67 added & to the return value:

const D\* operator->() const { return &d\_; }

2- same file, line 58, 62 added "const" to != and == operators

bool operator==(const Value& other) const {

return (d\_ == other.d\_);

}

bool operator !=(const Value& other) const {

return !(d\_ == other.d\_);

}

3- hpp\tdds\core\qos\EntityQos.hpp line 88 added "const" to declaration

const EntityQos& operator >> (POLICY& p) const {

4- dds\core\policy\CorePolicy.hpp, line 38. name() method is not public.

class policy\_name<POLICY> { \

public: \

static const std::string& name(); \

};

5- hpp\dds\core\Exception.hpp, line 202 wrong parameter type in the ctor

InvalidDataError(const InvalidDataError& src);

**Proposed Solution**: Add missing designators, etc.

**Resolution:** The missing “const” and “&” pointed out by the issue have been added as visible in the latest version of the dds-psm-cxx API available at https://github.com/kydos/dds-psm-cxx/commit/3fa67beb5c7555933b4b8c2d100609eb720a3377

**Revised Text:**

**Actions taken:**

**Proposed Disposition:** Resolved

**Disposition:** Under Discussion

# Issue 16563: RadarTrack uses anonymous types (dds-psm-cxx-ftf)

**Source:**  Remedy IT (Mr. Johnny Willemsen, [jwillemsen(at)remedy.nl](mailto:%20jwillemsen(at)remedy.nl))

**Nature:**  Style

**Severity:**  Minor

**Summary.** The RaderTrack uses an anonymous type in the example for plot on page 22

of the beta 1 spec.

So change

struct RadarTrack {

string id;

long x;

long y;

long z; //@optional

sequence<octet> plot; //@shared

};

To

typdef sequence<octet> plot\_type;

struct RadarTrack {

string id;

long x;

long y;

long z; //@optional

plot\_type plot; //@shared

};

**Resolution:**

**Revised Text:**

**Actions taken:**

**Discussion:**

**Proposed Disposition:** Defer

**Disposition:** Under Discussion

# Issue 16564: optional support (dds-psm-cxx-ftf)

**Source:**  Remedy IT (Mr. Johnny Willemsen, [jwillemsen(at)remedy.nl](mailto:%20jwillemsen(at)remedy.nl))

**Nature:**  Uncategorized

**Severity:**  Minor

**Summary.** DDS PSM does expose dds::core::optional to the end user when he uses

@optional. At some moment we also have this available at the CCM level,

what about using something that is not tied to DDS in the user defined

type? Is there nothing in STL that can be used for this?

**Proposed Disposition:** Defer

**Disposition:** Under Discussion

# Issue 16565: IDL mapping for non-trivial struct fields (dds-psm-cxx-ftf)

**Source:**  Remedy IT (Mr. Johnny Willemsen, [jwillemsen(at)remedy.nl](mailto:%20jwillemsen(at)remedy.nl))

**Nature:**  Uncategorized

**Severity:**  Minor

**Summary.** The “representative” example in Section 8.2 is not very “representative”. For example, everything in the struct can be trivially returned by value (either a basic type or a pointer) with no impact on performance.

If, instead, there were a substructure that was large and/or deeply nested, this example seems to break down. For example:

typedef sequence<long, 1000000> HugeLongSeq; // 4 MB

struct LargeStruct {

long id;

HugeLongSeq mySeq;

};

struct RadarTrack {

string id;

long x;

long y;

long z; //@optional

sequence<octet> plot; //@shared

LargeStruct myLargeMember; // new field

};

According to Section 7.4, the return value of the accessor for myLargeMember would either return by value or const reference. This would cause two copies (one for return value of getter and another for setter) of 4 MB of data in order to change one element of LargeStruct::mySeq in the instance.

I believe that use cases like this require accessors that return by non-const reference if the generated code is to be at all efficient. On page 17 of the Beta 1 spec, there is already use of this pattern for the non-const accessors on the History class (where it probably makes less sense since the types are small). I believe that this pattern should also be used for generated code where many large types can be arbitrarily nested.

Resolution:

Revised Text:

Actions taken:

**Proposed Disposition:** Defer

**Disposition:** Under Discussion