Extending the Real-Time Toolkit with your own Data Types

Copyright © 2006,2007,2009, 2010 Peter Soetens, FMTC, Peter Soetens, The SourceWorks

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation, with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of this license can be found at http://www.fsf.org/copyleft/fdl.html.

Revision History

Revision 2.0.1	7 Sept 2010	ps
	Updated status of typegen	
Revision 2.0.0	21 Jun 2010	ps
	Reworked for RTT-2.0	
Revision 1.8.0	5 Feb 2009	ps
	Added Dot operator and toolkit plugin examples	
Revision 1.0.1	24 Nov 2006	ps
	Separated from the Developer's Manual.	

Abstract

This document is an introduction to making user defined types (classes) visible within Orocos. You need to read this document when you want to see the value of an object you defined yourself, for example in the TaskBrowser component or in an Orocos script. Other uses are reading and writing objects to and from XML and generally, anything a built-in Orocos type can do, so can yours.

Table of Contents

1. The Orocos Type System : Typekits

Most applications define their own classes or structs to exchange data between components. It is easy to tell the RTT about these user defined types such that they can be displayed, stored to XML, used in the scripting engine and transfered between processes or over a network connection.

1.1. Loading Typekits

Orocos uses the 'Typekit' principle to make it aware of user types. The RTT's typekit has built-in support for the C++ types int,unsigned int,double,float, char,bool, vector<double> and string. The idea is that user contributed typekits are added at runtime.

A typekit is loaded as a plugin, as shown in the RTT Plugin Manual, from the types sudirectories of the RTT_COMPONENT_PATH directories. In case you want to load typekits manually, you can use the plugin::PluginLoader class which manages all loading and locating of RTT plugins.

1.2. Generating Typekits

There are two tools available to generate Typekits for users. The primary one is typegen, which generates typekits from existing C++ structs defined in headers. The second is based on ROS, and generates typekits for ROS Messages (which are used in ros topics).

This table lists the pro's and con's of these two approaches:

Table 1. Typekit generators

Generator	Pro's	Con's	Status
typegen	• Works with existing C++ classes • Stable and tested • Usable with the CORBA transport • Usable with the mqueue transport • Usable for scripting, XML.	• Requires all data members to be public • Ignores base classes and all data within • Does not handle typedefs • Requires to be built with 'autoproj' • Does not provide scripting constructors or operators	Available for RTT 2.x, Available for RTT 1.x
rosgen	• Compatible with ROS messages • Easy to use with many predefined data structures available.	• Can not be used with existing C++ data types • Depends on ROS • Not compatible with the CORBA or mqueue transport • Does not provide scripting constructors or operators	Not available for RTT 1.x, Available for RTT 2.x.

Important

These tools are mature for most applications and users should only write typekits by hand if they want to extend beyond what typegen/rosgen offer. In practice, this means writing a typekit for a complex data type or adding specialized constructors or operators for scripting.

1.2.1. Using typegen

Typegen requires that you have built the Orocos Toolchain with **autoproj** and that you have 'sourced' the env.sh file in your shell. You can use typegen as such:

```
cd myproject/src
typegen -o types myproject MyData.hpp
```

Which will create a directory types which contains all necessary files to build the typekit with name *myproject* for all classes and structs defined in MyData.hpp. You can list more than one header such that they are handled by the same typekit.

To test your typekit, you can do afterwards:

```
cd myproject/src/types
CMAKE_INSTALL_PREFIX=/opt/orocos CMAKE_PREFIX_PATH=/opt/orocos cmake .
make install
```

These variables tell cmake where to find Orocos and where to install the typekit:

- CMAKE_INSTALL_PREFIX : where to install to (a single directory)
- CMAKE_PREFIX_PATH : where to look for the installed Orocos Toolchain (a list of directories, similar to the PATH variable)

After make install finishes, you will see that your data types show up in the deployer and taskbrowser applications, provided that the RTT_COMPONENT_PATH variable contains the '/opt/orocos/lib/orocos' directory (= CMAKE_PREFIX_PATH + lib/orocos).

1.2.2. Using rosgen

Under development. Consult the mailinglist or the wiki pages for this tool. Announcement [http://www.ros.org/news/2010/09/first-development-release-of-orocos-toolchain-ros-v010.html].

2. Creating a Typekit in C++

Important

Only continue here if you are sure you can not use typegen/rosgen !

In case you don't use any of the typekit generators, you need to write a typekit yourself. This has been simplified in RTT 2.x, but the hardest part remains providing network transport for data types.

2.1. Telling the RTT about a struct

Say that you have an application which transfers data in a struct ControlData :

```
/** Note: you may also use 'class' instead of 'struct': */
struct ControlData {
  double x, y, z;
  int sample_nbr;
};
```

When you would use a DataPort<ControlData> and ask the taskbrowser to display the data port. You would see:

```
... unknown_t ...
```

instead of *ControlData*. The RTT has no clue on the name of your data and how to display it.

How can you tell the RTT how to handle this data type? In case your data type is a struct and allows public read/write access to its data members, you are encouraged to use the form: types::StructTypeInfo<ClassName>. Read it as: provide RTT type information for this class or struct.

This type info kind requires a helper function which is compabible with the boost::serialization library (and *must* be declared in that namespace) and which is easy to write:

```
#include <rtt/types/StructTypeInfo.hpp>

namespace boost {
  namespace serialization {
    // The helper function which you write yourself:
    template<class Archive>
    void serialize( Archive & a, ControlData & cd, unsigned int) {
      using boost::serialization::make_nvp;
      a & make_nvp("x", cd.x);
      a & make_nvp("y", cd.y);
      a & make_nvp("z", cd.z);
      a & make_nvp("sample_nbr", cd.sample_nbr);
    }
  }
}

// The RTT helper class which uses the above function behind the scenes:
struct ControlDataTypeInfo
  : public RTT::types::StructTypeInfo<ControlData>
{
   ControlDataTypeInfo()
     : RTT::types::StructTypeInfo<ControlData>("ControlData")
   {}
};

// Tell the RTT the name and type of this struct:
RTT::types::Types()->addType( new ControlDataTypeInfo() );
```

Note

In case you write your serialize with 'getter' functions, for example,

```
cd.getX()
```

, the getX() function *must* return a reference to x, i.e. have the signature: x& getX(). Returning const x& getX() or x getX() will not work and the code will not compile.

From now on, the RTT knows the 'ControlData' type name and allows you to write it to XML, use it in scripts and access its member variables. For example, you may write in a script:

```
var ControlData mycd;
mycd.x = 2 * mycd.y  = 2 * mycd.z = 1; // (x,y,z)=(4,2,1)
mycd.sample_nbr = 1;

var ControlData mycd2 = mycd;
// ...
```

Note

The type is now usable as a 'var' in a script, however, you may need to add a constructor as well. See Section 3.2, “Loading Constructors”.

If your struct or class contains other complex types, for example, ControlData's sample_nbr field is a struct 'Sample' itself, you need to apply the same method for that struct: create a serialize() function and register the type with the StructTypeInfo class. In case one of the fields is an array or sequence type, another approach is taken, as explained below.

2.2. Telling the RTT about a complex data type

It is not recommended to use complex data types for communicating between components. Especially if your data contains pointers to other data, it is possible that it can't be sent between components. This section tells you how you can add 'whatever' type to a typekit, but you'll have to implement all functions yourself.

Reasons to follow this path are:

- It's impossible to provide a serialize() function.
- You want full control over XML format, member access in scripting etc.
- The XML and scripting representations look different.

It is however recommend to use the StructTypeInfo if a serialize() function can be written, and then to override the required functions as shown in the next sections.

Complex classes must be carefully written such that they contain:

- A default constructor
- A copy constructor that can initialize a default constructed object

```
 /** class has read-only members */
class ControlClass {
  const int joints;
public:
  // Mandatory !
  ControlClass() : joints(-1) {}
  // Mandatory !
  ControlClass(const ControlClass& orig ) : mjoints(orig.mjoints) {}

  ControlClass( int joints ) : mjoints(joints)
 int getJoints() { return joints; }
 };
```

The way to add this type to the typekit is by inheriting from the types::TemplateTypeInfo<ControlClass> class and then to specialize one by one the required functions, as explained in the next sections:

```
 // The RTT helper class for any class which has default constructor and copy constructor:
struct ControlClassTypeInfo
  : public RTT::types::TemplateTypeInfo<ControlClass>
{
   ControlClassTypeInfo()
     : RTT::types::TemplateTypeInfo<ControlClass>("ControlClass")
   {}

   // Note: you'll have to implement virtual functions here,
   // as documented by the types::TypeInfo class.
};

// Tell the RTT the name and type of this class:
RTT::types::Types()->addType( new ControlClassTypeInfo() );
```

2.3. Struct versus Sequence

The ControlData struct is clearly a 'struct' in the C/C++ sense. But sometimes, you have datatypes that behave more like sequences. For example, a std::vector<ControlData> or a ControlData[100] field in another struct. In that case, we register the resulting type as a types::SequenceTypeInfo<ClassName>. For example:

```
 #include <rtt/types/SequenceTypeInfo.hpp>
 #include <rtt/types/CArrayTypeInfo.hpp>
 #include <rtt/types/BoostArrayTypeInfo.hpp>

// Register a std::vector<ControlData> (or compatible) :
RTT::types::Types()->addType( new types::SequenceTypeInfo<std::vector<ControlData>
>("std.vector<ControlData>") );

// Register a C-Array ControlData[ N ] or ControlData* :
RTT::types::Types()->addType( new types::CArrayTypeInfo<types::carray<ControlData>
>("ControlData[]") );

// Register a Boost-Array boost::array<ControlData> :
RTT::types::Types()->addType( new types::BoostArrayTypeInfo<boost::array<ControlData>
>("boost.array<ControlData>") );
```

Note that we have adapted TypeInfo's type name argument to fit the scripting type name syntax.

In case a type is one of these sequence, it does not need to provide a serialize() function !

Here's a complete example of combining a Struct and a Sequence:

```
#include <rtt/types/SequenceTypeInfo.hpp>
#include <rtt/types/StructTypeInfo.hpp>
#include <rtt/types/CArrayTypeInfo.hpp>

struct ControlData {
  double x, y, z;
  int sample_nbr;
};
```

```
struct ControlDataSet {
  // warning: strings may render your type non-real-time.
  string setname;

  // Variable size data set:
  vector<ControlData> dataset;

  // Fixed size array:
  unsigned int timestamp[2];
};

namespace boost {
  namespace serialization {
   // The helper function which you write yourself for ControlData:
   template<class Archive>
   void serialize( Archive & a, ControlData & cd, unsigned int) {
      using boost::serialization::make_nvp;
      a & make_nvp("x", cd.x);
      a & make_nvp("y", cd.y);
      a & make_nvp("z", cd.z);
      a & make_nvp("sample_nbr", cd.sample_nbr);
   }

   // The helper function which you write yourself for ControlDataSet:
   template<class Archive>
   void serialize( Archive & a, ControlDataSet & cds, unsigned int) {
      using boost::serialization::make_nvp;
        using boost::serialization::make_array;
      a & make_nvp("setname", cds.setname);
      a & make_nvp("dataset", cds.dataset);

        // NOTE: we require 'make_array' + size of array for fixed size arrays.
      a & make_nvp("timestamp", make_array( cds.timestamp, 2) );
   }
}
}

// Tell the RTT the name and type of a struct:
RTT::types::Types()->addType( new types::StructTypeInfo<ControlData>("ControlData") );
// Register a std::vector (or compatible) for ControlData:
RTT::types::Types()->addType( new types::SequenceTypeInfo<std::vector<ControlData>
>("std.vector<ControlData>") );

// Register an array for unsigned ints (NOTE: use of types::carray<unsigned int> instead of
'unsigned int'):
RTT::types::Types()->addType( new types::CArrayTypeInfo<types::carray<unsigned int>
>("uint[]") );

// Tell the RTT the name and type of a struct:
RTT::types::Types()->addType( new types::StructTypeInfo<ControlDataSet>("ControlDataSet") );
```

2.4. Displaying and Reading

There is no default implementation for reading or writing your data to a stream. You need to implement this yourself. This is optional for most types, since most code will use the introspection functions (getMember()) of your type to learn about the internals.

In order to tell the RTT how to display your type, you may overload the TypeInfo::read and TypeInfo::write functions OR define operator<<() and operator>>() for your type (preferred). The code below shows the latter option:

```
#include <rtt/types/StructTypeInfo.hpp>
#include <ostream>
#include <istream>

// Displaying:
std::ostream& operator<<(std::ostream& os, const ControlData& cd) {
  return os << '('<< cd.x << ',' << cd.y << ',' << cd.z << '): ' << cd.sample_nbr;
}

// Reading :
std::istream& operator>>(std::istream& is, ControlData& cd) {
  char c;
  return is >> c >> cd.x >> c >> cd.y >> c >> cd.z >> c >> c >> cd.sample_nbr; // 'c' reads '(' ',' ','
')' and ':'
}
// ...
// The 'true' argument means:  it has operator<< and operator>>
struct ControlDataTypeInfo
 : public RTT::types::StructTypeInfo<ControlData,true>
{
  ControlDataTypeInfo()
    : RTT::types::StructTypeInfo<ControlData,true>("ControlData")
  {}
};

// Tell the RTT the name and type of this struct
RTT::types::Types()->addType( new ControlDataTypeInfo() );
```

If you use the above line of code to add the type, the RTT will be able to display it as well, for example in the TaskBrowser. Other subsystems may use your operators to exchange data in a text-based form. However, in order to 'construct' your type in a script or at the TaskBrowser prompt, you need to add a constructor as shown in Section 3.2, “Loading Constructors”.

2.5. Reading/Writing data from/to XML

Every data type that has been defined using the StructTypeInfo and a proper serialization function, can be written to XML. The ControlStruct data type will be encoded like this:

```
<struct name="MyData" type="ControlData">
  <simple name="x" type="double">
    <value>0.12</value>
  </simple>
  <simple name="y" type="double">
    <value>1.23</value>
  </simple>
  <simple name="z" type="double">
    <value>3.21</value>
  </simple>
  <simple name="sample_nbr" type="short">
    <value>3123</value>
  </simple>
</struct>
```

In case you didn't use the StructTypeInfo or you want to override the default you may implement the composeType() and decomposeType() functions of the types::TypeInfo class, which is explained in the next two sections.

2.5.1. Changing how your data is written to XML

In case the default XML format is not good for you, or you inherited from TemplateTypeInfo, you need to inform Orocos of the structure of your data type. It must be given a 'decompose' function: Of which primitive types does the data consists ? Representing structured data is what Orocos Property objects do. Here is how to tell Orocos how the "ControlData" is structured:

```
// We use StructTypeInfo, so we override the defaults...
struct ControlDataTypeInfo
  : public StructTypeInfo<ControlData,true>
{
   // ... other functions omitted

   // this is a helper function, which is called by composeType() of the same class:
   virtual bool decomposeTypeImpl(const ControlData& in, PropertyBag& targetbag ) const {
      targetbag.setType("ControlData");
      targetbag.add( new Property<double>("X", "X value of my Data", in.x ) );
      targetbag.add( new Property<double>("Y", "Y value of my Data", in.y ) );
      targetbag.add( new Property<double>("Z", "Z value of my Data", in.z ) );
      targetbag.add( new Property<int>("Sample", "The sample number of the Data",
in.sample_nbr ) );
      return true;
   }
}
```

This function reads as such: For each member of your struct, add a Property of the correct type to the targetbag and initialize it with the value of the 'in' parameter. setType() is mandatory and can be used lateron to determine the version or type of your XML representation. Next, if Orocos tries to write an XML file with ControlData in it, it will look like:

```
<struct name="MyData" type="ControlData">
  <simple name="X" type="double">
    <description>X value of my Data</description>
    <value>0.12</value>
  </simple>
  <simple name="Y" type="double">
    <description>Y value of my Data</description>
    <value>1.23</value>
  </simple>
  <simple name="Z" type="double">
    <description>Z value of my Data</description>
    <value>3.21</value>
  </simple>
  <simple name="Sample" type="short">
    <description>The sample number of the Data</description>
    <value>3123</value>
  </simple>
</struct>
```

2.5.2. Changing how data is read from XML

When you modified the default writing to XML, you probably need to modify the default reading as well. This step is called 'composition', and means: put all the individual XML elements back into one data structure.

Here is how to tell Orocos how the "ControlData" is read:

```
// ...
struct ControlDataTypeInfo
  : public TemplateTypeInfo<ControlData,true>
{

  // ... other functions omitted

  virtual bool composeTypeImpl(const PropertyBag& bag, ControlData& out ) const
  {
    if ( bag.getType() == std::string("ControlData") ) // check the type
    {
      Property<double>* x = targetbag.getProperty("X");
      Property<double>* y = targetbag.getProperty("Y");
      Property<double>* z = targetbag.getProperty("Z");
      Property<int>* t = targetbag.getProperty("Sample");

      if ( !x || !y || !z || !t )
          return false;

      out.x = x->get();
      out.y = y->get();
      out.z = z->get();
      out.sample_nbr = t->get();
      return true;
    }
    return false; // unknown type !
  }
}
```

First the type is checked and then the properties are located in the bag, it should look just like we stored them. If not, return false, otherwise, read the values and store them in the out variable.

2.6. Network transfer using CORBA

In order to transfer your data between components using the CORBA network transport, the RTT requires that you provide the conversion from your type to a CORBA::Any type and back. This procedure is done automatically if you use **orogen**.

The first step is describing your struct in IDL and generate the 'client' headers with 'Any' support. Next you create such a struct, fill it with your data type's data and next 'stream' it to an Any. The other way around is required as well.

In addition, you will need the CORBA support of the RTT enabled in your build configuration.

Note

This procedure is discussed in detail on the Orocos wiki [http://www.orocos.org/wiki/rtt/simple-examples/developing-plugins-and-toolkits/part-3-transport-plugin].

3. Building your own Typekit

The number of types may grow in your application to such a number or diversity that it may be convenient to build your own typekit and make it a plugin. Non-Orocos libraries benefit from this system as well because they can introduce their data types into Orocos.

Each typekit must define a class that inherits from the types::TypekitPlugin class and implement four functions: loadTypes(), loadConstructors, loadOperators() and getName().

The name of a typekit must be unique. Each typekit will be loaded no more than once. The loadTypes function contains all 'StructTypeInfo' constructs to tell the RTT about the types of your typekit. The loadOperators function contains all operations that can be performed on your data such as addition ('+'), indexing ('[i]'), comparison ('==') etc. Finally, type constructors are added in the loadConstructors function. They allow a newly created script variable to be initialised with a (set of) values.

Mimick the code of the types::RealTimeTypekitPlugin to build your own.

Your typekit header file might look like:

```
#ifndef ORO_MY_TYPEKIT_HPP
#define ORO_MY_TYPEKIT_HPP

#include <rtt/types/TypekitPlugin.hpp>

namespace MyApp
{
  /**
   * This interface defines the types of my application
   */
  class MyTypekitPlugin
    : public RTT::types::TypekitPlugin
  {
  public:
      virtual std::string getName();

      virtual bool loadTypes();
      virtual bool loadConstructors();
      virtual bool loadOperators();
  };
}
#endif
```

The mytypekit.cpp files looks like:

```
#include "mytypekit.hpp"

namespace MyApp {
  std::string MyTypekitPlugin::getName() { return "MyTypekit"; }
```

```
  bool MyTypekitPlugin::loadTypes() {
    // Tell the RTT the name and type of this struct
    RTT::types::Types()->addType( new ControlDataTypeInfo() );
  }

  /** ...Add the other example code of this manual here as well... */
  bool MyTypekitPlugin::loadConstructors() {
    // ...
  }
  bool MyTypekitPlugin::loadOperators() {
    // ...
  }
} // namespace MyApp

/** Register the class as a plugin */
ORO_TYPEKIT_PLUGIN( MyApp::MyTypekitPlugin );
```

Next compile the .cpp file as a shared library and put it in the types subdirectory of the RTT_COMPONENT_PATH.

The Plugin Manual explains more in detail how plugins are located or can be loaded from C ++ code. The class that manages plugin loading is the plugin::PluginLoader.

3.1. Loading Operators

Some data types may support mathematical operations. In that case, you can register these operators to the RTT type system such that you can use them in your scripts. In case your type does not need such operations, this section can be skipped.

Operator are stored in the class OperatorRepository in Operators.hpp. The list of supported operators is set by the typekit and added to the OperatorRepository It looks something like this:

```
bool loadOperators() {
  OperatorRepository::shared_ptr oreg = RTT::operators();
  // boolean stuff:
  oreg->add( newUnaryOperator( "!", std::logical_not<bool>() ) );
  oreg->add( newBinaryOperator( "&&", std::logical_and<bool>() ) );
  oreg->add( newBinaryOperator( "||", std::logical_or<bool>() ) );
  oreg->add( newBinaryOperator( "==", std::equal_to<bool>() ) );
  oreg->add( newBinaryOperator( "!=", std::not_equal_to<bool>() ) );
  return true;
}
```

Adding your own should not be terribly hard. The hardest part is that as the second argument to newUnaryOperator, newBinaryOperator or newTernaryOperator, you need to specify a STL Adaptable Functor, and even though the STL provides many predefined one's, it does not provide all possible combinations, and you might end up having to write your own. The STL does not at all provide any "ternary operators", so if you need one of those, you'll definitely have to write it yourself.

Note that this section is only about adding overloads for existing operators, if you want to add new operators to the scripting engine, the parsers need to be extended as well.

3.2. Loading Constructors

Constructors can only be added *after* a type has been loaded using addType. Say that the ControlData struct has a constructor:

```
struct ControlData {
  // Don't forget to supply the default constructor
  // as well.
  ControlData()
    : x(0), y(0), z(0), sample_nbr(0)
  {}
  ControlData(double a, double b, double c)
    : x(a), y(b), z(c), sample_nbr(0)
  {}
  double x, y, z;
  int sample_nbr;
}
```

This constructor is not automatically known to the type system. You need to write a constructor function and add that to the type info:

```
// This 'factory' function constructs one object.
ControlData createCD(double a, double b, double c) {
  return ControlData(a,b,c);
}

// Tell the RTT a constructor is available:
// Attention: "ControlData" must have been added before with 'addType' !
RTT::types::Types()->type("ControlData")->addConstructor( newConstructor(&createCD) );
```

From now on, one can write in a script:

```
var ControlData cd = ControlData(3.4, 5.0, 1.7);
```

Multiple constructors can be added for the same type. The first one that matches with the given arguments is then taken. For example:

```
// Add 'ControlData( 3.0 )' constructor:
ControlData createCD_2(double a) {
  return ControlData(a,a,a);
}

// Tell the RTT a constructor is available:
RTT::types::Types()->type("ControlData")->addConstructor( newConstructor(&createCD_2) );
```