Event QoS Aspect Language (EQAL) User Guide

Gan Deng
gan.deng@vanderbilt.edu

1. Introduction to EQAL and CoSMIC

The term CoSMIC stands for the name of a tool suite – Component Synthesis with Model Integrated Computing. CoSMIC is an ongoing project, and lots of information about it can be found at:

http://www.dre.vanderbilt.edu/cosmic/

Basically, CoSMIC is manifested in the integration of Model Driven Development (MDD) with QoS-enabled component middleware.

EQAL, as a tool integrated in CoSMIC tool suite, stands for Event QoS Aspect Language, which is the modeling tool you need to use to set up and configure real-time event channels. Since this is an on-going project, all the pieces of the CoSMIC tool suite, including EQAL are constantly being improved, extended, maintained, and integrated with other tools.

EQAL is a graphical modeling language for the building blocks of component applications, that is, applications that are built on top of component middleware.
EQAL itself was designed using the Generic Modeling Environment (GME), a powerful modeling tool developed at the Institute for Software Integrated Systems (ISIS) at Vanderbilt University. Documentation, download and other information can be found on the GME web page at:

http://www.isis.vanderbilt.edu/Projects/gme/

Although many GME-specific details will be explained as they come up in this manual, it is not intended to be both an EQAL manual and a GME manual, so the user should refer to the GME manual whenever appropriate or necessary.

2. Motivation of the EQAL (When/why is it useful?)

Component middleware generally supports two models for component interaction: (1) a *request-response* communication model, in which a component invokes a point-to-point operation on another component, and (2) an *event-based* communication model, in which a component transmits arbitrarily-defined, strongly-typed messages, called *events*, to other components. Event-based communication models are particularly relevant for large-scale DRE systems (such as avionics mission computing, distributed audio/video processing, and distributed interactive simulations) because they help reduce software dependencies and enhance system composability and evolution. In particular, the publisher/subscriber architecture of event-based communication allows application components to communicate anonymously and asynchronously. The publisher/subscriber communication model defines three software roles:

- *Publishers* generate events to be transmitted.
- *Subscribers* receive events via hook operations.
- *Event channels* accept events from publishers and deliver events to subscribers.

The publisher/subscriber design is an especially powerful architecture for event-based communication because it provides (1) *anonymity* by decoupling event publishers and subscribers and (2) *asynchrony* by automatically notifying subscribers when a specified event is generated.

Component-based distributed, real-time and embedded (DRE) software deployments often require custom QoS configurations to target multiple OS, network, and hardware platforms, each of which may have slightly different requirements. In such cases, event QoS requirements (such as event dispatching strategy, latency thresholds and priorities) may only be known when components are deployed, rather than when they are developed. Contemporary component middleware frameworks use XML descriptor files to specify the publisher/subscriber configurations and QoS constraints associated with particular software deployments. The component deployment and configuration framework is responsible for parsing these XML files and making the appropriate invocations on the publisher/subscriber configuration interface provided by the middleware run-time infrastructure.

However, it is often tedious and error-prone to *manually* specify the QoS requirements of large-scale DRE component deployments by *handwriting* XML files. Component middleware has become complex to configure due to an increasing number of operating policies (such as persistence and lifecycle management, and publisher/subscriber QoS configurations) that exist at multiple middleware layers and employ legacy specification mechanisms not based on XML. To further complicate matters, many combinations of policies are semantically invalid and will most likely result in disastrous system failure in deployment phase!

In the context of component applications, modeling tools enable the creation of reusable component deployment models that are easier to build, understand, and maintain than manually written deployments. Moreover, modeling tools can automatically (1) generate XML descriptor files that describe component interactions and configuration files that specify middleware operating policies and (2) validate models to ensure consistency and coherency among component properties.

As a result, EQAL modeling tool was developed to help to configure the publisher/subscriber service in component middleware. Currently EQAL supports the service setup and service configuration of Real-Time Event Service, which is one of the most widely used publisher/subscriber services in component middleware.

3. EQAL Model Building

This manual describes how to use EQAL for graphically specifying publisher/subscriber service configurations. Basically EQAL consists of two complimentary entities:

- A *meta-model* that defines a modeling language, or *paradigm*. The meta-model specifies the types of modeling elements, their properties, and relationships.
- Model *interpreters* that synthesize middleware configuration files from models. The EQAL model interpreters automatically generate publisher/subscriber service configuration files and component property description files.

Publisher/subscriber service policies have differing scope, from a single port to an entire event channel. The EQAL tool allows modelers to provision reusable and sharable configurations at each level of granularity. A modeler assigns configurations to individual event connections.

3.1 Getting Started with EQAL

Installing EQAL

A Windows installer for PICML is available from the CoSMIC web page listed above. Just look at the first item under the Downloads header near the top of the page. It is shown as "*Windows Installer for PICML available here*". The EQAL tool is integrated in this installer.

Prerequisites

After installing PICML there are two steps that are required before one can model the service setup and service configuration of the real-time event service. These are common to someone building models from scratch of existing PICML models. These include:

1. Component interfaces: The interfaces provided and required by components.
2. Component implementations: Component instances and their interconnections.

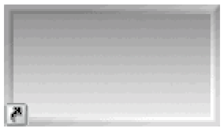
The step (1) above is accomplished via the ComponentTypes aspect in PICML. This aspect deals with specifying the interfaces of the ports. Please refer to the Interfaced Definitions Modeling Language (IDML) manual present with the installation of PICML.

The step (2) above is provided via ComponentImplementation aspect in PICML. In this view, the user can drag and drop components and draw the component interactions. For more information please refer to the user guide exclusively dealing with PICML.

From this point, I assume that the models have both the component interactions and component interfaces. The assumption also includes how to open an existing model, create a new model (PICML) in GME and some GME terminology.

3.2 EQAL Modeling Elements

This section describes the key EQAL modeling elements, both their syntax and semantics. For the purpose of real-time event channel setup and configuration, EQAL provides the following model elements:



`Component_Reference` points to a component instance. The actual component is defined in the *ComponentImplementation* aspect of the PICML. EQAL uses this information to specify which components you want to set up the real-time event service for. In terms of GEM terminology, it is the type of "reference to model type element".

The **`RTEC_Resource_Factory`** **s**tands for real-time event channel resource factory, which is responsible for creating many strategy objects that control the behavior of the event channel, such as dispatching, locking options, consumer and supplier control options. This modeling element is useful to set up the "*per-channel*" scope QoS properties, i.e., the molder could set up the per-channel behavior by modifying the attributes of this modeling element. EQAL will use such attributes to generate the event channel service configuration file, which could be registered by the event channel's service configurator. In terms of GME terminology, this is with the type of "atom element".

This documentation doesn't want to devote to the semantic meaning of each strategy object, so please refer to additional reference, such as "TAO Developer's Guide version 1.3a, OCI".

The **`RTEC_Resource_Factory_Reference`** points to an actual

RTEC Resource Factory. It is contained to configure the QoS properties of the publisher/emitter port. The same RTEC Resource Factory configuration could be referenced by a number of different event publisher/emitter port QoS configuration.

The **`RTEC_Proxy_Consumer`** is used to configure the "event source" port scope QoS properties. Each **`RTEC_Proxy_Consumer`** should be used for each event source that will use the Real-time Event Service. In GME terminology, this is with the type of "Model element".

The **`RTEC_Proxy_Supplier`** is used to configure the "event sink" port scope QoS properties. Each **`RTEC_Proxy_Supplier`** should be used for each event sink that will use the Real-time Event Service. In GME terminology, this is with the type of "Model element".

The **`RT_Info`** modeling element could be used to configure the Real-Time properties of the associated event source and event sink by setting the desired QoS attributes. Since each event delivery operation (event source pushes an event to event channel, event channel pushes an event to event sink) is usually a CORBA operation, so **`RT_Info`** is used to set up the QoS properties of such an operation. In GME terminology, this is with the type of "Atom element".

Alternatively, the **`Null_RT_Info`** modeling element could be used if the modeler doesn't want to configure the real-time properties of the event source or event sink port, then default behavior (no scheduling service) will be executed.

The **`RT_Info`** structure contains the following operation characteristics as described below:

(1) **Criticality**: Criticality is an application-supplied value that indicates the significance of a CORBA operation's completion prior to its deadline. Higher criticality should be assigned to operations that incur greater cost to the application if they fail

to complete execution before their deadlines. Some scheduling strategies, such as MUF, take criticality into consideration, so that more critical operations are given priority over less critical ones.

(2) **Worst-case execution time**: This is the typical execution time it takes to execute a single dispatch of the operation.

(3) **Worst-case execution time**: This is the longest time it can take to execute a single dispatch of the operation.

(4) **Cached execution time**: This is the execution time that accounts for server data caching time.

(5) **Period**: Period is the interval between dispatches of an operation.

(6) **Importance**: Importance is a lesser indication of a CORBA operation's significance. Like its criticality, an operation's importance value is supplied by an application. Importance is used as a "tie-breaker" to distinguish between operations that otherwise would have identical priority.

(7) **Dependencies**: An operation depends on another operation if it is invoked only via a flow of control from the other operation.

(8) **Entry point**: This is an application-defined string that uniquely identifies the operation.

(9) **Threads**: The number of internal threads contained by the operation used for the dispatching.

(10)**Quantum**: For time-slicing, which means when the operation is preempted by other operations, the minimum time slice specified by this Quantum should be guaranteed.

3.3 Setup and Configure an RT Event Channel: An Example Walk Through

Basically, the real-time event channel setup and configuration could be broken into two phases, the first one is called **Setup Phase**, and the second is called the **Configuration Phase**. Among these two phases, the **Configuration Phase** is optional, which means that a modeler doesn't have to explicitly specify the QoS properties of the real-time event service, and as a result the default behaviors will be utilized.

The detailed modeling process about each phase is described as following:

Setup Phase

In this phase, a modeler could specify which event source/sink connection(s) among a set of component to use the RT Event Service by simply setting a single enumeration type attribute. Then all the underlying real-time event service setup and lifecycle management will be automatically done by the component middleware's deployment/runtime framework.

Step 1: Go to the ComponentImplementations folder**,** and then open the paradigm sheet for the component assembly implementation of your application. This paradigm sheet shows the component instances and how they are interconnected with each other, such as the one illustrated in figure 1.

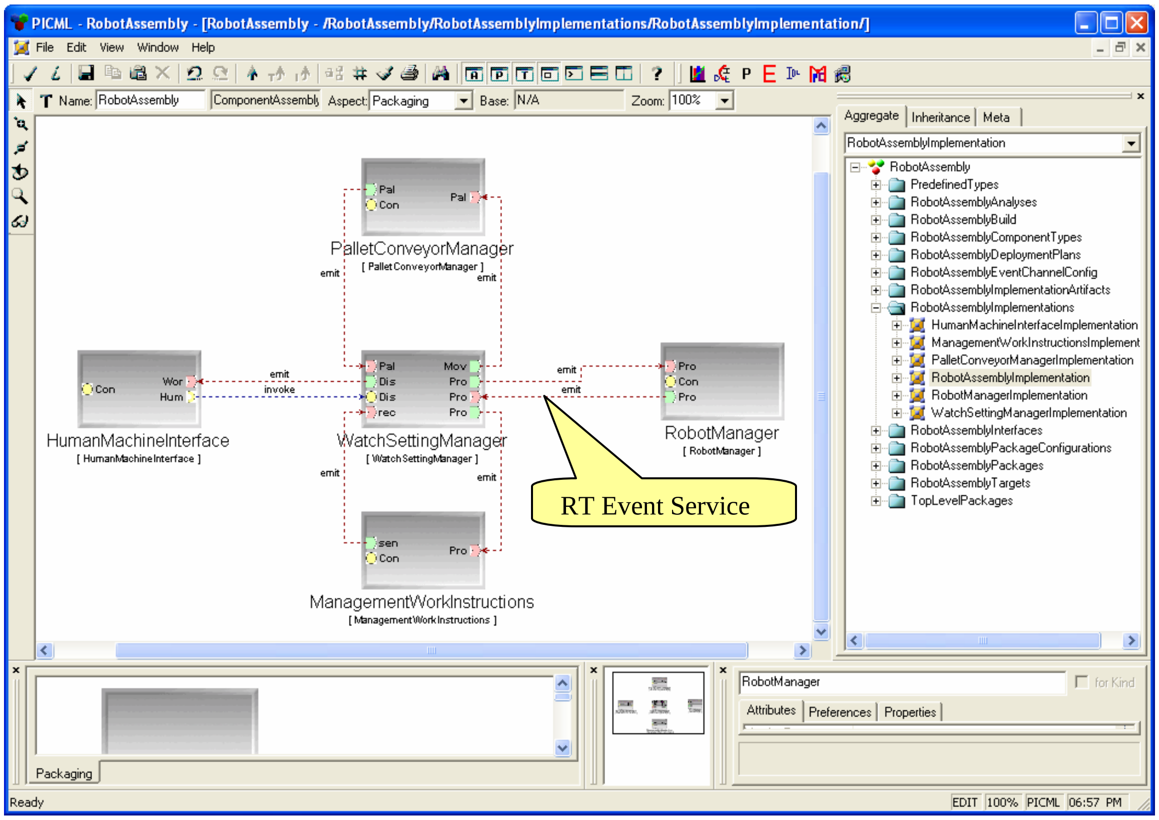
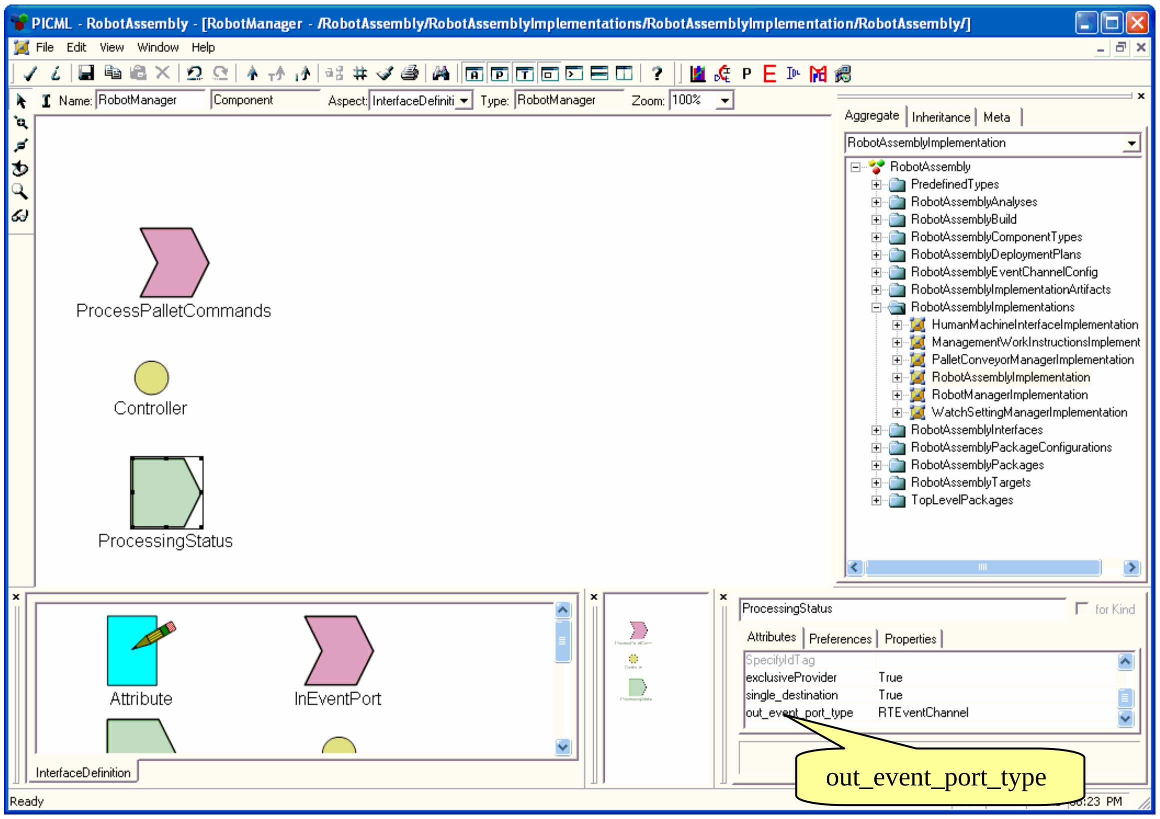

PICML - RobotAssembly - [RobotAssembly - /RobotAssembly/RobotAssemblyImplementations/RobotAssemblyImplementation/]
File Edit View Window Help
Name: RobotAssembly
ComponentAssembly
Aspect: Packaging
Base: N/A
Zoom: 100%
PalletConveyorManager
[PalletConveyorManager]
HumanMachineInterface
[HumanMachineInterface]
WatchSettingManager
[WatchSettingManager]
RobotManager
[RobotManager]
ManagementWorkInstructions
[ManagementWorkInstructions]
emit
invoke
RT Event Service
Aggregate
Inheritance
Meta
RobotAssemblyImplementation
RobotAssembly
PredefinedTypes
RobotAssemblyAnalyses
RobotAssemblyBuild
RobotAssemblyComponentTypes
RobotAssemblyDeploymentPlans
RobotAssemblyEventChannelConfig
RobotAssemblyImplementationArtifacts
RobotAssemblyImplementations
HumanMachineInterfaceImplementation
ManagementWorkInstructionsImplement
PalletConveyorManagerImplementation
RobotAssemblyImplementation
RobotManagerImplementation
WatchSettingManagerImplementation
RobotAssemblyInterfaces
RobotAssemblyPackageConfigurations
RobotAssemblyPackages
RobotAssemblyTargets
TopLevelPackages
Packaging
RobotManager
for Kind
Attributes
Preferences
Properties
Ready
EDIT 100% PICML 06:57 PM

Figure 1: RobotAssembly Implementation paradigm sheet for the assembly

Step 2: Identify the first event connection which you want to set up using the RT Event Service, and then double click the component icon which contains the event source as port, i.e., the component that contains the event publisher port or the event emitter port for the event connection you are interested in. For example, if you want to use the real-time event between component RobotManager and WatchSettingManager with the *ProcessingStatus* as the event port (as show on the above figure), then you double click the RobotManager component instance since it contains the event emitter port called *ProcessingStatus*. After you do this, you will see all the contained model elements of the RobotManager component as illustrated in Figure 2.



Name: RobotManager | Component | Aspect: InterfaceDefiniti | Type: RobotManager | Zoom: 100%

RobotAssemblyImplementation
- RobotAssembly
  - PredefinedTypes
  - RobotAssemblyAnalyses
  - RobotAssemblyBuild
  - RobotAssemblyComponentTypes
  - RobotAssemblyDeploymentPlans
  - RobotAssemblyEventChannelConfig
  - RobotAssemblyImplementationArtifacts
  - RobotAssemblyImplementations
    - HumanMachineInterfaceImplementation
    - ManagementWorkInstructionsImplement
    - PalletConveyorManagerImplementation
    - RobotAssemblyImplementation
    - RobotManagerImplementation
    - WatchSettingManagerImplementation
  - RobotAssemblyInterfaces
  - RobotAssemblyPackageConfigurations
  - RobotAssemblyPackages
  - RobotAssemblyTargets
  - TopLevelPackages

ProcessingStatus

SpecifyIdTag	
exclusiveProvider	True
single_destination	True
out_event_port_type	RTEventChannel

Figure 2: RobotManager Component Instance Inner View

Step 3: Highlight the *ProcessingStatus* OutEventPort model element, and then change the value of the attribute "**out_event_port_type**" from the default value "direct_connect" to "RTEventChannel".

After above three steps, the real-time event channel setup phase is finished for the particular interested event connection. You can continue to setup other interested event connections by the similar way illustrated above.

NOTE:

(1) As we can see, all a modeler needs to do is to modify the attribute on the OutEventPort (Event Publisher or Event Emitter) element. Nothing needs to be done on the InEventPort (Event Sink) elment.

(2) If a OutEventPort is a event publisher port and connects to multiple event sinks, then all the event connections with this publisher port will use the real-time event service, more particularly, all the connections will share the same real-time event channel.

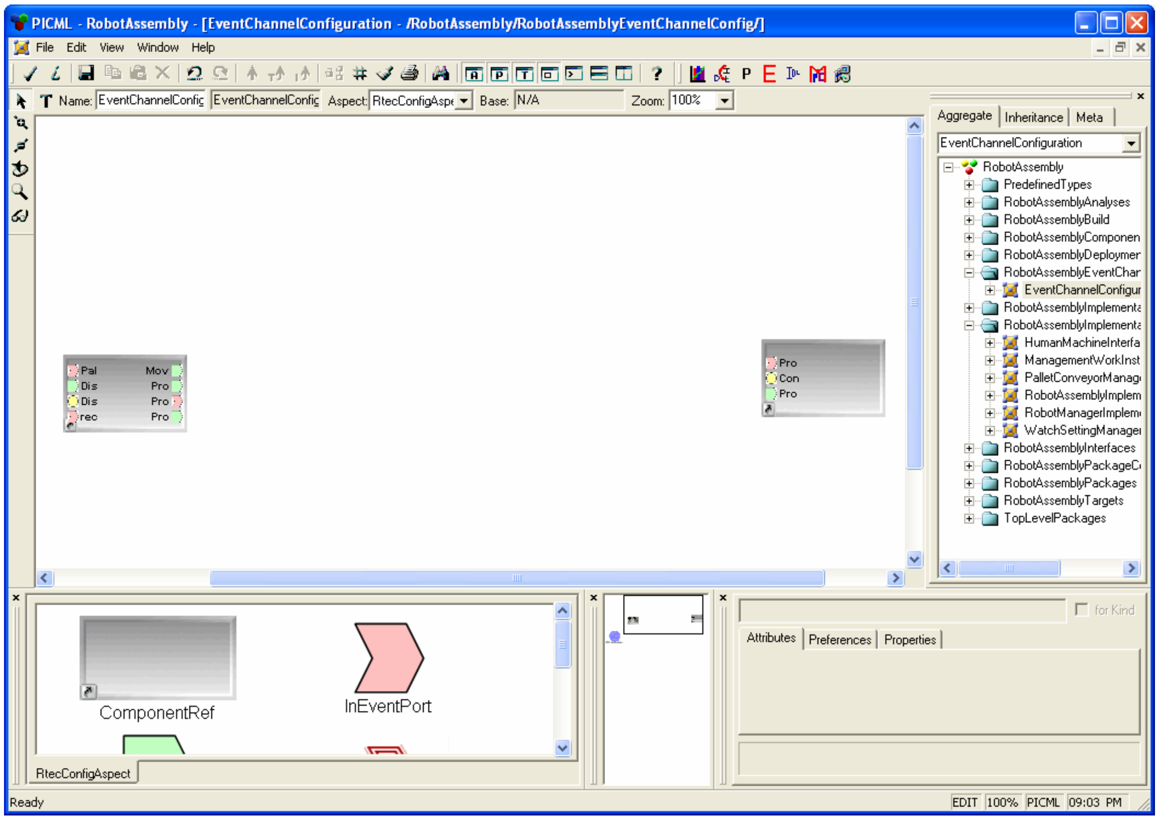
Configuration Phase

NOTE: You must finish Setup Phase before proceed. However, as we mentioned earlier, this configuration modeling phase is not a mandatory phase. Instead, if the modeler wants to set up QoS properties of the specified real-time event service, he/she could accomplish this through this phase as described below. The QoS properties could be specified in two different scopes, event channel scope and event port scope.

Step 1: Open PICML model and right click on the RootFolder. This opens up a dialog box. In this box choose Insert Folder. This should open up a menu with different folders roughly corresponding to each aspect in PICML. Choose the **EventChannelConfiguration** aspect. This step should create a folder named "NewEventChannelConfiguration" folder, and you can change the name to any meaningful name for the modeler.

Step 2: Right click on this newly created folder to insert an EventChannelConfiguration paradigm sheet, which corresponds to an EQAL model. Also, give this model the name to any meaningful name to the modeler.

Step 3: Create Component References. Go to the assembly implementation paradigm sheet as shown in Figure 1, copy the components which you want to setup by using the real-time event service. For example, in Setup Phase, we specified that we wanted to set up the real-time event service between components RobotManager and WatchSettingManager with the *ProcessingStatus* event port, then copy the RobotManager and WatchSettingManager components. (Please refer to GME manual for how to copy model elements.) After this is done, go back to the EventChannelConfiguration paradigm sheet, and then paste the components **as reference**. This is illustrated in Figure 3.

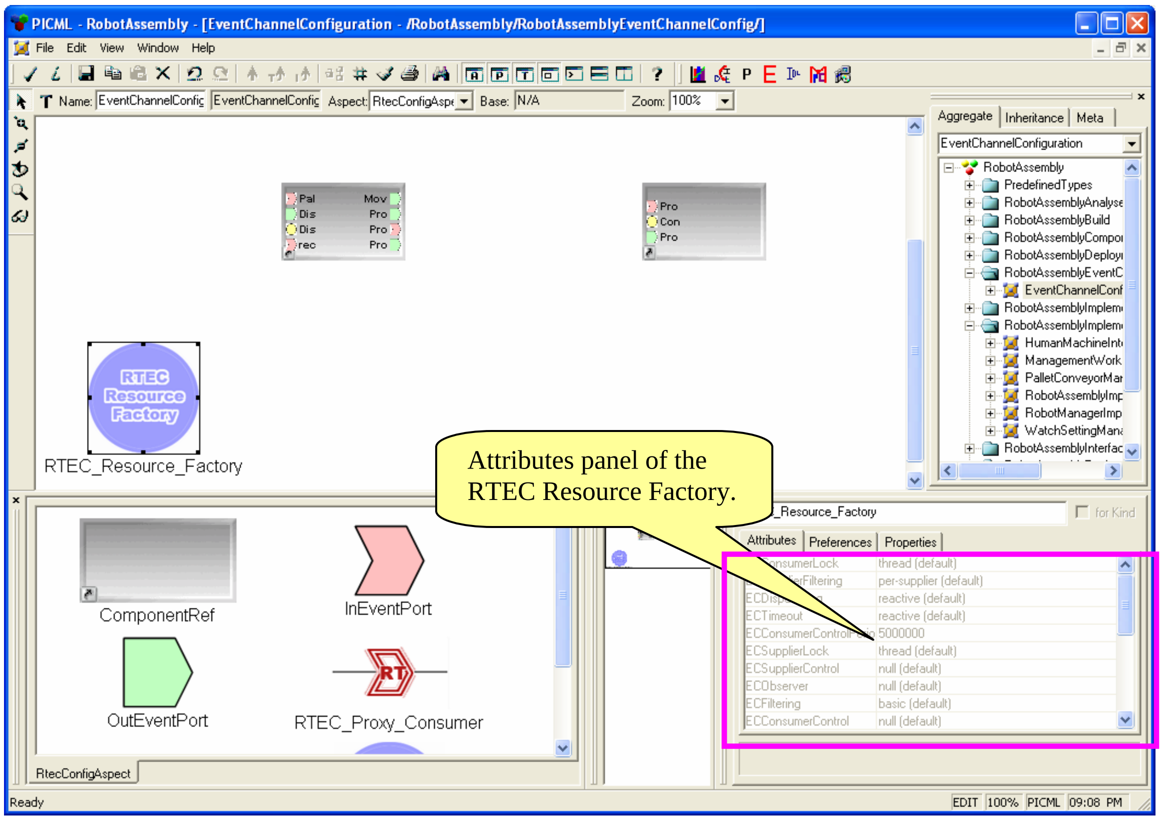

PICML - RobotAssembly - [EventChannelConfiguration - /RobotAssembly/RobotAssemblyEventChannelConfig/]

Name: EventChannelConfig   EventChannelConfig   Aspect: RtecConfigAspect   Base: N/A   Zoom: 100%

Figure 3: Copy component and paste as component references

Step 4: Set up and configure a reusable RT_Event_Channel_Resource_Factory. Drag and drop an RTEC_Resource_Factory modeling element into the EventChannelConfiguration paradigm sheet. Single click the modeling element, then all the attributes for this model atom will be shown in the attribute panel as illustrated in Figure 4. These attributes corresponds to the "per-channel" scope QoS strategy policy objects. EQAL interpreter then will use such specified attributes to generate the event channel service configuration files. *The EQAL constraints will prevent you from specifying inconsistent or invalid combinations of attributes*. Please make sure you invoke the constraint checker before invoking the interpreter.

NOTE: As for the semantic meaning of each attribute, please refer to some outside references, such as TAO Developer's Guide offered by OCI or Internet resources.



PICML - RobotAssembly - [EventChannelConfiguration - /RobotAssembly/RobotAssemblyEventChannelConfig/]
RTEC Resource Factory
RTEC_Resource_Factory
ComponentRef
InEventPort
OutEventPort
RTEC_Proxy_Consumer
RtecConfigAspect
Attributes panel of the RTEC Resource Factory.
Attributes
Preferences
Properties

onsumerLock	thread (default)
lierFiltering	per-supplier (default)
ECDisp	reactive (default)
ECTimeout	reactive (default)
ECConsumerControlP	5000000
ECSupplierLock	thread (default)
ECSupplierControl	null (default)
ECObserver	null (default)
ECFiltering	basic (default)
ECConsumerControl	null (default)

Figure 4: Attribute Panel for the RTEC Resource Factory

Step 5: Set up and configure event source and event sink port scope QoS properties. Add one RTEC_Proxy_Consumer for each event publisher/emitter port, and one RTEC_Proxy_Supplier for each event consumer port. This is illustrated as Figure 5 below.

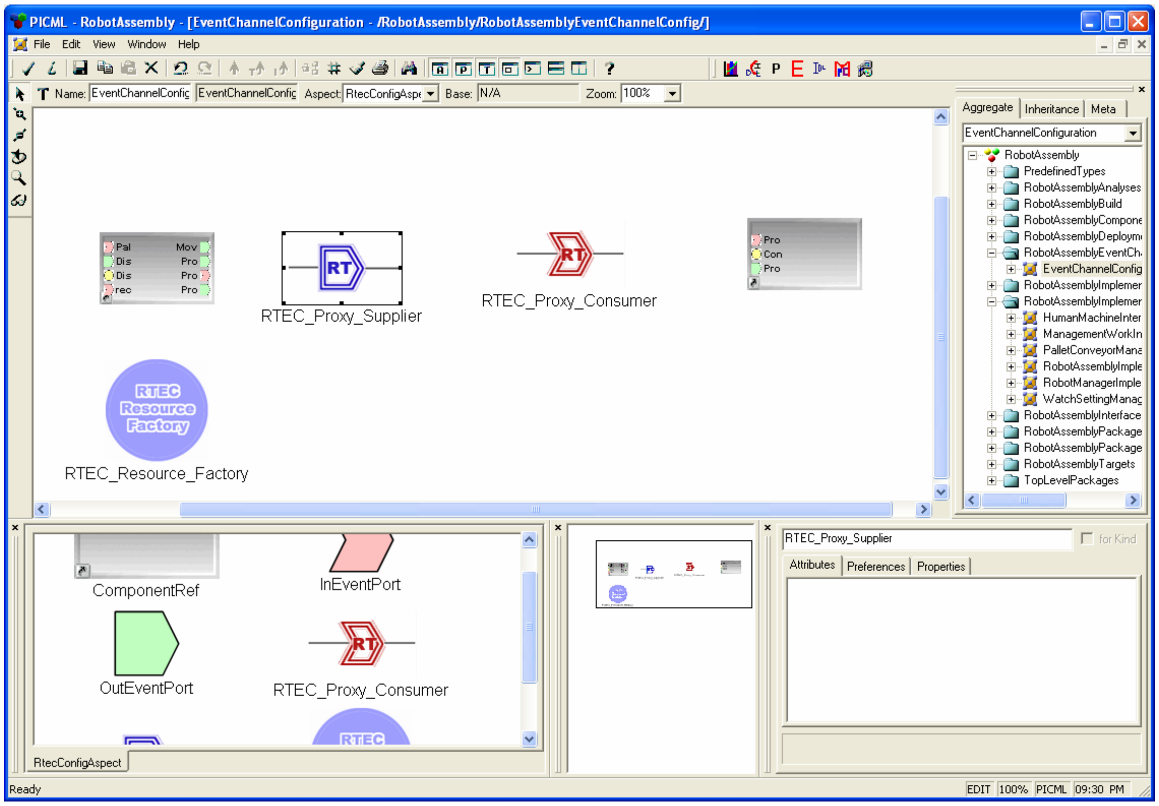
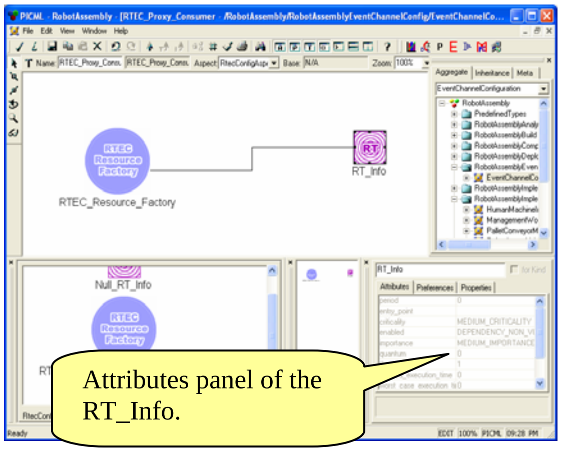


Figure 5: Adding Port Scope Proxies to the paradigm sheet

Please note that each port-level proxy above (RT Proxy Supplier and RT Proxy Consumer) is a model and could be configured by double clicking it to add other modeling atoms. Below Figure 6 and Figure 7 show how each event port proxy could be configured.

In Figure 6, RTEC_Resource_Factory_Reference is a reference to the RTEC Resource Factory we modeled in step 4. Add an RT_Info atom (or Null_RT_Info if no scheduling service is required) and set up the corresponding QoS attributes for the OutEvent Port.



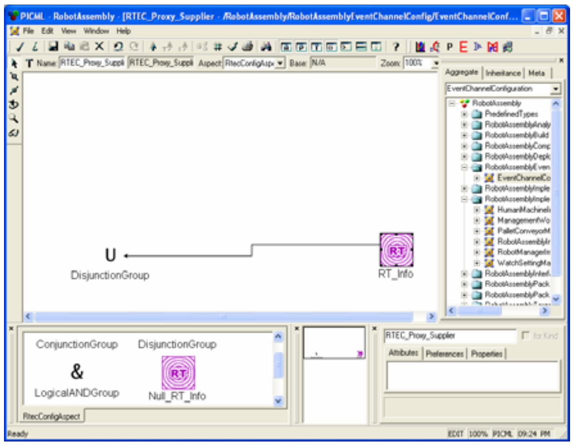


Figure 6: Proxy Consumer – For OutEvent Port

Figure 7: Proxy Supplier – For InEvent Port

In Figure 7, add an RT_Info atom (or Null_RT_Info if no scheduling service is required) and set up the corresponding QoS attributes for the InEvent Port, then connects this RT_Info atom with a disjunction group atom.

Step 6: Finalize and real-time event channel configuration connections. This step is straightforward, just as illustrated in Figure 8 below, connecting the publisher/emitter port to the RTEC_Proxy_Consumer RT_Info port, connecting the RTEC_Proxy_Consumer RTEC_Resource_Factory_Reference port to the RTEC_Proxy_Supplier RT_Info port, and finally connecting the RTEC_Proxy_Supplier Logical_Disjucntion Port to the event consumer port.

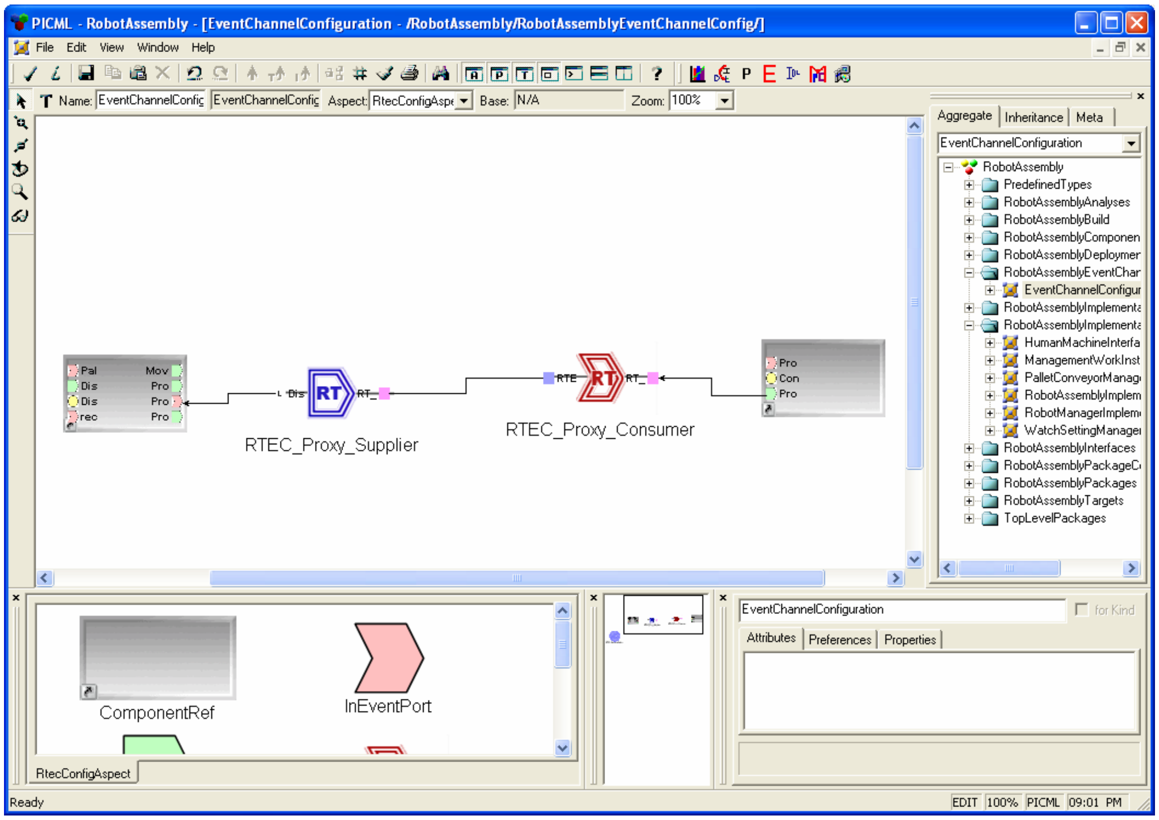
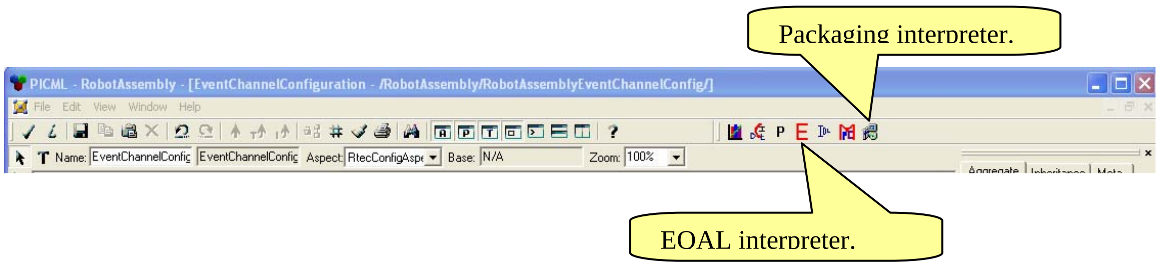


Figure 5: Finalize the Real-Time Event Channel Connections

Now your EQAL model is now complete and ready for interpretation. However, before invoking the EQAL interpreter, please explicitly check constraints before interpretation. (Invoke from menu File->Check->Check All).

4. EQAL Interpreters

Based on which modeling phases in Section 3 you have gone through, the interpreter invocation situation will be different.


Packaging interpreter.
PICML - RobotAssembly - [EventChannelConfiguration - /RobotAssembly/RobotAssemblyEventChannelConfig/]
File Edit View Window Help
Name: EventChannelConfig EventChannelConfig Aspect: RtecConfigAspe Base: N/A Zoom: 100%
Aggregate Inheritance Meta
EOAL interpreter.

If you only modeled Phase 1 (Event Service Setup Phase), then the only interpreter you want to invoke is the “**`Packaging interpreter`**”. Run the **`Packaging Interpreter`** will generate the “.ccd” file for each component. The .ccd file basically described the ports of each individual component, so if you specify a component to use the real-time event service, then the port type will be reflected in the generated .ccd file.

If you modeled both Phase 1 (Event Service Setup Phase) and Phase 2 (Event Service Configuration Phase), then besides invoking the “**`Packaging interpreter`**”, you also need to invoke the **`EQAL Interpreter.`** To invoke this interpreter, you must start from the EventChannelConfiguration paradigm sheet, and the interpreter will generate the XML descriptors and service configuration files for your event channel configuration. You are allowed to specify which directory to store the files.

A CPF file will be generated for each component event port, and a CONF file will be generated for each event channel configuration.

The CPF files contain the port RT and QoS properties. For example, the RobotManager-ProcessingStatus-SupplierQoS-.cpf file contains the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE properties SYSTEM "properties.dtd">
<properties>
  <struct name="\RobotManager-ProcessingStatus-SupplierQoS-"
type="ACEXML_RT_Info">
    <description>TAO Real-time Scheduler info for
RobotManager::ProcessingStatus</description>
    <simple name="port" type="string">
      <value>ProcessingStatus</value>
    </simple>
    <simple name="period" type="string">
      <value>0</value>
    </simple>
    <simple name="cached_execution_time" type="string">
      <value>0</value>
    </simple>
    <simple name="worst_case_execution_time" type="string">
      <value>0</value>
    </simple>
    <simple name="info_type" type="string">
      <value>OPERATION</value>
    </simple>
    <simple name="typical_execution_time" type="string">
      <value>0</value>
    </simple>
    <simple name="entry_point" type="string">
```

```
      <value></value>
    </simple>
    <simple name="criticality" type="string">
      <value>MEDIUM_CRITICALITY</value>
    </simple>
    <simple name="enabled" type="string">
      <value>DEPENDENCY_NON_VOLATILE</value>
    </simple>
    <simple name="importance" type="string">
      <value>MEDIUM_IMPORTANCE</value>
    </simple>
    <simple name="quantum" type="string">
      <value>0</value>
    </simple>
    <simple name="threads" type="string">
      <value>1</value>
    </simple>
  </struct>
</properties>
```

The CONF files contain the channel configurations. For example, the RTEC_Resource_Factory.conf file contains the following:

```
static EC_Factory " -ECConsumerLock null -ECSupplierFiltering null -
ECDispatching priority -ECTimeout priority -ECSupplierLock recursive -
ECSupplierControl reactive -ECObserver basic -ECFiltering prefix -
ECConsumerControl reactive -ECProxyPushConsumerCollectionSynch st -
ECProxyPushConsumerCollectionFlag rb_tree -ECScheduling priority"
```

After all the interpretation is done, you could deploy your system by the deployment/runtime framework on the component middleware.