Chapter 1

Model-Driven Development of Distributed Real-time and Embedded Systems

ABSTRACT

Despite advances in standards-based commercial-off-the-shelf (COTS) technologies, key challenges must be addressed before mission-critical distributed real-time and embedded (DRE) systems can be developed effectively and productively using COTS component-based software. For example, developers of DRE systems continue to use *ad hoc* means to select and compose their applications and middleware due to the lack of formally analyzable and verifiable building block components. This chapter shows how *Model-Driven Development* (MDD) techniques and tools can be used to specify, analyze, optimize, synthesize, validate, and deploy standards-compliant component middleware platforms that can be customized for the needs of next-generation DRE systems. Our results show how MDD techniques and tools have been integrated successfully with standards-based QoS-enabled component middleware to significantly improve the quality and productivity associated with developing mission-critical DRE systems.

Keywords: Model-Driven Development, component middleware, quality of service, distributed systems.

Chapter written by Douglas C. SCHMIDT, Krishnakumar BALASUBRAMANIAN, Arvind S.KRISHNA, Emre TURKAY, and Aniruddha GOKHALE, Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, USA
{schmidt,kitty,arvindk,turkaye,gokhale}@dre.vanderbilt.edu

1.1. INTRODUCTION

1.1.1. EMERGING TRENDS AND CHALLENGES

Over 90 percent of all microprocessors are now used for embedded systems, in which computer processors control physical, chemical, or biological processes and devices in real-time. Examples of real-time and embedded systems include mobile phones, pacemakers, and electronic games. Creating high quality software for real-time embedded systems has historically been hard due to memory constraints and processors with limited capacity, which precluded the use of modern software languages, tools, and techniques.

Due to advances in hardware technology, however, real-time and embedded systems now often have more memory and computational power. Moreover, individual computing nodes are increasingly combined to form distributed real-time and embedded (DRE) systems containing many processors that interoperate via networks and interconnects. Examples of DRE systems include hot rolling mill control systems, particle accelerators, electrical power grids, chemical plants, and air- traffic control systems, as shown in Figure 1.

Figure 1. Example DRE Systems

It is hard to design DRE systems that implement their required quality of service (QoS) capabilities, are dependable and predictable, and are parsimonious in their use of computing resources. It is even harder to build them on time and within budget. Moreover, due to global competition for market share and engineering talent, developers now also face the problem of delivering new products in compressed time-frames. It is therefore essential that the production of DRE systems take advantage of languages, tools, platforms, and methods that enable higher levels of software productivity by moving from a third-generation language *programming-centric* paradigm to a component-based *assembly-centric* paradigm.

DRE systems have historically been developed in a hard-coded manner, e.g., with dedicated software written for specific types of hardware, using unstructured "spaghetti" designs and code. This approach has yielded stove-piped and proprietary solutions, such as legacy avionics and radio systems shown in Figure 2, that are tedious, error-prone, and costly to develop, validate, and evolve. In particular, small changes to software structure or functionality in these tightly coupled systems often led to large (negative) impacts on DRE system QoS and maintainability.

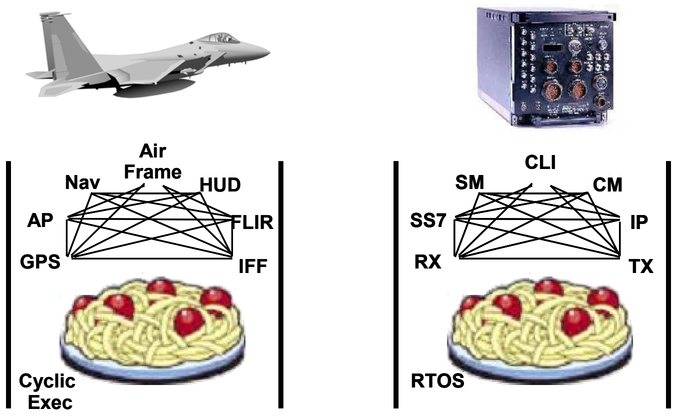


Figure 2. *Unstructured DRE Systems Based on Stove-piped and Proprietary Software*

1.1.2. A PARTIAL SOLUTION: QOS-ENABLED COMPONENT MIDDLEWARE

Over the past decade, standards-based QoS-enabled distributed computing middleware, such as Real-time CORBA [CORBA:02b] and Real-time Java [RTSJ:00], has emerged to reduce the complexity of DRE systems. This type of middleware simplifies the development of DRE systems by factoring out reusable mechanisms and services from application code, thereby off-loading many tedious and error-prone aspects of the software process from developers of vertical applications to developers of horizontal middleware platforms. Figure 3 illustrates examples of middleware-based DRE systems, such as modern avionics mission computing [Sharp:03] and software-defined radio [SCA:01] systems, where application developers are shielded from low-level, tedious, and error-prone computing and communication details. Moreover, middleware amortizes software lifecycle costs by leveraging previous development expertise and capturing implementations of key patterns in reusable frameworks, rather than rebuilding them manually for each use.

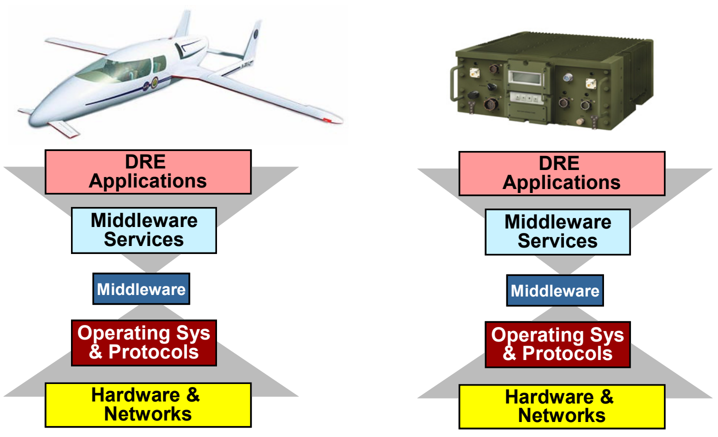


Figure 3. *Structured DRE Systems Based on Reusable and Standard Middleware*

During the past decade, a substantial amount of R&D effort has focused on developing component middleware [Szyperski:02, Heineman:01], which enables reusable services to be composed, configured, and installed to create applications rapidly and robustly. The CORBA Component Model (CCM) [CorbaComponents:02] is standard component middleware that extends earlier versions of CORBA [CORBA:02a] to support the concept of components and establishes standards for specifying, implementing, packaging, assembling, and deploying components.

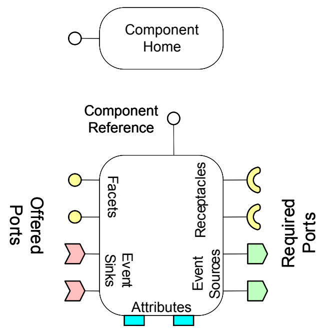


Figure 4. *Ports in the CORBA Component Model*

A **component** in CCM is the primary unit of implementation, reuse, and composition that exposes a set of **ports**, named interfaces and connection points that components use to collaborate with each other. Ports include the interfaces and connection points shown in Figure 4 and described below:

- **Facets**, which define a named interface and an implementation that synchronously services operation invocations called from other components.
- **Receptacles**, which provide named connection points to facets provided by other components.
- **Event sources and event sinks**, which indicate a willingness to exchange event messages with other components asynchronously.

A unique component **home,** which is a factory, creates and manages each component instance.

Figure 5 illustrates the layered architecture of CCM. A **container** provides the run-time environment for one or more components that manages various pre-defined hooks and strategies, such as persistence, event notification, transaction, and security, used by the component(s). Developer-specified metadata expressed in XML instruct CCM deployment mechanisms on how to control the lifetime of these containers and the components they manage. **A component assembly** is a *virtual* component consisting of metadata that describes how components are grouped together to form higher-level units. Each component's metadata describes the features it provides (*e.g.*, its interfaces and properties) or the features that it requires (*e.g.*, its dependencies). A **component server** is an abstraction that is responsible for aggregating *physical* entities (*i.e.*, implementations of component instances) into *logical* entities (*i.e.*, distributed application services and subsystems).

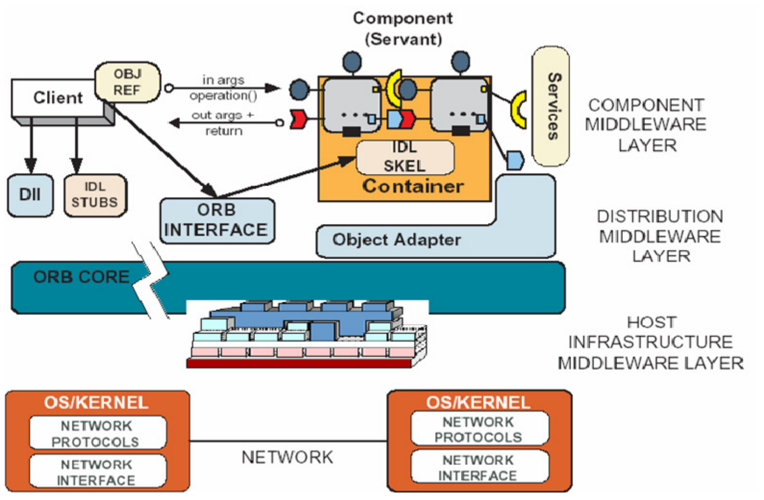


Figure 5. The Layered CCM Architecture

In addition to the run-time building blocks outlined above, CCM also standardizes component implementation, packaging, and deployment mechanisms. **Packag-**

ing involves grouping the implementation of component functionality – typically stored in dynamic link libraries (DLL) – together with other metadata that describes salient properties of this particular implementation. The CCM **Component Implementation Framework (CIF)** helps generate the component implementation skeletons and persistent state management automatically using the **Component Implementation Definition Language (CIDL)**, which associates component interfaces with their executor implementations.

In conjunction with colleagues at Washington University [Schmidt:04d], we have developed the *Component-Integrated ACE ORB* (CIAO) [Schmidt:03b], which is a real-time implementation of CCM. CIAO extends our previous work on The ACE ORB (TAO) [Schmidt:97] by providing more powerful component-based abstractions using the specification, validation, packaging, configuration, and deployment techniques defined by the OMG CCM and Deployment and Configuration (D&C) [DandC:03] specifications. Moreover, CIAO integrates the CCM capabilities outlined above with TAO's Real-time CORBA [RTCorba:02] features, such as thread-pools and client-propagated and server-declared policies.

1.1.3. RESOLVING KEY CHALLENGES OF COMPONENT-BASED DRE SYSTEMS WITH MODEL-DRIVEN DEVELOPMENT

Despite advances in standards-based QoS-enabled component middleware, however, significant challenges remain that make it hard to support large-scale DRE systems in domains (such as shipboard combat systems and supervisory control and data acquisition (SCADA) systems) requiring stringent support for multiple QoS properties. Key unresolved challenges include:

- **Lack of tools for effectively composing DRE systems from components.** DRE component middleware enables application developers to develop individual QoS-enabled components and package them into assemblies that form complete DRE systems. Although this approach supports the use of "plug and play" components, DRE system integrators still face the challenge of composing the right set of compatible components that will deliver the desired semantics and QoS to applications.
- **Lack of tools for configuring component middleware.** In QoS-enabled component middleware frameworks, many attributes and parameters of application and middleware components are configured at various stages of the development lifecycle. Manual techniques for ensuring that these parameters are semantically consistent throughout a large-scale DRE system are tedious and error-prone, however. Moreover, manual techniques often have no formal basis for validating and verifying that the configured middleware will deliver the end-to-end QoS requirements of applications throughout a DRE system.
- **Lack of tools for automating the deployment of DRE system components onto heterogeneous target platforms.** The component assemblies described above need to be deployed in a distributed target environment before applica-

tions can run. DRE system integrators must therefore perform the complex task of mapping individual component assemblies onto specific nodes of the target environment. This mapping must ensure that a particular deployment meets the end-to-end QoS requirements given the capabilities of the nodes in the target environment.

This chapter describes how we are addressing the challenges described above using *Model-Driven Development* (MDD) techniques and tools. MDD is an emerging paradigm [Greenfield:04] that helps resolve software development and validation challenges encountered in development of component middleware and DRE systems by combining (1) *domain-specific modeling languages* (DSMLs), which provide programming notations that formalize the process of specifying application logic and QoS-related requirements, (2) *metamodeling*, which helps to automate the definition of type systems that precisely express key characteristics and constraints associated with DSMLs for particular application domains, such as software-defined radios, avionics mission computing, and total ship computing environments, and (3) *model transformations and code generation* that automate and ensure the consistency of software implementations via analysis information associated with functional and QoS requirements captured by models of domain-specific structure and behavior.

We have developed an MDD tool-suite called *Component Synthesis using Model Integrated Computing* (CoSMIC) [Schmidt:04a], which is an integrated collection of open-source[1] DSMLs that support the development, configuration, deployment, and evaluation of component-based DRE systems. The CoSMIC MDD tools can be used to specify requirements, compose DRE applications and their supporting infrastructure from the appropriate set of middleware components, synthesize the metadata, collect data from application runs, and analyze the collected data to re-synthesize the required metadata. These activities can be performed via an iterative process until the QoS constraints are satisfied end-to-end.

1.1.4. CHAPTER ORGANIZATION

The remainder of this chapter is organized as follows: Section 1.2 describes the structure and functionality of a component-based video distribution system we developed as a case study; Section 1.3 presents an overview of CoSMIC and describes how we have applied its MDD tools to address key challenges of applying component-based middleware to our case study; Section 1.4 compares our work on CoSMIC with related work; and Section 1.5 presents concluding remarks.

[1] CoSMIC's MDD tools are open-source and available for download at www.dre.vanderbilt.edu/cosmic.

1.2. OVERVIEW OF VIDEO DISTRIBUTION CASE STUDY

To motivate and explain the features of CoSMIC we use a running example of a representative DRE system shown in Figure 6. This system is designed for emergency response situations (such as disaster recovery efforts stemming from floods, earthquakes, hurricanes) and consists of interacting subsystems with a variety of DRE QoS requirements. Our focus in this chapter is on the unmanned aerial vehicle (UAV) video distribution portion of this system, which is used to monitor terrain for flood damage, spot survivors that need to be rescued, and assess the extent of damage. The UAVs transmit this imagery to various other emergency response units, including the National Guard, law enforcement agencies, health care systems, fire-fighting units, and utility companies.

Figure 6. UAV Emergency Response System

Developing and deploying emergency response systems is hard. For example, there are multiple modes of operations for the UAVs, including aerial imaging, survivor tracking, and damage assessment. Each mode is associated with a different set of QoS requirements. For example, a key QoS criterion is the latency requirements in sending images from the UAVs to ground stations under varying bandwidth availability. Similar QoS requirements occur in the traffic management, rescue missions, and fire-fighting operations.

In conjunction with colleagues at BBN Technologies [Schantz:04], we have developed a prototype of the video distribution system described above using the CCM and Real-time CORBA capabilities provided by CIAO and TAO, respectively. The components in the video distribution application are shown in Figure 7. Each UAV is associated with a stream of images. Each image stream is composed of `Sender`, `Qosket`, and `Receiver` components. `Sender` components are responsible for collecting the images from each image sensor on the UAV. Each `Sender` component passes the images it receives to a chain of `Qosket` [Schmidt:03b] components that perform operations on the images to ensure that the QoS requirements are satisfied.

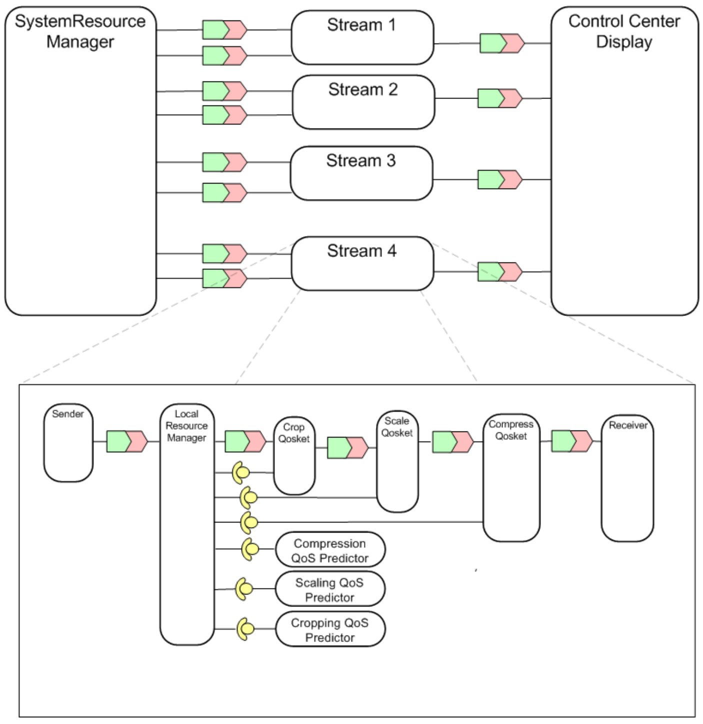


***Figure** 7. Emergency Response System Components*

`Qosket` components in our video distribution application include:

- `CompressQosket`, which compresses images passed along each stream to reduce the bandwidth required to transmit the images.
- `ScaleQosket`, which is scales images to reduce the bandwidth required to send images.

- CropQosket, which crops images so that only interesting portions of a large image is transmitted to receivers.
- PaceQosket, which paces the transmission of images in an ordered fashion to avoid bursty network traffic.
- DiffServQosket, which sets DiffServ codepoints on routers in the path between each Sender-Receiver pair.

The final Qosket component in the chain then passes the images to a Receiver component, which collects the images and renders them on a display in the control room of the emergency response team.

Each Sender, Receiver, and the various Qosket components pass images via CCM event source and sink ports. There are also manager components that define policies, such as the relative importance of the different mission modes of each UAV. These policies in turn modify existing resource allocations by the Qosket components. For example, the global SystemResourceManager component monitors resource allocation across all the UAVs that are operational at any moment. It is responsible for communicating policy decisions from the control center to each UAV by triggering mode changes. The per-stream LocalResourceManager component uses the facets exposed by the Qosket components to instruct the Qosket components to adapt their internal QoS requirements according to the mode in which a UAV is currently operating.

The component-based implementation of the video distribution application has ~18 component types, which results in ~18 C++ classes. Each Qosket component has ~4 ports that must be implemented by application developers. A typical deployment of the video distribution application employs ~6 UAVs contributing to 6 image streams with ~5 different components per stream. Since each component in a stream receives images and pushes them along the pipeline, each component participates in at least two connections[2] to send/receive the images and in one connection for controlling QoS properties. As a result, there are ~15 connections per stream, resulting in ~90 such connections related to the image distribution. Each LocalResourceManager also receives policy and resource allocation events from the SystemResourceManager component, resulting in about ~100 connections in a typical deployment scenario. Using CIAO and standard CCM and D&C capabilities, each connection must be hand-written in the XML deployment descriptor files, while also being careful to ensure that the ~30 component instances are each assigned unique identifiers.

[2] These connections are not network connections, but rather represent logical interconnections between component ports.

1.3. APPLYING COSMIC TO ADDRESS VIDEO DISTRIBUTION NEEDS

As discussed in [Schmidt:04d], the use of CIAO-based QoS-enabled component middleware to develop the video distribution system described in Section 1.2 significantly improved the software quality and flexibility of an earlier prototype [Schmidt:03a] of this application that was developed using the previous generation of distributed object computing (DOC) middleware based on TAO. In the absence of support from MDD tools, however, the following challenges remain unresolved when using component middleware [Schmidt:04f]:

- The need to define consistent component interfaces,
- The need to specify valid interactions and connections between components,
- The need to generate valid component deployment descriptors,
- The need to configure the component and the underlying middleware and platform,
- The need to evaluate the chosen configuration to ensure QoS satisfaction,
- The need to ensure that requirements of components are met by target nodes where components are deployed, and
- The need to validate that changing system structure and/or behavior does not leave it in an inconsistent state.

The lack of simplification and automation in resolving the challenges outlined above can significantly hinder the effective transition to – and adoption of – component middleware technology to develop DRE systems.

This section presents an overview of CoSMIC and describes how we have applied its MDD tools to address the challenges of applying component-based middleware to our case study described in Section 1.2.

1.3.1. OVERVIEW OF COSMIC AND GME

CoSMIC is an integrated set of MDD tools that support the development, configuration, and deployment of component-based DRE systems. The MDD tools provided by CoSMIC address key lifecycle phases involved in developing component-based DRE systems, as shown in Figure 8 and described below:

- **Specification and implementation**, which involves defining, partitioning, and implementation of application functionality as standalone components.
- **Packaging**, which involves, bundling a suite of software binary modules and metadata representing application components.
- **Installation**, which involves populating a repository with the packages required by the application.

- **Configuration**, which involves configuring the middleware with the appropriate parameters to satisfy the functional and systemic requirements of application.
- **Planning**, which involves making appropriate deployment decisions, including identifying the entities (such as CPUs) of the target environment where the packages will be deployed.
- **Preparation**, which involves moving the implementation artifacts to the identified entities of the target environment.
- **Launching**, which involves triggering the installed binaries and the application to a ready state.
- **QoS assurance and adaptation**, which involves runtime reconfiguration and resource management to maintain end-to-end QoS.

A comprehensive discussion of CoSMIC appears in [Schmidt:04a]. Below we discuss the portions of CoSMIC necessary to understand how we applied it to the component-based video distribution system described in Section 1.2.

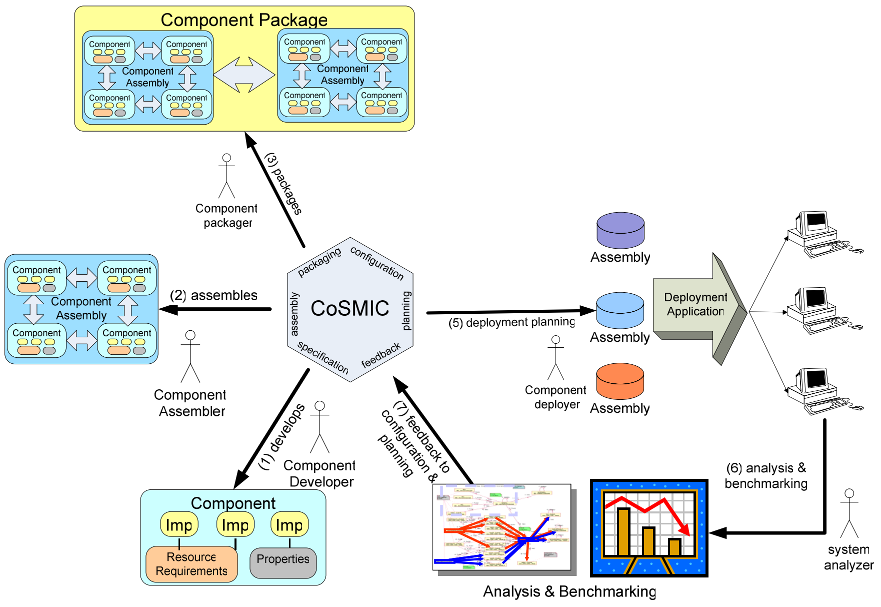


Figure 8. *MDD Capabilities Supported by CoSMIC*

All MDD tools in CoSMIC are developed using the Generic Modeling Environment [GME:01], which is an open-source[3] MDD environment that provides a visual

[3] GME is available in binary and open-source from www.isis.vanderbilt.edu/Projects/gme/.

interface to simplify the development of DSMLs. GME contains a metamodeling environment that supports the definition of *paradigms*, which are type systems that describe the roles and relationships between elements in a particular domain. GME has a flexible object-oriented type system that supports inheritance and instantiation of elements of DSMLs. It also provides an integrated constraint definition and enforcement module based on OMG's Object Constraint Language (OCL) [OCL:03], which enables the definition of rules that must be adhered to by elements of models built using a particular DSML.

Figure 9 illustrates the GME metamodel for CCM used in CoSMIC. This metamodel uses UML structure diagrams and OCL constraints to define the abstract syntax, static semantics, and visualization of CCM elements such as components, ports, homes, and containers. The dynamic semantics of CCM are implemented via GME *interpreters*, which traverse the graphical hierarchy of elements programmed by application modelers to generate various types of output from model elements, including C++ code, XML package and assembly descriptors, and component property configurations.

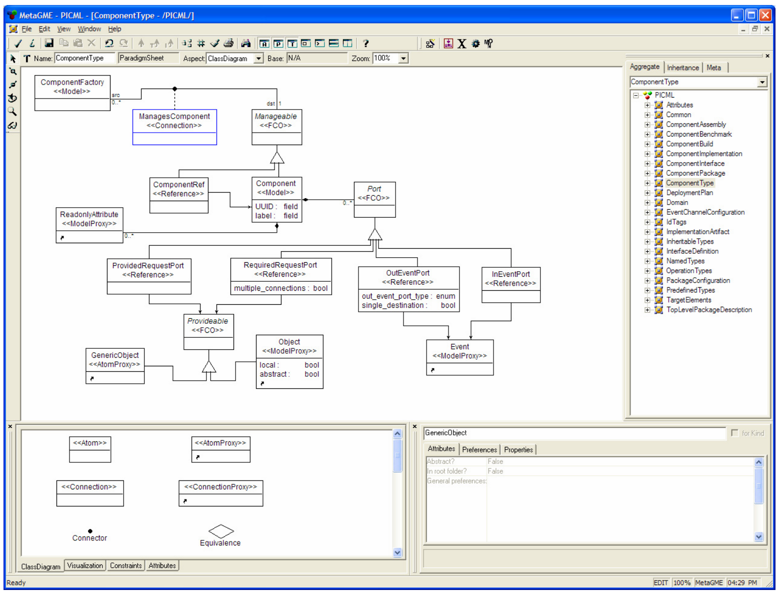

MetaGME - PICML - [ComponentType - /PICML/]
ComponentFactory <<Model>>
ManagesComponent <<Connection>>
Manageable <<FCO>>
ComponentRef <<Reference>>
Component <<Model>>
UUID : field
label : field
Port <<FCO>>
ReadonlyAttribute <<ModelProxy>>
ProvidedRequestPort <<Reference>>
RequiredRequestPort <<Reference>>
multiple_connections : bool
OutEventPort <<Reference>>
out_event_port_type : enum
single_destination : bool
InEventPort <<Reference>>
Provideable <<FCO>>
GenericObject <<AtomProxy>>
Object <<ModelProxy>>
local : bool
abstract : bool
Event <<ModelProxy>>
<<Atom>>
<<AtomProxy>>
<<Connection>>
<<ConnectionProxy>>
Connector
Equivalence
ClassDiagram
Visualization
Constraints
Attributes
Aggregate
Inheritance
Meta
PICML
Ready

Figure 9. A GME-based Metamodel for CCM

The elements in a GME-based DSML represent the elements of the domain in a more intuitive manner than is possible via third-generation programming languages. Application developers use GME to create models that are instances of these modeling language paradigms within the same environment. GME supports facilities to plug-in analysis and synthesis tools that operate on the models.

Figure 10 illustrates how CoSMIC's modeling environment created by GME can be used to model the connections between components in a CCM assembly. Application modelers use the CoSMIC environment to address key lifecycle phases involved in developing component-based DRE systems, as shown in Figure 8. CoSMIC's interpreters then generate various types of output associated with the packaging, configuration, and planning MDD tools described in the remainder of this section.

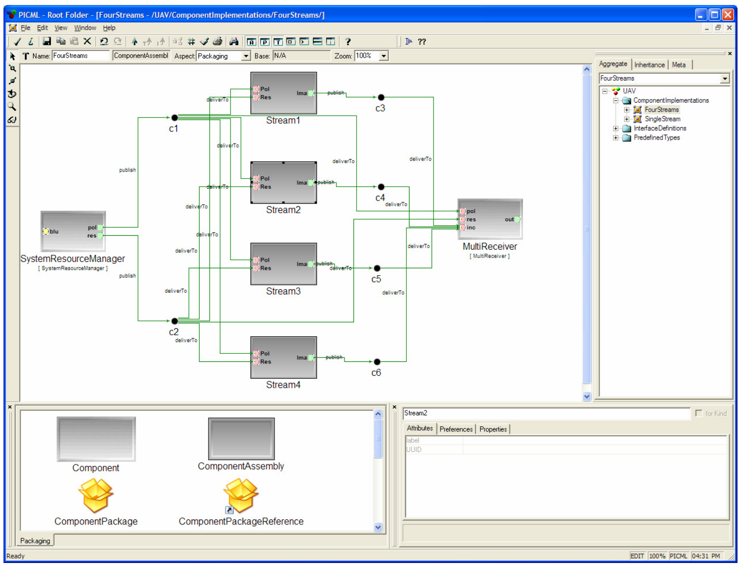


Figure 10. A CoSMIC Model of a CCM Assembly

1.3.2. APPLYING COSMIC TO THE PACKAGING PHASE

In component-based systems, application components and their associated metadata, which specify the connections between component ports, are packaged together into assemblies. Different assemblies in a package can be tailored to deliver different end-to-end QoS behaviors and/or algorithms. Large-scale DRE systems may require creation of assemblies containing hundreds or thousands of components, which leads to the following complexities:

- **Inherent complexities**, which involve ensuring syntactic and semantic compatibility. For example, it is essential to ensure that ports of the components connected in an assembly have matching types.

- **Accidental complexities**, which stem from handcrafting XML files that describe the component metadata such as the hundreds of connections between components in the assemblies. For example, XML files that describe assemblies are often very large, even for relatively simple groups of connected components.

To address these challenges we developed *the Platform-Independent Component Modeling Language* (PICML) [Schmidt:04f]. PICML is a GME-based DSML that provides capabilities to handle complex component engineering tasks, such as multi-aspect visualization and manipulation of components and the interactions of their subsystems, component deployment planning, and hierarchical modeling and generation of component assemblies. Figure 11 illustrates how PICML assists component developers with the packaging phase in the context of our video distribution application.

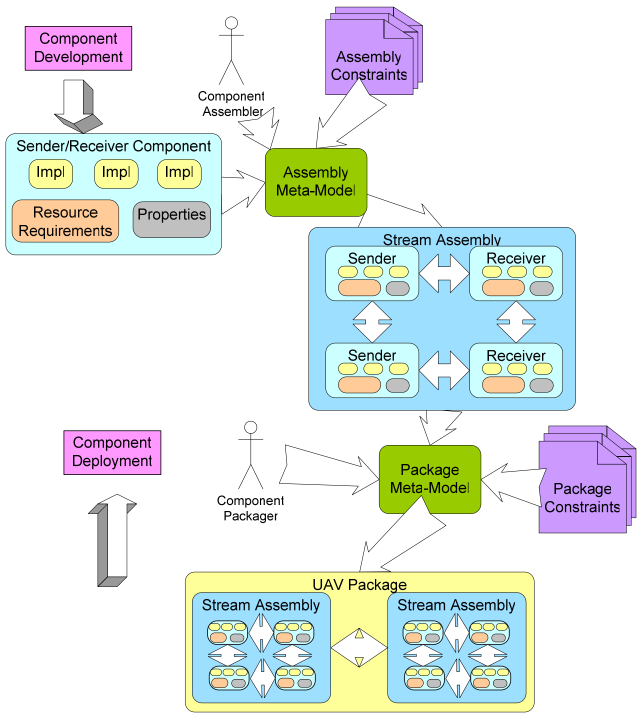
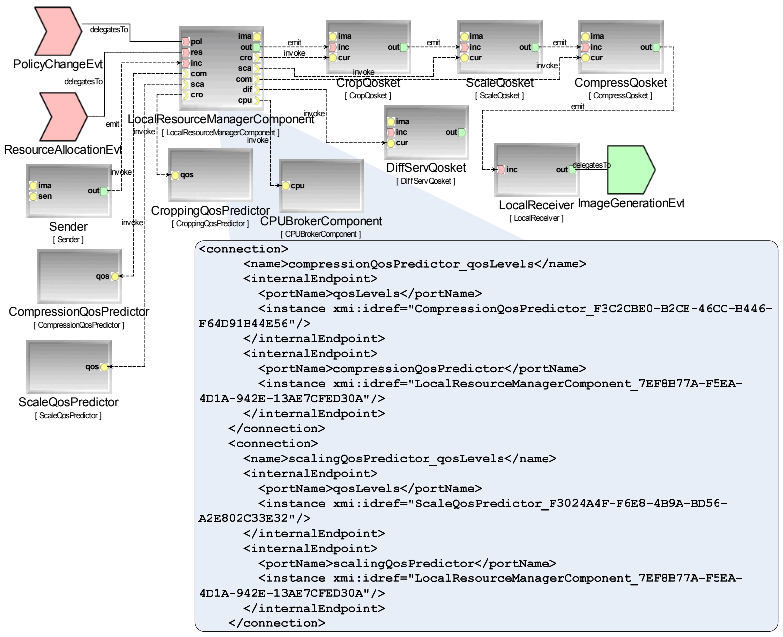


Figure 11. Activities in the Packaging phase

During component development, application developers use CORBA IDL 3.x to specify the interfaces of the different components, such as `Sender` and `Receiver`. The information stored in these IDL interfaces is imported into PICML in one of two ways, i.e., IDL files can be directly imported into PICML using the IDL importer interpreter of PICML or PICML graphical input interface can be used to model the elements that are present in the IDL files manually.

After the information about component interfaces is captured in a PICML model, application modelers can connection various components visually. In our video distribution application, for example, PICML is used to model the connection between the event source and event sink ports of `Sender` and `Receiver` components to form a *component assembly*. The semantic rules that determine the valid connections between components are enforced during component assembly by constraints defined in PICML's metamodel. In our video distribution application, assemblies that represent a single stream of image data between every `Sender-Receiver` pair are obtained at the end of this composition process.

PICML supports hierarchical composition of component assemblies into higher-level assemblies and grouping of multiple these assemblies into component packages. In our video distribution application, for example, multiple instances of image stream assemblies can be composed to form a complete video distribution application assembly. Hierarchical assemblies allow multiple instantiation of the same assembly type, which reduces the complexity of changes that would occur if each assembly were defined separately. A hierarchical assembly is a *logical* assembly, i.e., the ports of a component present in one assembly are connected directly to the ports of the component at the other end of each connection in a hierarchical component assembly. The logical hierarchy feature in PICML therefore does not impose any extra run-time overhead on component applications.



```
<connection>
      <name>compressionQosPredictor_qosLevels</name>
      <internalEndpoint>
        <portName>qosLevels</portName>
        <instance xmi:idref="CompressionQosPredictor_F3C2CBE0-B2CE-46CC-B446-
F64D91B44E56"/>
      </internalEndpoint>
      <internalEndpoint>
        <portName>compressionQosPredictor</portName>
        <instance xmi:idref="LocalResourceManagerComponent_7EF8B77A-F5EA-
4D1A-942E-13AE7CFED30A"/>
      </internalEndpoint>
    </connection>
    <connection>
      <name>scalingQosPredictor_qosLevels</name>
      <internalEndpoint>
        <portName>qosLevels</portName>
        <instance xmi:idref="ScaleQosPredictor_F3024A4F-F6E8-4B9A-BD56-
A2E802C33E32"/>
      </internalEndpoint>
      <internalEndpoint>
        <portName>scalingQosPredictor</portName>
        <instance xmi:idref="LocalResourceManagerComponent_7EF8B77A-F5EA-
4D1A-942E-13AE7CFED30A"/>
      </internalEndpoint>
    </connection>
```

Figure 12*. XML Descriptors generated by PICML*

After PICML has created component assemblies for our video distribution application, its packaging interpreter is executed to generate the metadata needed to deploy CCM applications, such as our video distribution application. As shown in Figure 12, this metadata includes the list of implementation artifacts associated with each component instance, the list of connections between the different component instances, the organization of the application into different levels of hierarchy, and the default properties with which each component instance is initialized. PICML's packaging interpreter generates the different types of metadata in the form of XML descriptors that are tedious and error-prone to write manually. This metadata is used by the CIAO component middleware uses to drive the deployment of the complete applications.

By automatically generating the artifacts needed to deploy applications, therefore, PICML enforces the *correct-by-*construction paradigm in component-based application development.

1.3.3. APPLYING COSMIC TO THE CONFIGURATION PHASE

Component middleware is often characterized by a large configuration space [Schmidt:04c], which includes various alternatives for (de)marshaling, event/request de-multiplexing, connection management, concurrency, synchronization, and transport protocols, as shown in Figure 13.

These alternatives can be selected at one or more configuration points. Common configuration points include during (1) *component development*, where default values for these mechanisms can be specified, (2) *application integration*, where component defaults can be overridden with domain-specific defaults, and (3) *application deployment*, where domain-specific defaults can be overridden based on the actual capabilities of the target system. The configuration process is thus the phase during component-based software development that maps known variations in the application requirements space to known variations in the middleware solution space.

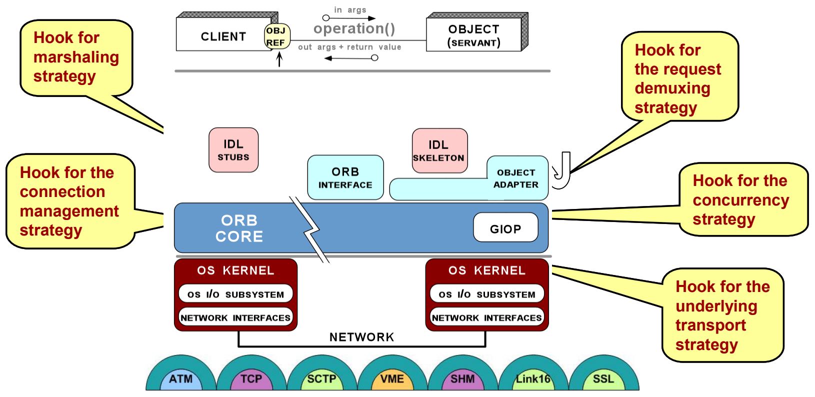


Figure 13. Middleware Configuration Space

To address the configuration-related challenges described above, we developed the *Options Configuration Modeling Language* (OCML) [Schmidt:04e]. OCML is a GME-based DSML that simplifies the specification and validation of complex DRE middleware and application configurations. Figure 14 illustrates the process of configuring middleware and applications using OCML and PICML. This three-layered process of configuring middleware and applications is explained below.

The *metamodeling layer* is where the middleware configuration space is defined with the options, values for the options, and interdependencies of the options Middleware developers use OCML to design the CIAO options model. The constraints defined in CIAO's options model can have a dependency hierarchy, such as a specific value for an option that may depend on other options having specific values. OCML then uses the CIAO option model to generate a CIAO-specific *configurator*, which embodies the rules and dependencies among the various CIAO options.

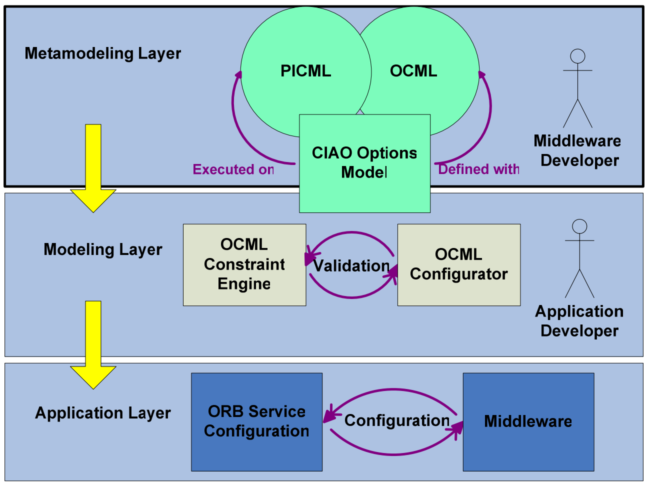
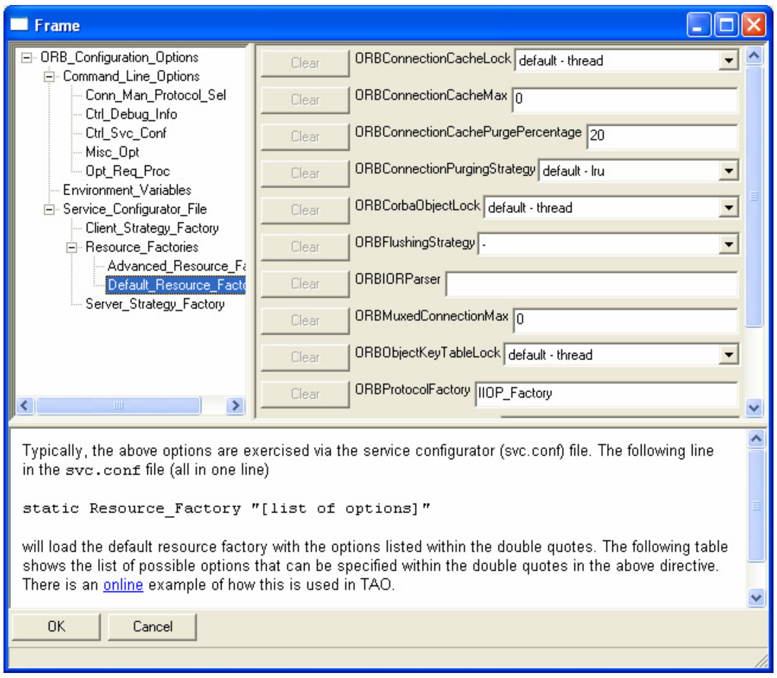


Figure 14. OCML Process View

The CIAO options model is related with PICML and OCML metamodels as shown in Figure 14 and described below:

- The CIAO Options Model is built using the OCML metamodel, which is intentionally designed to be general-purpose, i.e., the data types used to define the options are the fundamental data types (such as string and integer) so that they can define configurations for most applications. Although OCML is a generic modeling tool for configuring many types of applications and infrastructure, it is particular useful for highly configurable middleware frameworks, such as CIAO [Schmidt:04c].
- We have integrated the CIAO Options Model with the PICML metamodel. Users can therefore configure the underlying CIAO middleware by executing OCML on model instances with specific PICML elements.



ORBConnectionCacheLock: default - thread
ORBConnectionCacheMax: 0
ORBConnectionCachePurgePercentage: 20
ORBConnectionPurgingStrategy: default - lru
ORBCorbaObjectLock: default - thread
ORBFlushingStrategy: -
ORBIORParser:
ORBMuxedConnectionMax: 0
ORBObjectKeyTableLock: default - thread
ORBProtocolFactory: IIOP_Factory

Typically, the above options are exercised via the service configurator (svc.conf) file. The following line in the `svc.conf` file (all in one line)

```
static Resource_Factory "[list of options]"
```

will load the default resource factory with the options listed within the double quotes. The following table shows the list of possible options that can be specified within the double quotes in the above directive. There is an online example of how this is used in TAO.

Figure 15. *OCML-generated Configurator for CIAO*

The *modeling layer* is where application developers use the CIAO-specific configurator generated by OCML to customize the middleware according to specific application needs. As shown in Figure 15, the CIAO-specific configurator provides a wizard-like user interface that enables application developers to specify a set of values for different option configurations. These configurations are validated against the constraints defined in CIAO's options model using OCML's constraint engine. Application developers therefore cannot define inconsistent configurations that would be semantically meaningless or would yield indeterminate behavior for the CIAO middleware.

Whenever application developers modify an option parameter, the OCML constraint engine manipulates related options so that the constraint validation will succeed. If a value change request is invalidated by a previous value assignment by the user, the constraint validation will fail and the value change request will not succeed. This capability allows middleware developers to define higher-level options that control the values of many other options and broaden the middleware's configuration space, while also enabling application developers to fine-tune option to meet their needs.

The *application layer* illustrates how the OCML configuration process affects the CIAO middleware at run-time. For our video distribution example, application developers can use the generated CIAO-specific configurator to tailor the middleware for each group of components that will be collocated on a particular node. This configurator generates a CIAO-specific *service configuration* file, which is read by CIAO at the initialization time to configure its behavior, such as strategies for concurrency, communication protocols, debugging, and logging.

1.3.4. APPLYING COSMIC TO THE PLANNING PHASE

The planning phase is where component integrators must make appropriate deployment decisions, including identifying the nodes (e.g., computing devices) of the target environment where assembly packages will be deployed. Figure 16 illustrates the key activities and decisions during the planning stage, including selection of the appropriate (1) package to deploy on selected target, (2) target platform to deploy the packages, and (3) allocation of resources on target platforms. These decisions are encoded within a deployment plan, which is an XML-based descriptor file that describes a mapping of a configured application into a domain, including mapping monolithic implementations to nodes, connections to interconnects and bridges, and requirements to resources.

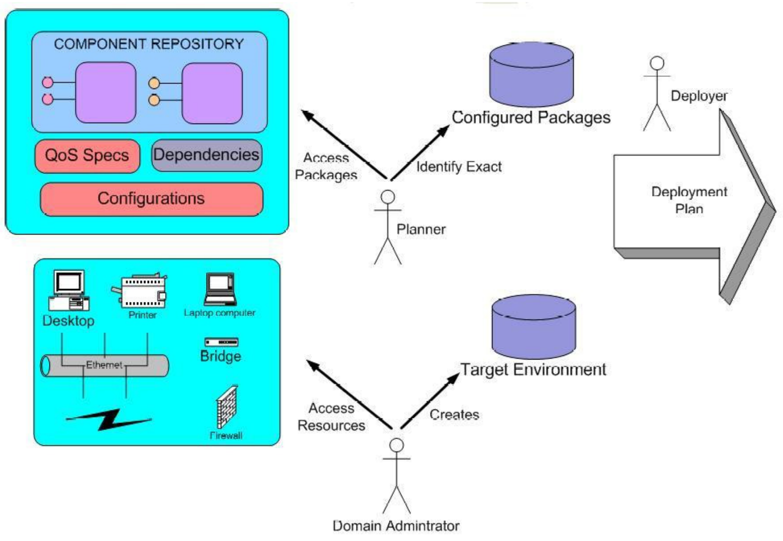


Figure 16. Planning Phase Activities

A key challenge of the planning phase is to ensure that the chosen configuration – together with the packages configured using OCML – deliver the appropriate QoS, e.g., end-to-end latency, minimal throughput and bounded jitter, required by the application. This challenge involves validating the deployment plan against the required QoS. To address the planning-related challenges described above, we have developed the *Benchmark Generation Modeling Language* (BGML) [Schmidt:04b].

BGML is a GME-based DSML that synthesizes benchmarking test suites to analyze the QoS performance of OCML-configured DRE systems. Figure 17 shows how BGML can be used in the planning phase to evaluate deployment plans (which map components to nodes) to provide feedback to developers as to whether a particular plan meets end-to-end QoS requirements or not. The following four steps shown in the figure characterize the BGML planning process:

1. In the first step, an experimenter uses PICML to represent the application scenario visually. This step also involves a visual representation of the deployment plan.
2. An experimenter uses BGML to associate QoS properties, such as latency, jitter, or throughput, with the application scenario.
3. BGML model interpreters then synthesize the appropriate test code to run the experiment and measure the resulting QoS.
4. Metrics are then fed back into models to verify whether the evaluated scenario meets the specified QoS.

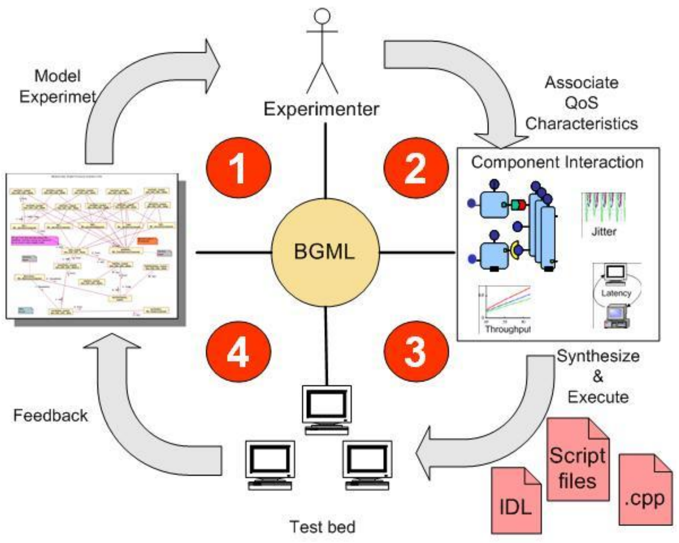
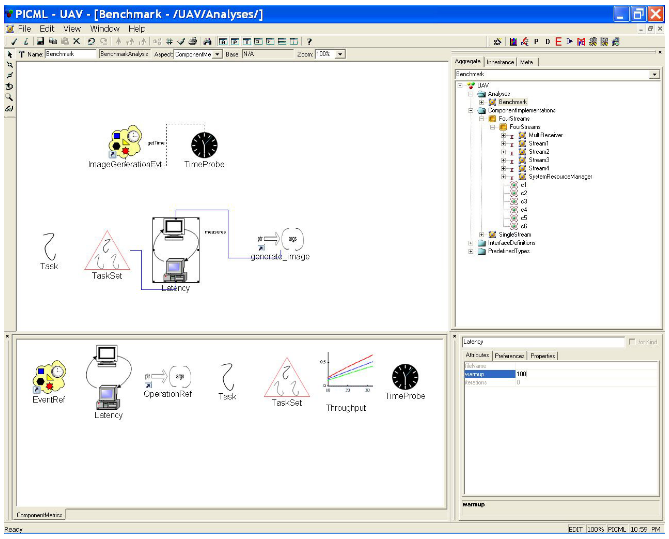


Figure 17. BGML Process View

BGML captures key QoS validation concerns of QoS-enabled middleware, such as modeling how distributed system components interact with each other and representing metrics that can be applied to specific configuration options and platforms. In particular, BGML provides test elements (such as operations, return-types, latency, throughput and timer elements) that can be used to represent a generic operation or a sequence or operation steps and associate non-functional QoS properties with them. BGML also provides workload elements, such as tasks and task-sets, that can be used to model and simulate background load present during the experimentation process. These workload elements are mapped to individual platform-specific code in the interpretation process. Finally, BGML synthesizes project build files (such as makefiles) needed to generate the executable code.

In our video distribution application, for example, an experimenter may want to conduct several performance studies. First, he/she may want to determine the best configuration for the individual `LocalResourceManagement` and the `Qosket` components that minimizes end-to-end latency. Figure 18 depicts how BGML can be used to associate the latency metric with the `generate_image()` operation provided by the `Qosket` components. For each stream, the `LocalResource-Management` component first uses the CropQosket to crop the image, then uses the `ScaleQosket` for image scaling based on the DiffServe priorities set at the `DiffServQosket`, and finally uses the `CompressQosket` to compress the image. The overall latency along the critical path thus equals the individual latencies at each component.



Latency

fileName	
warmup	100
iterations	0

warmup

Figure 18. *Applying BGML to the Video Distribution Application*

An experimenter may also want to conduct a different experiment, such as measuring the end-to-end latency between `LocalResourceManagement` component and `LocalReceiver` component. Figure 18 thus also shows a timer element associated with the `ImageGenerationEvt` event. For this experiment, the end-to-end latency between `LocalResourceManagement` component and `Local-Receiver` component is measured by observing the time interval between the `push_ImageGenerationEvt()` at the `LocalResourceManagement` component and the corresponding event at the `LocalReceiver` Component.

For the first latency experiment, BGML model interpreters synthesize the code by generating files, such as header and source benchmark files, header and source files to generate background load and a build file to create a benchmark library. The experimenter builds the benchmark as a library, specializes the benchmark with the type of the remote reference, operation signature and the parameter values. The CIAO run-time deployment infrastructure deploys the video distribution system along with the benchmark to generate results.

For the second end-to-end latency experiment, BGML interpreters can synthesize files, including a `ImageGeneration_Event.h`, header file that provides two operations, `start_time_probe()` and `stop_ time_probe()`, that measure the time at the local machines. If the components are at the same node, the different between the start and stop operations provide a good estimation of latency. The scenario also shows a restriction with the benchmark process, i.e., the benchmarks should run on a single host. When run on different hosts, an external entity needs to compute the difference between the time values to calculate latency. Moreover, the values may suffer from clock skew at individual nodes. These limitations are inherent to any benchmark process and are not specific to BGML.

1.4. RELATED WORK

This section summarizes related efforts associated with developing DRE systems using an MDD approach and compares these efforts with our work on CoSMIC.

Cadena [Hatcliff:03] is an integrated environment developed at Kansas State University (KSU) for building and modeling component-based DRE systems, with the goal of applying static analysis, model-checking, and lightweight formal methods to enhance these systems. Cadena also provides a component assembly framework for visualizing and developing components and their connections. Unlike PICML, however, Cadena does not support activities such as component packaging, generating deployment descriptors, component deployment planning, and hierarchical modeling of component assemblies. To develop a complete MDD environment that seamlessly integrates component development and model checking capabilities, we are collaborating [Schmidt:04g] with KSU to integrate PICML with Cadena's model checking tools, so we can accelerate the development and verification of DRE systems.

The Virginia Embedded Systems Toolkit (VEST) [VEST:03] and the Automatic Integration of Reusable Embedded Systems (ARES) [AIRES:03] are MDD analysis tools that evaluate whether certain timing, memory, power, and cost constraints of real-time and embedded applications are satisfied. Components are selected from pre-defined libraries, annotations for desired real-time properties are added, the resulting code is mapped to a hardware platform, and real-time and schedulability analysis is done. In contrast, PICML allows component modelers to model the complete functionality of components and intra-component interactions, and does not rely on predefined libraries. PICML also allows DRE system developers the flexibility in defining the target platform, and is not restricted to just processors.

The Embedded Systems Modeling Language (ESML) [Karsai:02] was developed by ISIS at Vanderbilt University to provide a visual metamodeling language based on GME that captures multiple views of embedded systems, allowing a diagrammatic specification of complex models. The modeling building blocks include software components, component interactions, hardware configurations, and scheduling policies. The user-created models can be fed to analysis tools (such as AIRES, VEST, and Cadena) to perform schedulability and event analysis. These analyses are used to perform design decisions (such as component allocations to the target execution platform). Unlike PICML, ESML is platform-specific since it is customized for the Boeing Boldstroke PRiSm QoS-enabled component model [Sharp:03, Roll:03]. ESML also does not support nested assemblies and the allocation of components are tied to processor boards, which is a proprietary feature of the Boldstroke component model. We are working with the ESML team at ISIS to integrate the ESML and PICML metamodels to produce a unified DSML suitable for modeling a broad range of QoS-enabled component models.

Corona [Greenfield:04] is a framework for developing MDD tools that can significantly increase the level of automation in application development by applying domain-specific visual languages to enable rapid assembly and configuration of framework-based components. The Corona MDD framework is similar to GME, i.e., it provides visual metamodeling tools, as well as interpreters that translate modeling elements to platform-specific code. We are collaborating with Microsoft to integrate our CoSMIC DSMLs for DRE systems to Corona.

1.5. CONCLUDING REMARKS

Although QoS-enabled component middleware represents an advance over previous generations of software infrastructure technologies, its additional complexities can also negate its key benefits when applied to complex distributed real-time and embedded (DRE) systems. A promising technology for resolving these complexities is Model-Driven Development (MDD) [Greenfield:04]. MDD tools provide correct-by-construction support for designing and validating DRE systems by integrating (1) analysis techniques that reason about DRE systems and (2) platform-independent generation capabilities that can target multiple component middleware technologies, such as CCM, J2EE, .NET, and ICE.

This chapter describes how the CoSMIC MDD toolsuite developed at Vanderbilt University help resolve key complexities of QoS-enabled component middleware. We applied several of CoSMIC's domain-specific modeling languages (DSMLs) to a video distribution application. Using this application as a representative example of common DRE systems, we showed how CoSMIC can support:

- **Design-time activities**, such as specification of the functionality of components, their interactions with other components, the assembly and packaging of components, and the configuration of the QoS-enabled component middleware on which the components run.
- **Deployment-time activities**, such as specification of target environment, and automatic deployment plan generation.
- **Quality Assurance (QA)-time activities**, such as validation of the configuration and deployment platform and their impact on QoS.

The CoSMIC MDD tools help bridge the gap between design-time verification and model-checking tools (such as Cadena [Hatcliff:03], VEST [VEST:03], and AIRES [AIRES:03]) and the actual deployed and validated component implementations [Schmidt:04g].

The lessons learned by applying our integrated CoSMIC MDD tools to the video distribution application case study described in Sections 1.2 and 1.3 illustrate that:

- Component and platform modeling improves DRE systems reasoning, and enables the comprehension of the system at a higher level of abstraction relative to conventional distributed object computing and component middleware approaches.
- Early detection of errors improves productivity significantly, which in turn helps increase the effectiveness of applying QoS-enabled component middleware technologies to the DRE systems domain.
- End-to-end tool-chains for DRE systems need to bridge analysis and empirical results.
- An MDD approach provides a lighter-weight technique for quickly evaluating QoS on different configurations and platforms. The generative capabilities of OCML and BGML ensure that most changes needed to conduct the evaluations are generated from higher-level models.
- MDD tools and process alleviate key complexities involved in understanding the impact of middleware configurations on application QoS and bring rigor to otherwise *ad hoc* processes used by developers to configure and deploy middleware for DRE systems.

- Current MDD approach of QoS evaluation requires human effort, for example to change the node component association in the models and re-running the interpreter to generate the XML metadata. Process automation is necessary to run the benchmarks independently without any intervention.

Our future work will focus on extending CoSMIC with support for dynamic component allocation, automated performance analysis of component systems by empirically evaluating component interactions with respect to various performance metrics, and flexible performance modeling of components to satisfy real-time QoS properties. We are developing MDD solutions for these problems and integrating the resulting tools as part of the broader CoSMIC end-to-end modeling tool-suite.

REFERENCES

[AIRES:03] Sharath Kodase, Shige Wang, Zonghua Gu, and Kang G. Shin, "Improving Scalability of Task Allocation and Scheduling in Large Distributed Real-time Systems using Shared Buffers," Proceedings of the 9th IEEE Real-time/Embedded Technology and Applications Symposium (RTAS), Washington, DC, May, 2003.

[CORBA:02a] Object Management Group, The Common Object Request Broker: Architecture and Specification, Object Management Group, May, 2002, 2.6.1.

[CORBA:02b] Object Management Group, "The Common Object Request Broker: Architecture and Specification," Object Management Group, Dec, 2002, 3.0.2.

[CorbaComponents:02] "CORBA Components," Object Management Group, OMG Document formal/2002-06-65, Jun, 2002.

[DandC:03] "Deployment and Configuration Adopted Submission," Object Management Group, OMG Document ptc/03-07-08, July, 2003.

[GME:01] Akos Ledeczi, Arpad Bakay, Miklos Maroti, Peter Volgysei, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai, "Composing Domain-Specific Design Environments," IEEE Computer, November, 2001.

[Greenfield:04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent, Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, John Wiley & Sons, New York, 2004.

[Hatcliff:03] John Hatcliff, William Deng, Matthew Dwyer, Georg Jung, and Venkatesh Prasad, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems," Proceedings of the 25th International Conference on Software Engineering, Portland, OR, May, 2003.

[Heineman:01] George T. Heineman and Bill T. Councill, Component-Based Software Engineering: Putting the Pieces Together, Addison-Wesley, Reading, MA, 2001.

[Karsai:02] Gabor Karsai, Sandeep Neema, Ben Abbott, and David Sharp, “A Modeling Language and Its Supporting Tools for Avionics Systems,” Proceedings of 21st Digital Avionics Systems Conference, August, 2002.

[OCL:03] “Unified Modeling Language: OCL version 2.0 Final Adopted Specification,” Object Management Group, OMG Document ptc/03-10-14, October, 2003.

[Roll:03] Wendy Roll, “Towards Model-Based and CCM-Based Applications for Real-Time Systems,” Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC), IEEE/IFIP, Hakodate, Hokkaido, Japan, May, 2003.

[RTCorba:02] “Real-time CORBA Specification,” Object Management Group, OMG Document formal/02-08-02, August, 2002.

[RTSJ:00] Greg Bollella, James Gosling, Ben Brosgol, Peter Dibble, Steve Furr, David Hardin, and Mark Turnbull, The Real-Time Specification for Java, Addison-Wesley, 2000.

[SCA:01] “Software Communications Architecture (SCA) Specification,” Modular Software-programmable Radio Consortium, Joint Tactical Radio Systems Joint Program Office, MSRC-5000 SCA Version2.2, Nov, 2001.

[Schmidt:97] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, “The Design and Performance of Real-Time Object Request Brokers,” Computer Communications, Elsevier, Volume 21, Number 4, April, 1998.

[Schmidt:03a] R. Schantz, J. Loyall, D. Schmidt, C. Rodrigues, Y. Krishnamurthy, and I. Pyarali, “Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware,” Proceedings of Middleware 2003, 4th International Conference on Distributed Systems Platforms, IFIP/ACM/USENIX, Rio de Janeiro, Brazil, June 2003.

[Schmidt:03b] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Craig Rodrigues, Balachandran Natarajan, Joseph P. Loyall, Richard E. Schantz, and Christopher D. Gill, “QoS-enabled Middleware,” Middleware for Communications, 2003, Qusay Mahmoud, Wiley and Sons, New York.

[Schmidt:04a] Aniruddha Gokhale, Krishnakumar Balasubramanian, Arvind S. Krishna, Jaiganesh Balasubramanian, George T. Edwards, Gan Deng, Emre Turkay, Jeff Parsons, Douglas C. Schmidt, “Model-driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications,” The Journal of Science of Computer Programming: Special Issue on Model-driven Architecture, Elsevier, Mehmet Aksit, 2005 (to appear).

[Schmidt:04b] Arvind S. Krishna, Douglas C. Schmidt, Adam Porter, Atif Memon, and Diego Sevilla-Ruiz, “Improving the Quality of Performance-intensive Software via Model-integrated Distributed Continuous Quality Assurance,” Proceedings of the 8th International Conference on Software Reuse, ACM/IEEE, Madrid, Spain, July, 2004.

[Schmidt:04c] Arvind S. Krishna, Cemal Yilmaz, Atif Memon, Adam Porter, Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan, "Preserving Distributed Systems Critical Properties: a Model-Driven Approach," IEEE Software special issue on Persistent Software Attributes, November/December. 2004.

[Schmidt:04d] Nanbor Wang, Chris Gill, Douglas C. Schmidt, and Venkita Subramonian, "Configuring Real-time Aspects in Component Middleware," Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04), Agia Napa, Cyprus, October, 2004.

[Schmidt:04e] Arvind S. Krishna, Emre Turkay, Aniruddha Gokhale, and Douglas C. Schmidt, "Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems," Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium, San Francisco, CA, March, 2005.

[Schmidt:04f] Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt, "A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems," Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium, San Francisco, CA, Mar, 2005.

[Schmidt:04g] Gabriele Trombetti, Aniruddha Gokhale, Douglas C. Schmidt, John Hatcliff, Gurdip Singh, and Jesse Greenwald, "An Integrated Model-driven Development Environment for Composing and Validating Distributed Real-time and Embedded Systems," Model-driven Software Development, Volume II of Research and Practice in Software Engineering, Sami Beydeda, Matthias Book, and Volker Gruhn, Springer-Verlag, New York, 2005.

[Sharp:03] David C. Sharp and Wendy C. Roll, "Model-Based Integration of Reusable Component-Based Avionics System," Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003, Washington DC, May, 2003.

[Szyperski:02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer, Component Software – Beyond Object-Oriented Programming – Second Edition, Addison-Wesley, Reading, Massachusetts, 2002.

[VEST:03] John A. Stankovic, Ruiqing Zhu, Ramasubramaniam Poornalingam, Chenyang Lu, Zhendong Yu, Marty Humphrey, and Brian Ellis, "VEST: An Aspect-based Composition Tool for Real-time Systems," Proceedings of the IEEE Real-time Applications Symposium, Washington, DC May, 2003.