MDE4DRE: Model Driven Engineering for Distributed Real-time & Embedded Systems

Dr. Aniruddha S. Gokhale
a.gokhale@vanderbilt.edu
Assistant Professor, EECS
Vanderbilt University
www.dre.vanderbilt.edu

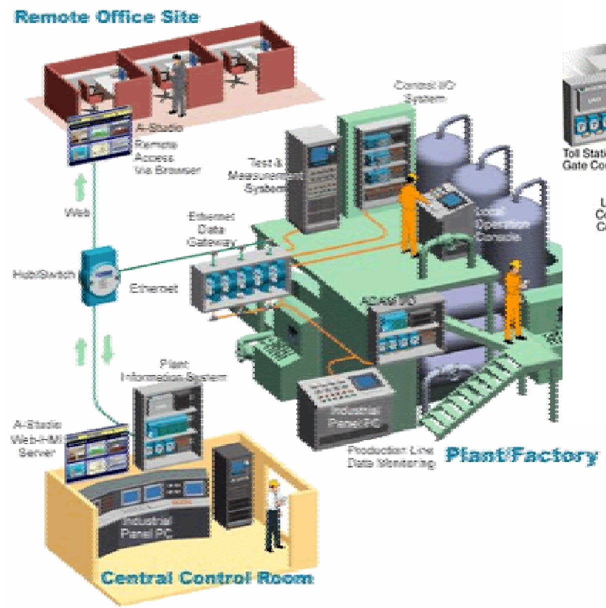
OMG RTWS 2007 Tutorial
Arlington, VA
July 9, 2007

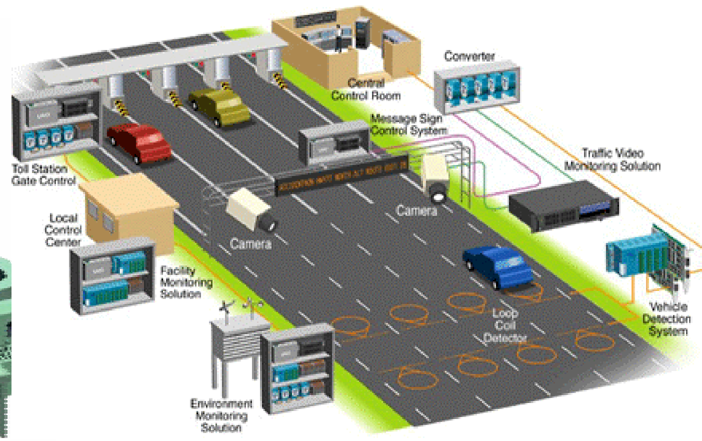
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee

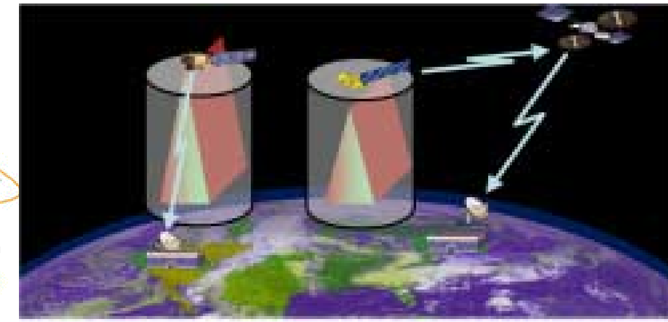
Tutorial Outline

1. Introduction
2. The Basics
3. Case Studies
4. Assembly & Packaging
5. QoS Provisioning
6. Deployment planning
7. Middleware Configuration
8. Continuous QoS Validation
9. Middleware Customization & Analysis
10. Future directions/Work-in-progress
11. Concluding Remarks

Distributed Real-time & Embedded (DRE) Systems







- Network-centric and large-scale “systems of systems”
  - e.g., industrial automation, emergency response
- Different communication semantics
  - e.g., pub-sub

Satisfying tradeoffs between multiple (often conflicting) QoS demands

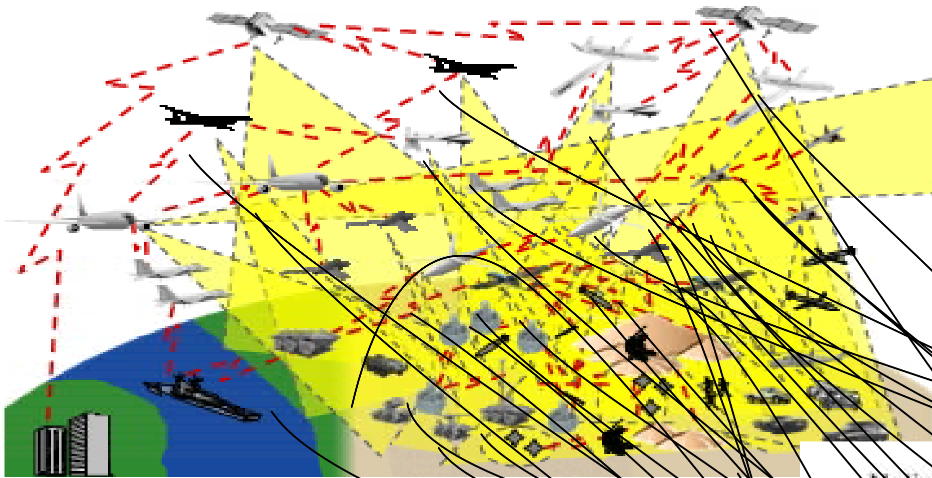
  - e.g., secure, real-time, reliable, etc.

Regulating & adapting to (dis)continuous changes in runtime environments

  - e.g., online prognostics, dependable upgrades, keep mission critical tasks operational, dynamic resource mgmt

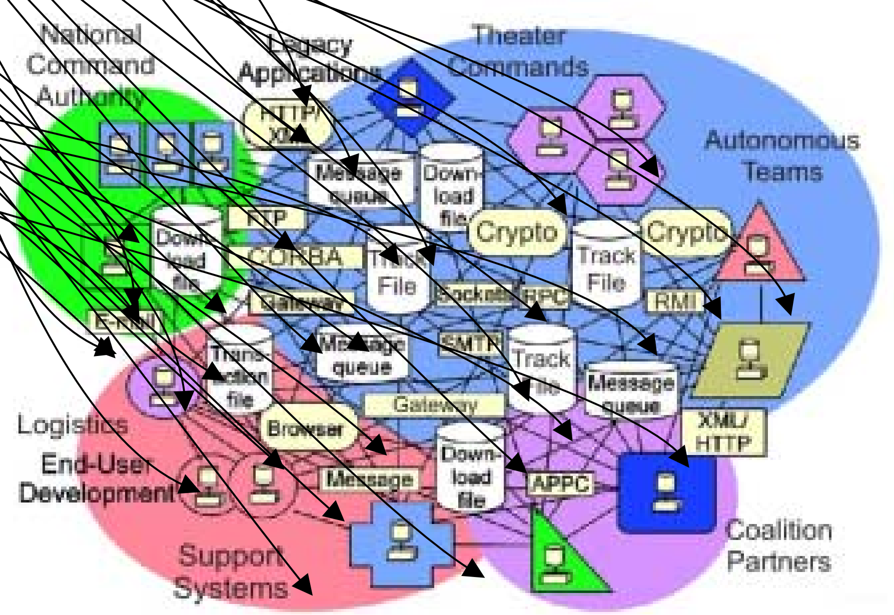
DRE systems increasingly adopting service oriented architectures

Challenges in Realizing DRE Systems



Variability in the ***problem space*** (domain expert role)

- Functional diversity
- Composition, deployment and configuration diversity
- QoS requirements diversity

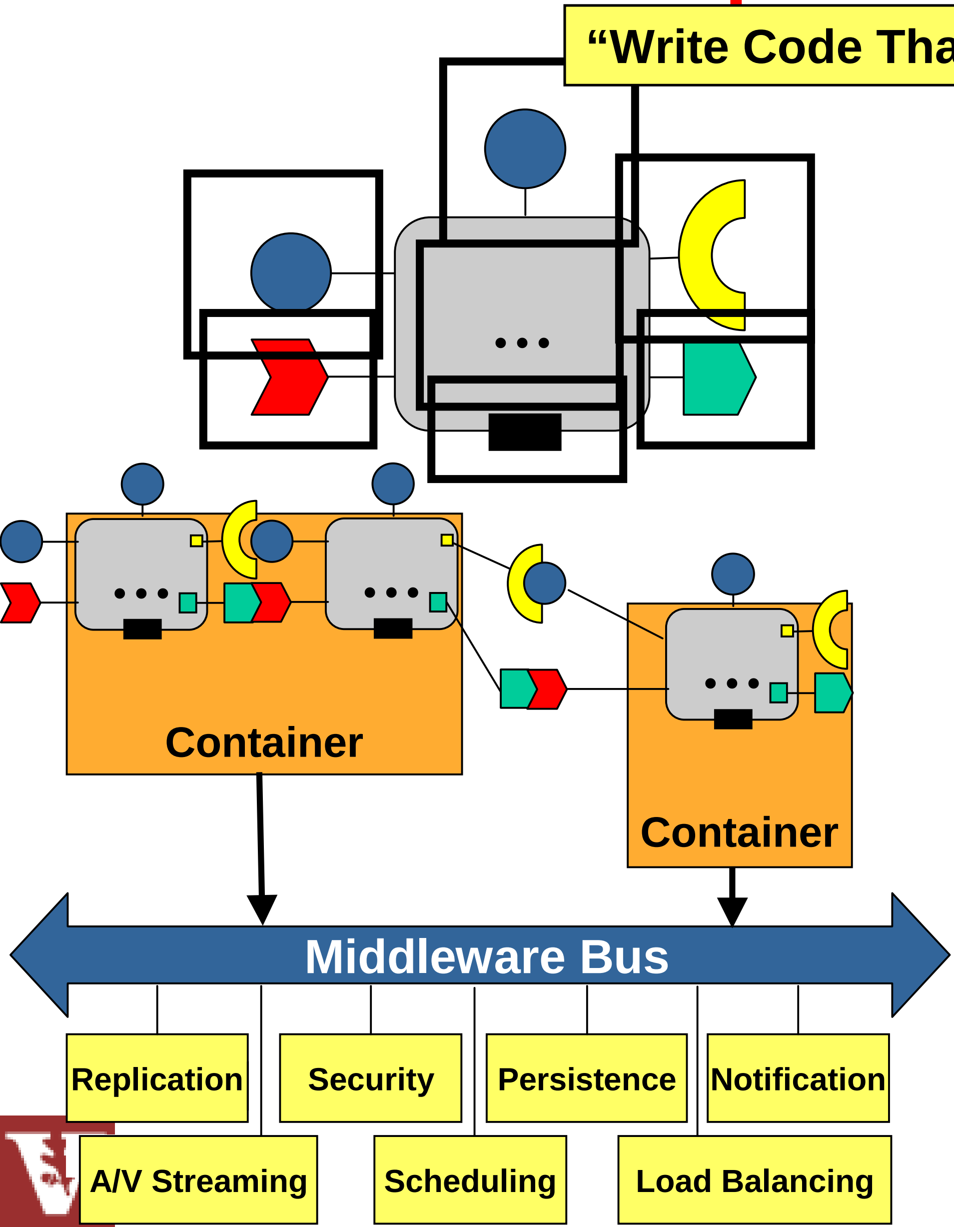


Variability in the ***solution space*** (systems integrator role)

- Diversity in platforms, languages, protocols & tool environments
- Enormous accidental & inherent complexities

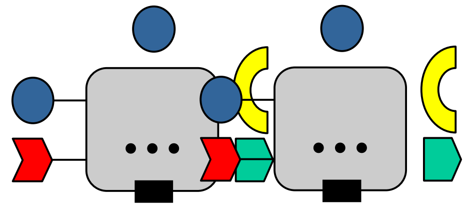
Mapping *problem* artifacts to *solution* artifacts is hard

Technology Enablers for DRE systems: Component Middleware

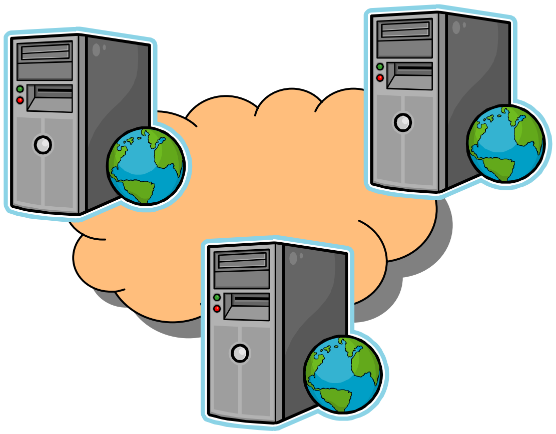


- *Components* encapsulate application "business" logic
- Components interact via *ports*
  - *Provided interfaces*, e.g.,facets
  - *Required connection points*, e.g., receptacles
  - *Event sinks & sources*
  - *Attributes*
- *Containers* provide execution environment for components with common operating requirements
- Components/containers can also
  - Communicate via a *middleware bus* and
  - Reuse *common middleware services*

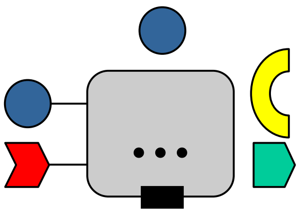
Lifecycle Issues in DRE Systems



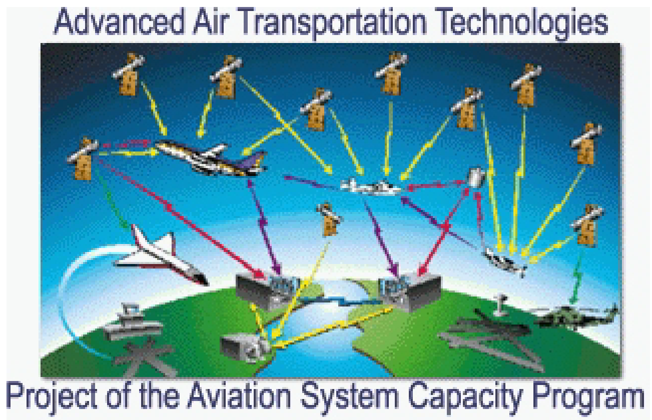
composition & packaging



deployment planning & QoS provisioning

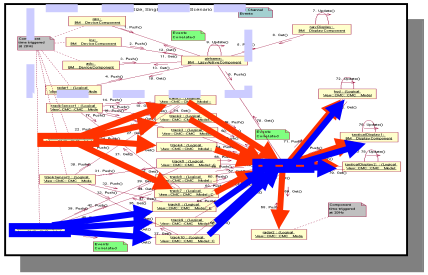


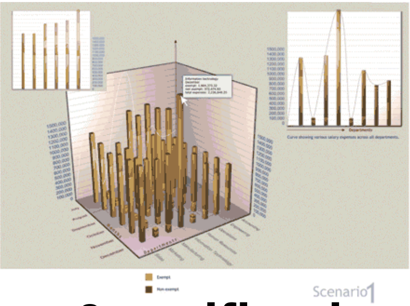
specification



Domain-specific Services
Common Services
Distribution Middleware
Infrastructure Middleware
Operating System & Network Stack

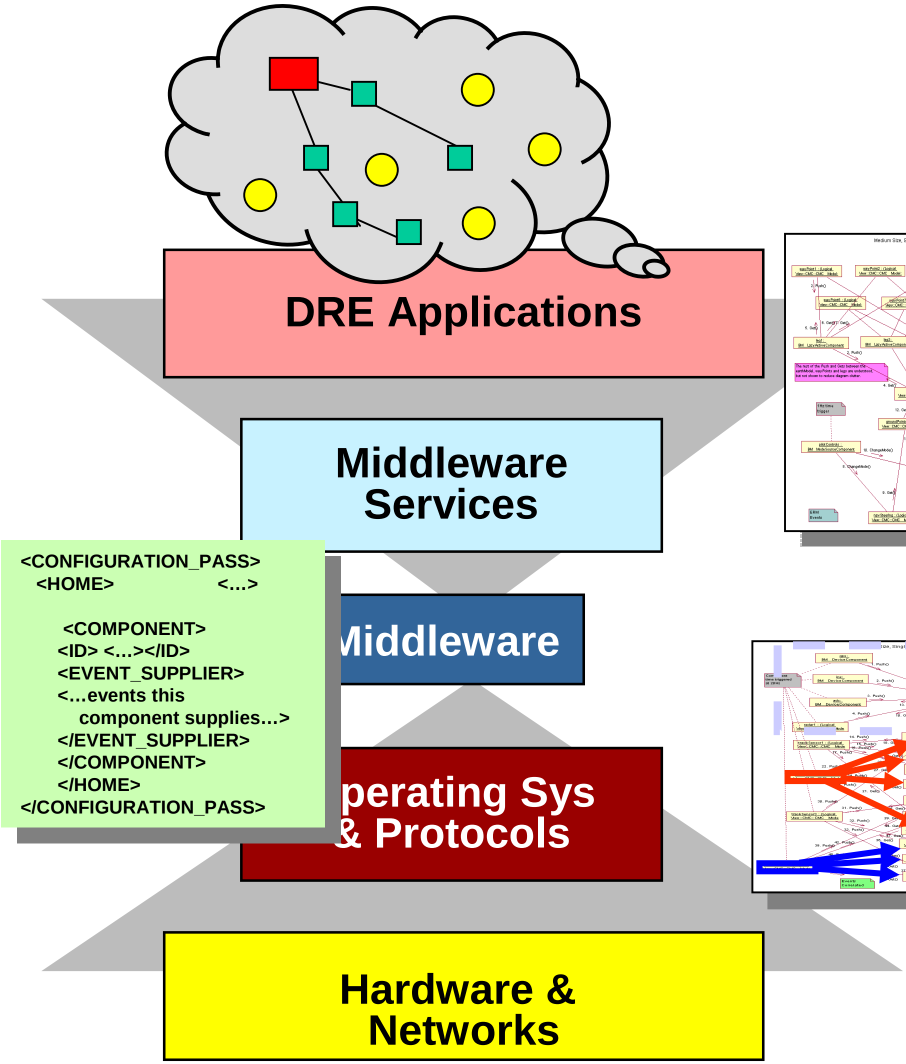
configuration & optimization

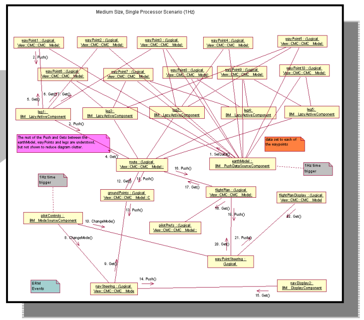


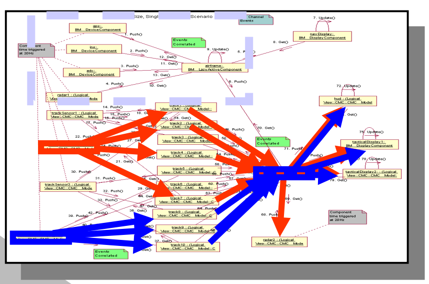


analysis, validation & verification

Tutorial Focus: *Model Driven Engineering (MDE)*







- Develop, validate, & standardize generative software technologies that:
  1. ***Model***
  2. ***Analyze***
  3. ***Synthesize &***
  4. ***Provision***

  multiple layers of middleware & ***application components*** that require ***simultaneous control*** of ***multiple quality of service properties end-to-end***
- Specialization is essential for inter-/intra-layer optimization & advanced product-line architectures

Goal is ***not*** to replace programmers per se – it is to provide ***higher-level domain-specific languages*** for middleware/application developers & users

OMG D&C Specification

Specification & Implementation

- Defining, partitioning, & implementing appln functionality as standalone components

Assembly & Packaging

- Bundling a suite of software binary modules & metadata representing app components

Installation

- Populating a repository with packages required by app

Configuration

- Configuring packages with appropriate parameters to satisfy functional & systemic requirements of an application without constraining to physical resources

Planning

- Making deployment decisions to identify nodes in target environment where packages will be deployed

Preparation

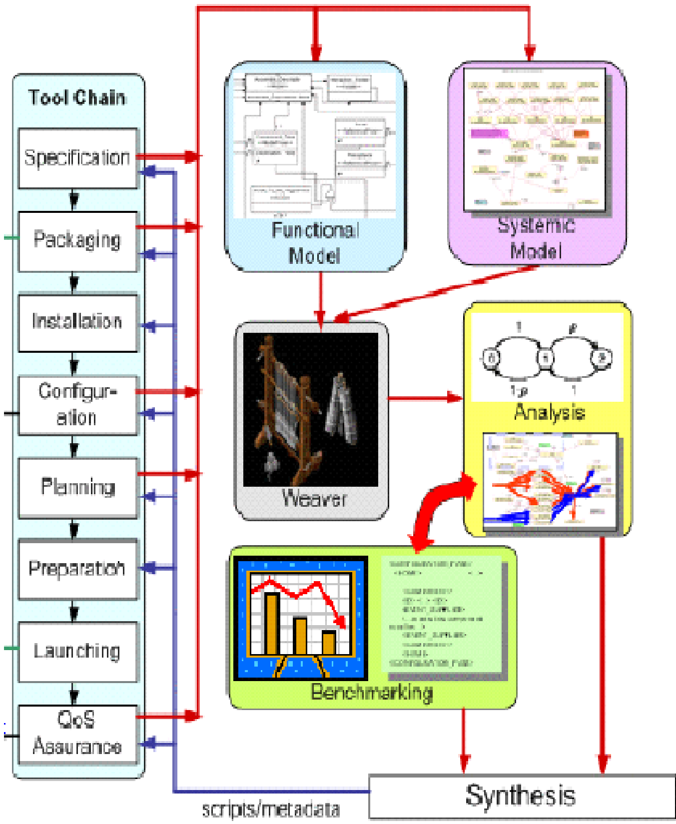
- Moving binaries to identified entities of target environment

Launching

- Triggering installed binaries & bringing appln to ready state

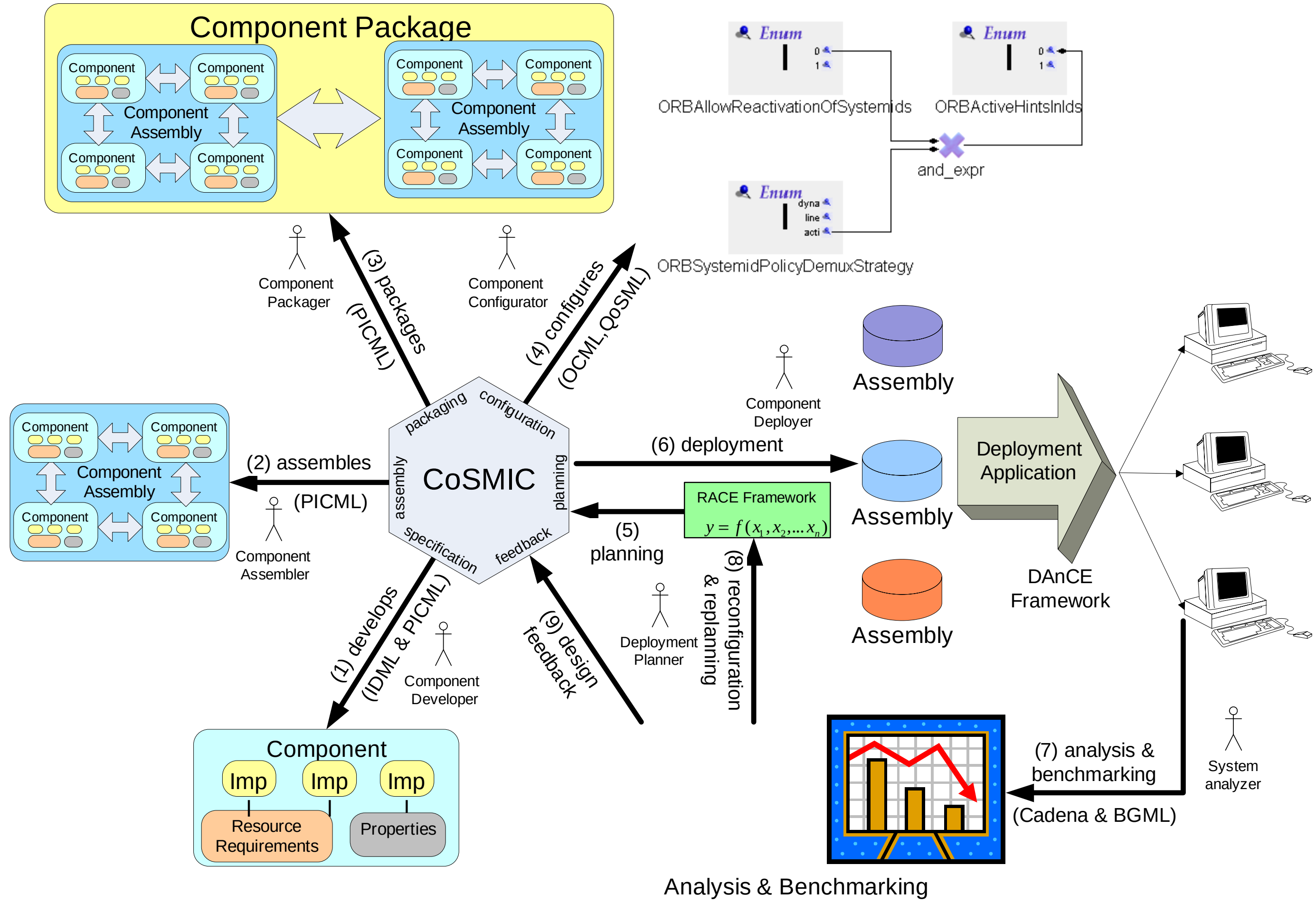
QoS Assurance & Adaptation

- QoS validation, runtime (re)configuration & resource management to maintain end-to-end QoS



OMG Deployment & Configuration (D&C) specification (ptc/05-01-07)

Our MDE Solution: CoSMIC



- CoSMIC tools e.g., PICML used to model application components, CQML for QoS
- Captures the data model of the OMG D&C specification
- Synthesis of static deployment plans for DRE applications
- Capabilities being added for QoS provisioning (real-time, fault tolerance)

CoSMIC can be downloaded at www.dre.vanderbilt.edu/cosmic

Part 1
The Basics

Tools & Technologies

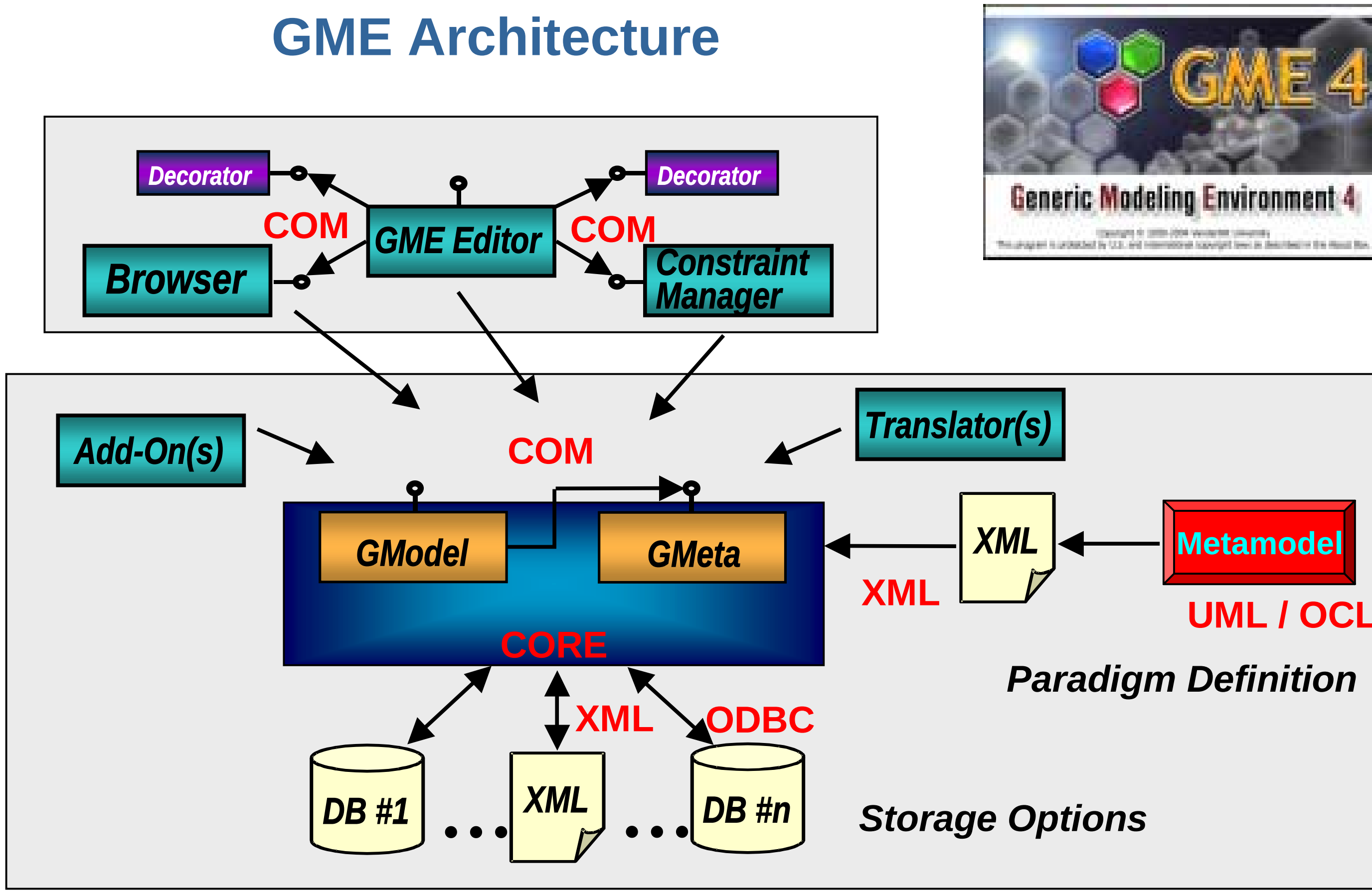
Technology Enabler: Generic Modeling Environment (GME)

"Write Code That Writes Code That Writes Code!"

GME Architecture

Application Developers (Modelers)

MDE Tool Developer (Metamodeler)

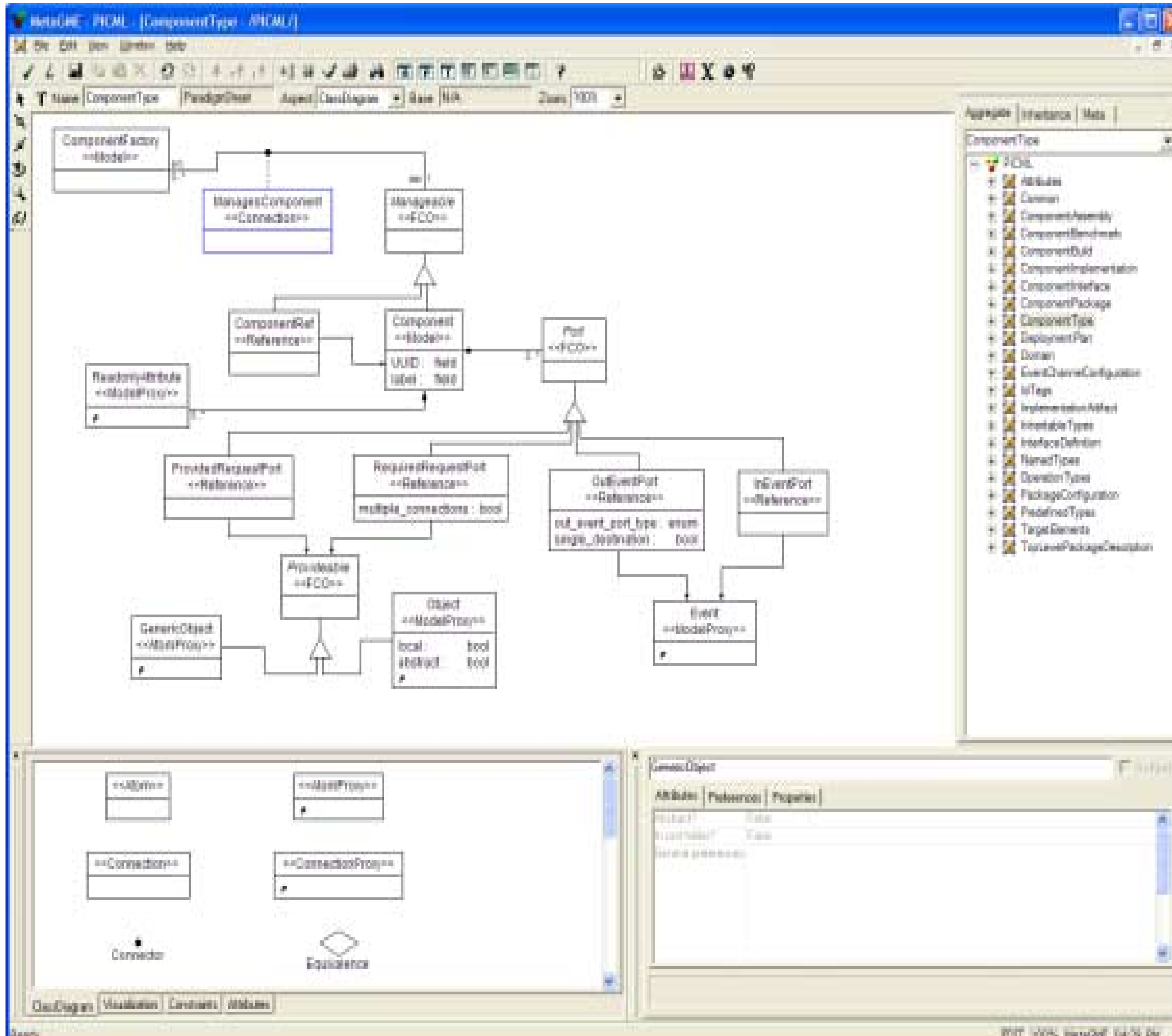


Goal: Correct-by-construction DRE systems

www.isis.vanderbilt.edu/Projects/gme/default.htm

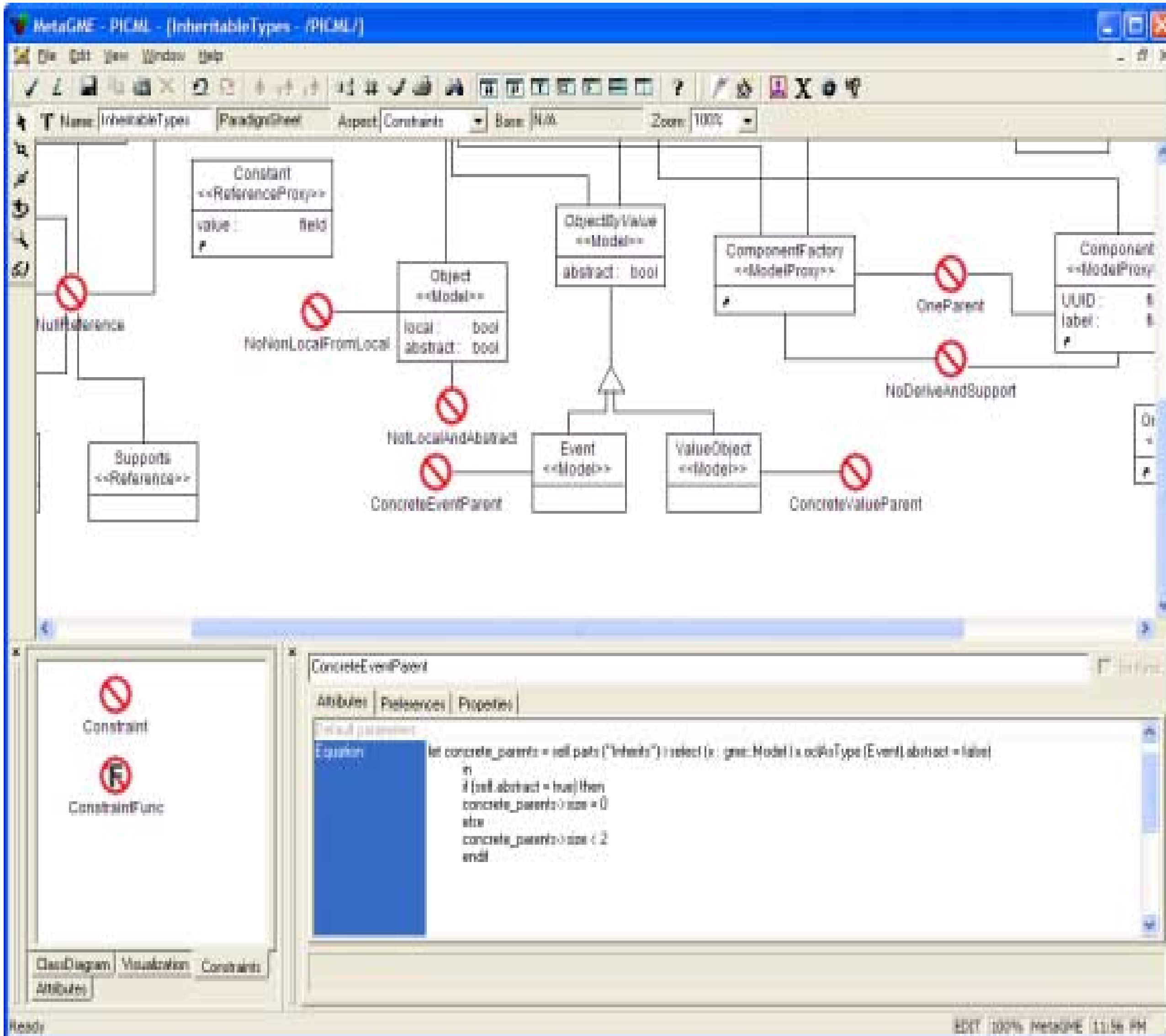
MDE Tool Development in GME

- **Tool developers** use MetaGME to develop a *domain-specific graphical modeling environment*
  - Define syntax & visualization of the environment via *metamodeling*



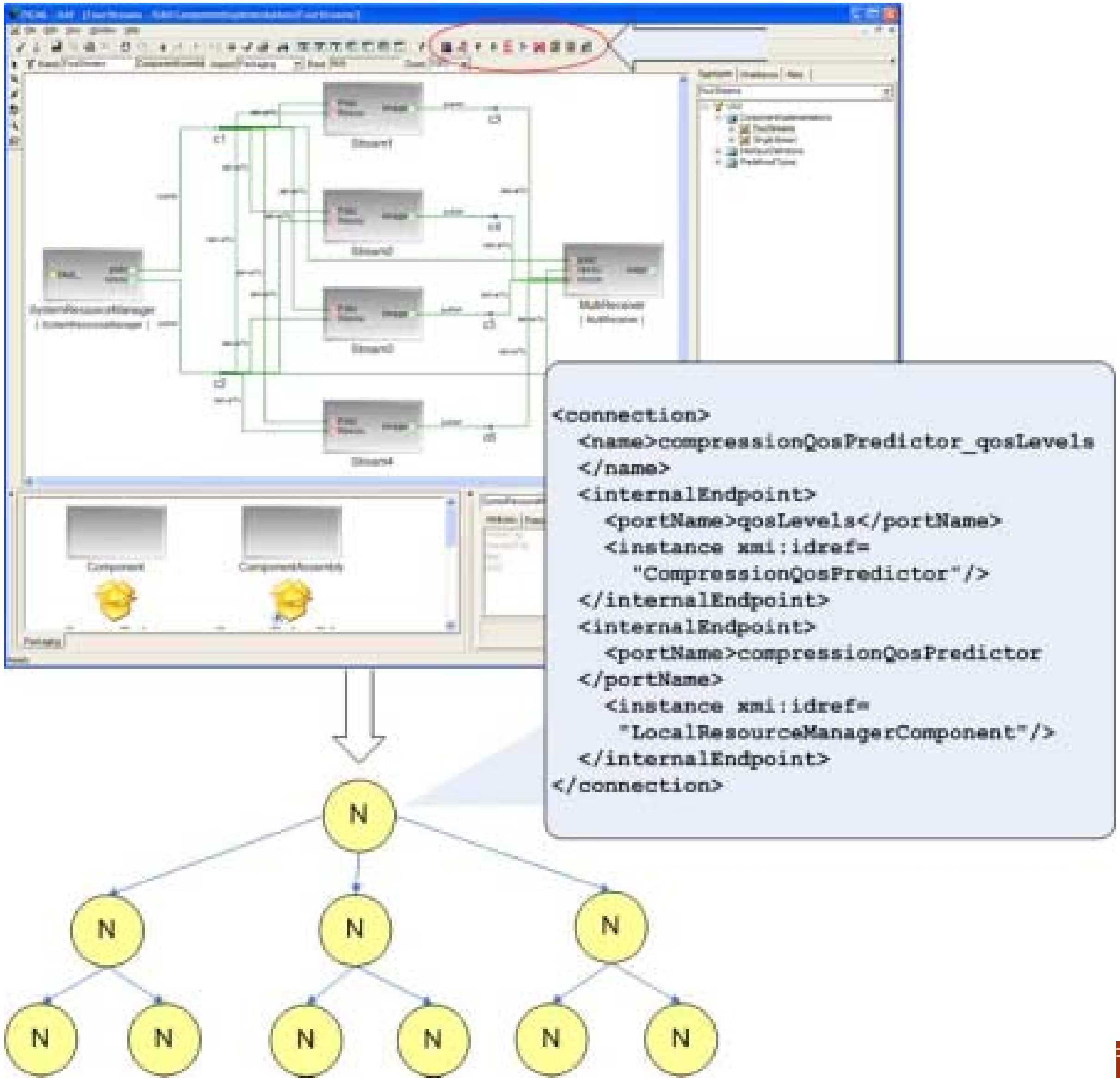
MDE Tool Development in GME

- **Tool developers** use MetaGME to develop a *domain-specific graphical modeling environment*
  - Define syntax & visualization of the environment via *metamodeling*
  - Define static semantics via *Object Constraint Language (OCL)*



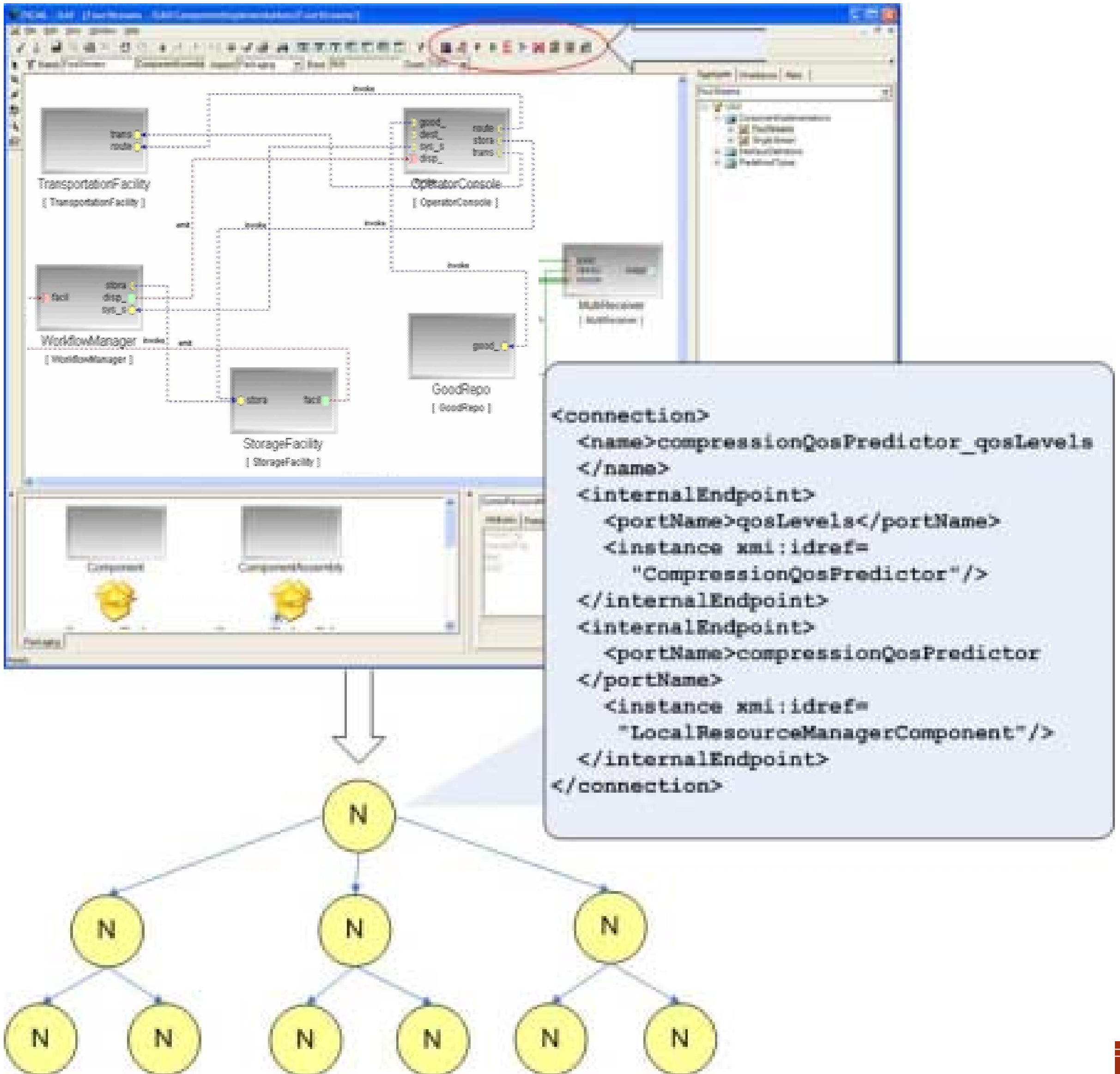
MDE Tool Development in GME

- **Tool developers** use MetaGME to develop a *domain-specific graphical modeling environment*
  - Define syntax & visualization of the environment via *metamodeling*
  - Define static semantics via *Object Constraint Language (OCL)*
  - Dynamic semantics implemented via *model interpreters*



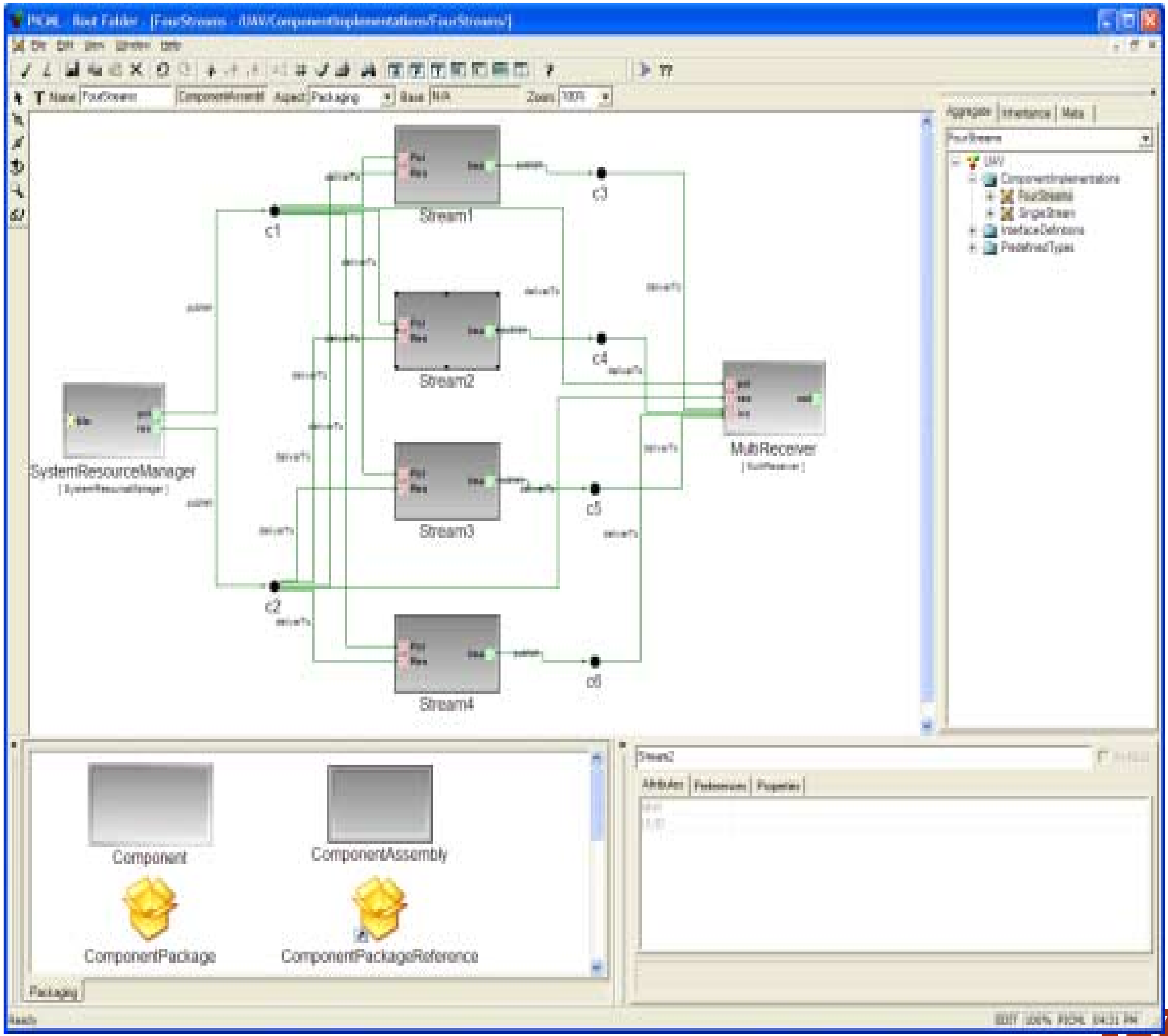
MDE Tool Development in GME

- **Tool developers** use MetaGME to develop a *domain-specific graphical modeling environment*
  - Define syntax & visualization of the environment via *metamodeling*
  - Define static semantics via *Object Constraint Language (OCL)*
  - Dynamic semantics implemented via *model interpreters*



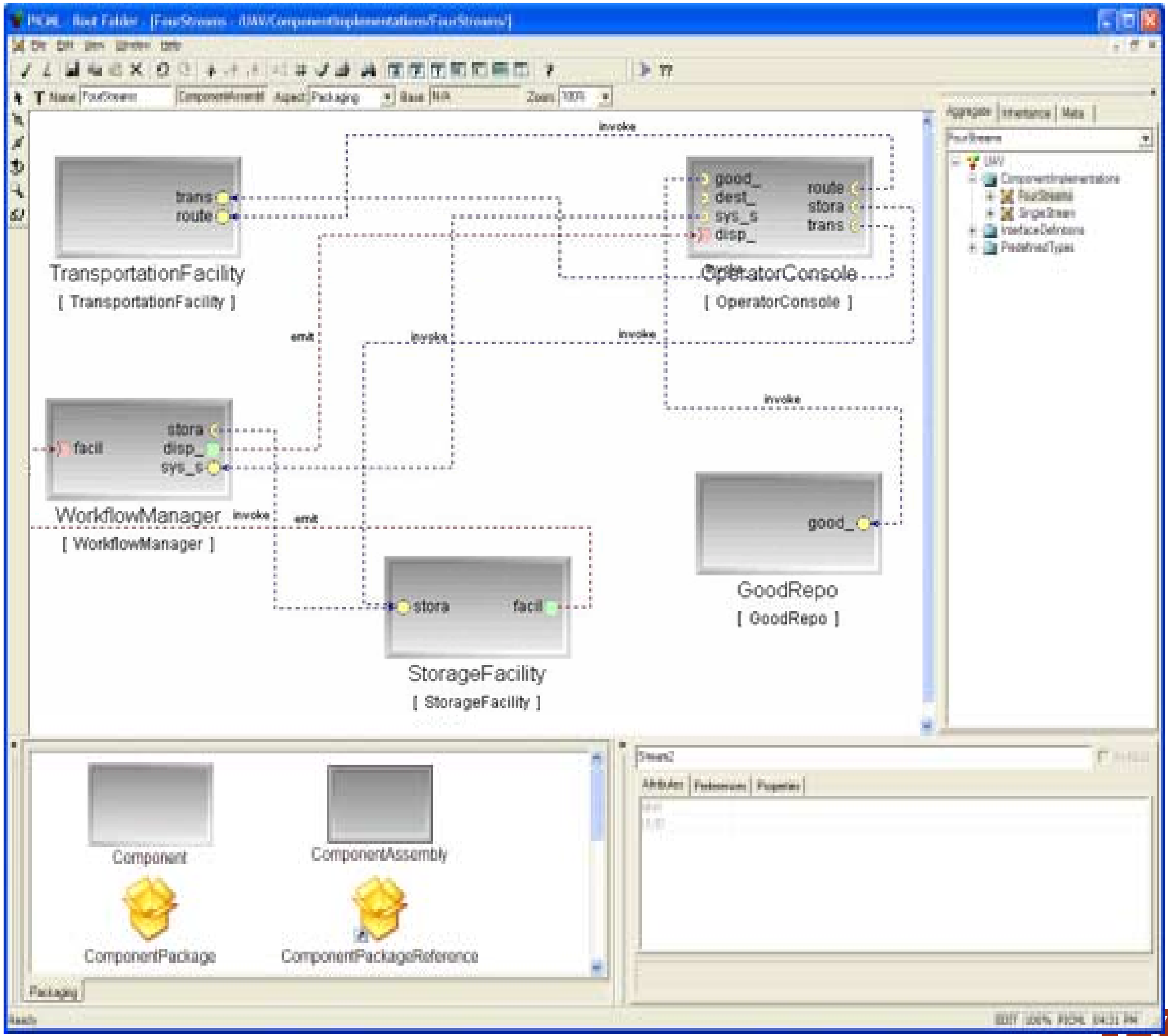
MDE Application Development with GME

- **Application developers** use modeling environments created w/MetaGME to build *applications*
  - Capture elements & dependencies visually



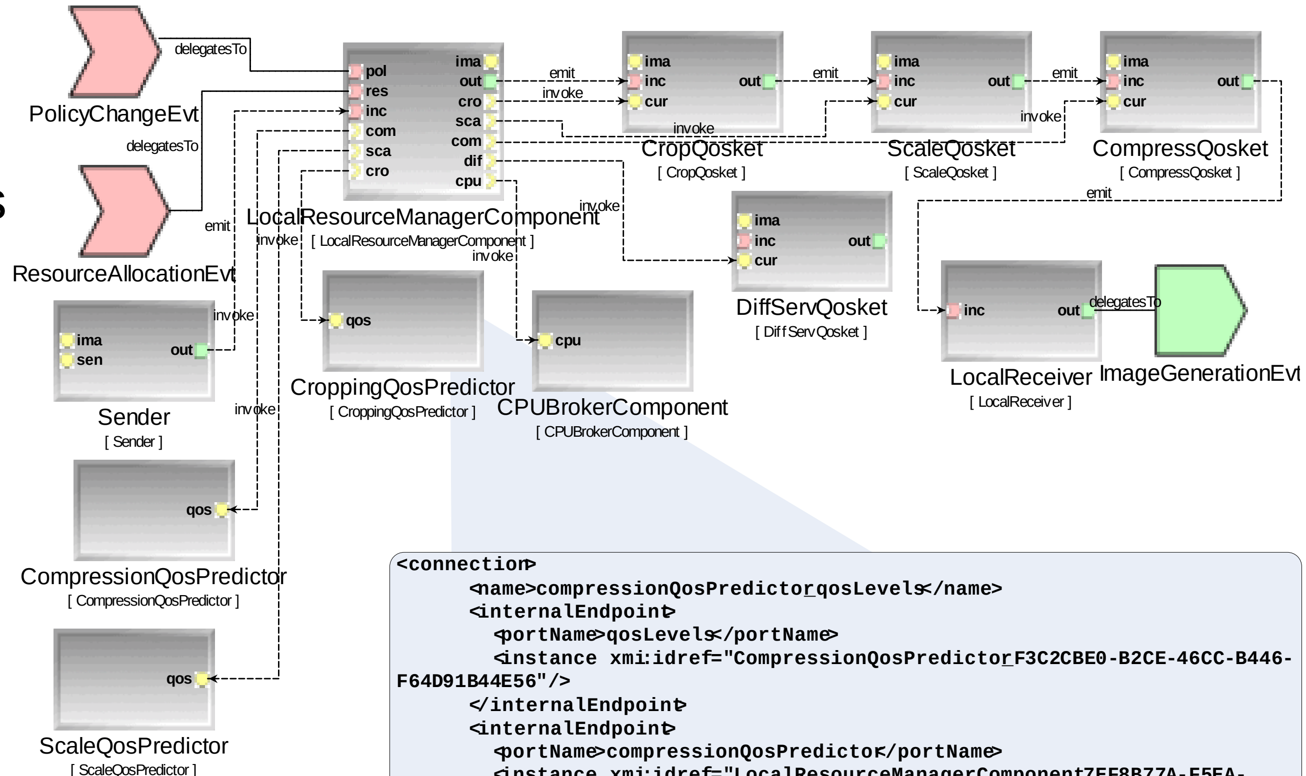
MDE Application Development with GME

- **Application developers** use modeling environments created w/MetaGME to build *applications*
  - Capture elements & dependencies visually



MDE Application Development with GME

- **Application developers** use modeling environments created w/MetaGME to build *applications*
  - Capture elements & dependencies visually
  - Model interpreter produces something useful from the models
    - e.g., code, simulations, deployment descriptions & configurations

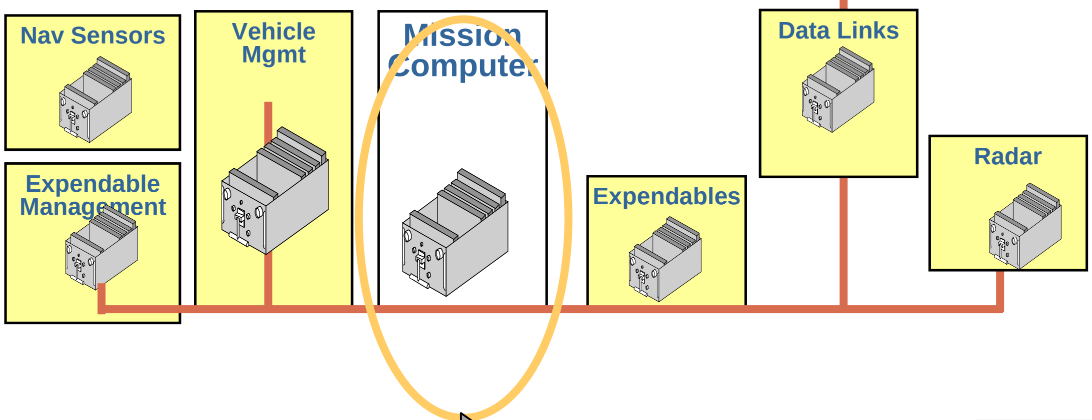


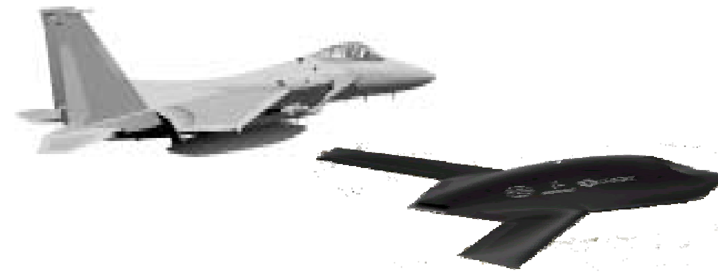

```
<connection>
      <name>compressionQosPredictor_qosLevels</name>
      <internalEndpoint>
        <portName>qosLevels</portName>
        <instance xmi:idref="CompressionQosPredictor_F3C2CBE0-B2CE-46CC-B446-F64D91B44E56"/>
      </internalEndpoint>
      <internalEndpoint>
        <portName>compressionQosPredictor</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>
```

Part 2
Case Studies

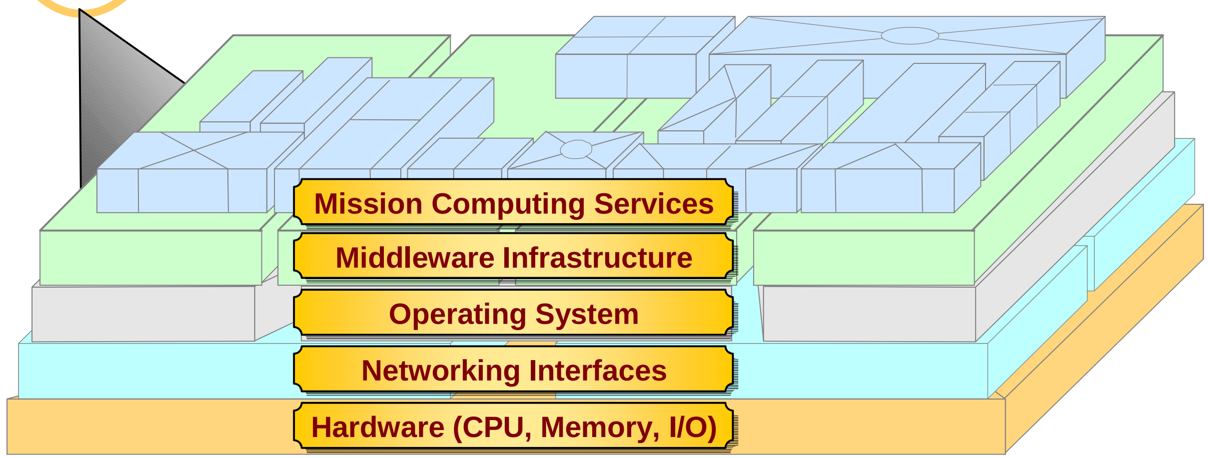
Bold Stroke
Robot Assembly
NASA Space Mission
Shipboard Computing
Modern Office
Stock Application

DRE System Example 1: Boeing Bold Stroke

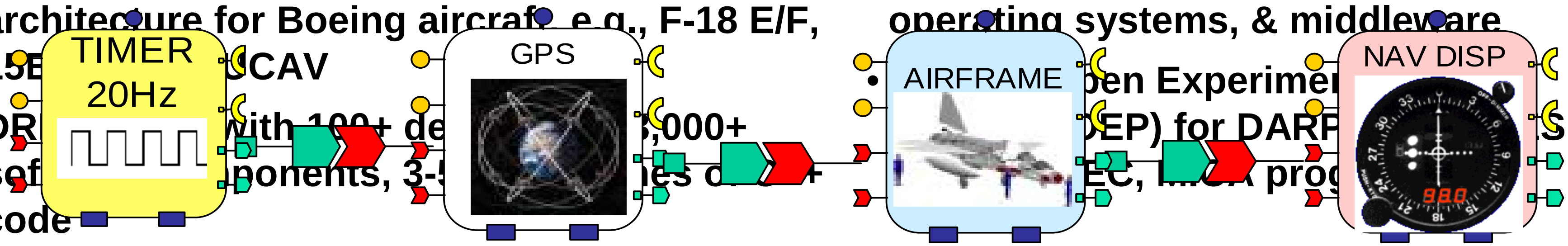




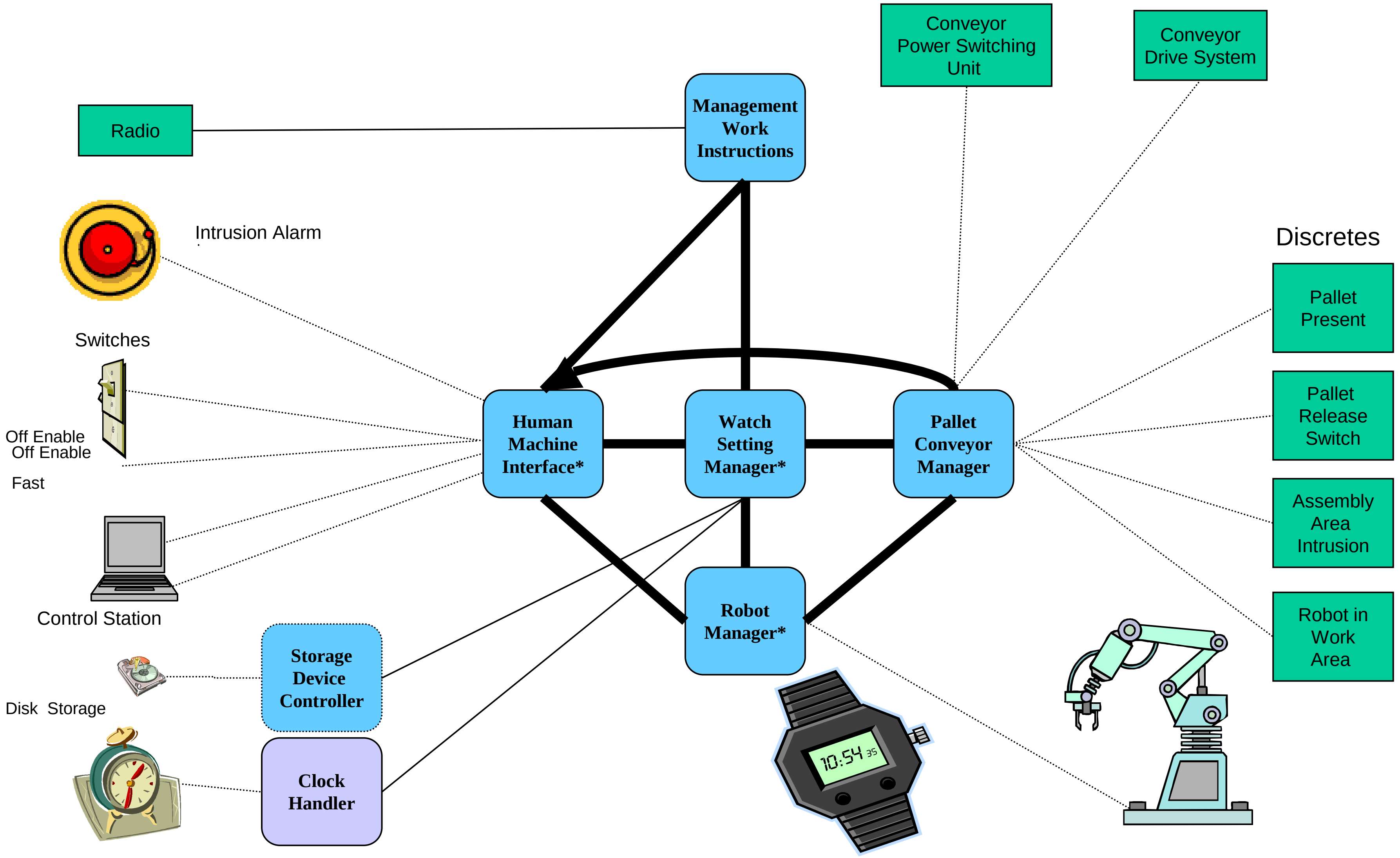
Bold Stroke Architecture



- Avionics mission computing product-line architecture for Boeing aircraft, e.g., F-18 E/F, 15E ... UCAV
- DR... with 100+ de... 3,000+ software components, 3-5 ... code
- Based on COTS hardware, networks, operating systems, & middleware
- ...en Experiment... EP) for DARPA ... S, ...EC, ... program

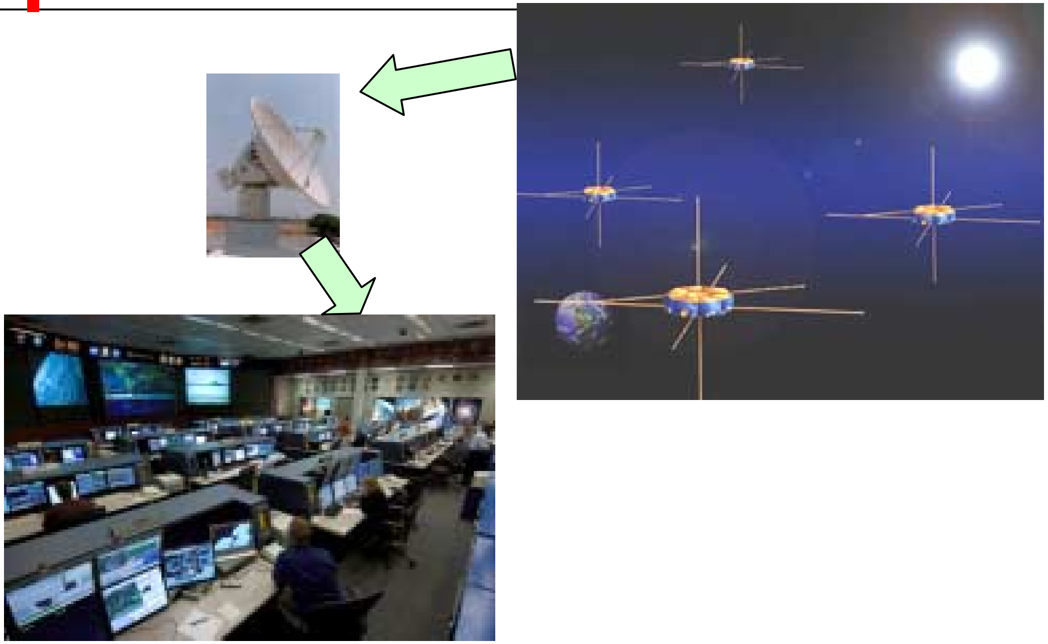


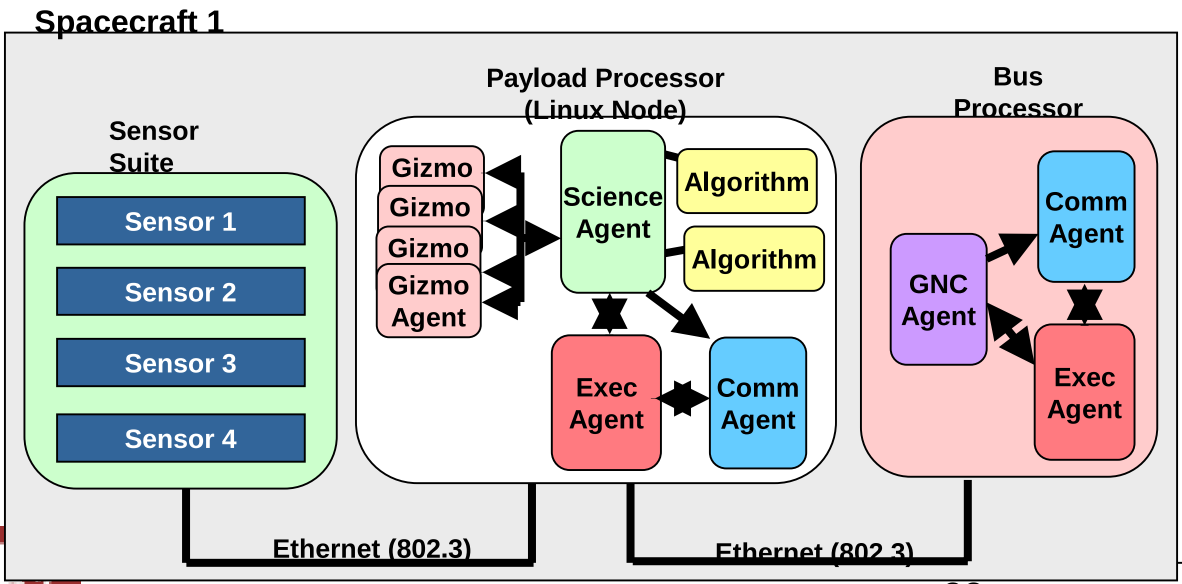
DRE System Example 2: RobotAssembly



DRE System Example 3: NASA MMS Mission

- NASA’s Magnetospheric MultiScale (MMS) space mission consists of four identically instrumented spacecraft & a ground control system
  - Collect mission data
  - Send it to ground control at appropriate time instances



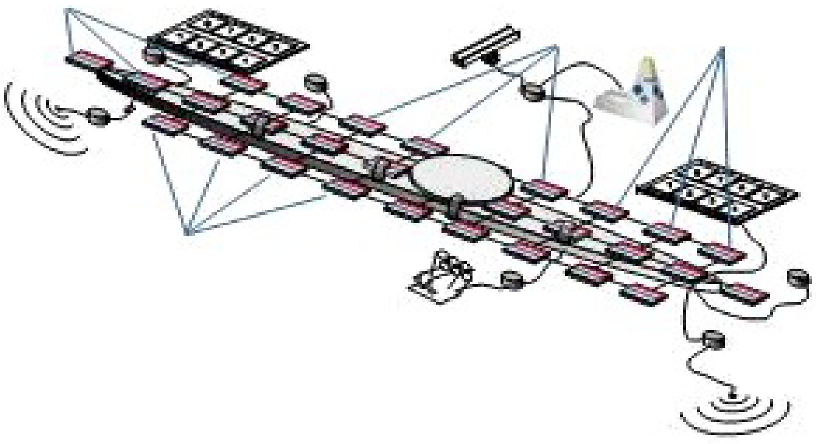


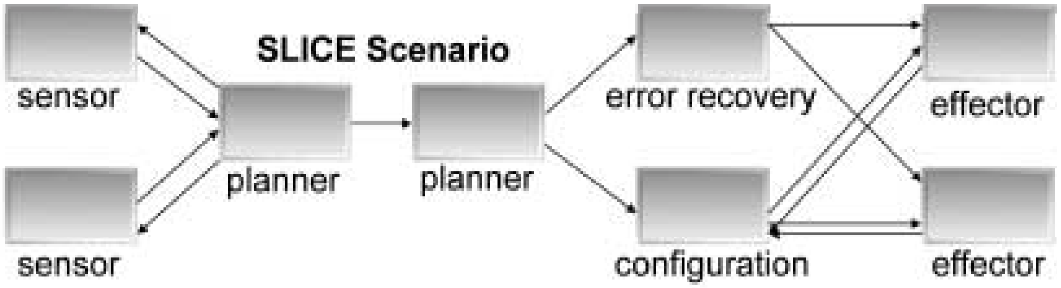
- MMS application components are bundled together into *hierarchical assemblies*
- Assembly package metadata conveys component interconnections & implementation alternatives

22

DRE System Example 4: Shipboard Computing

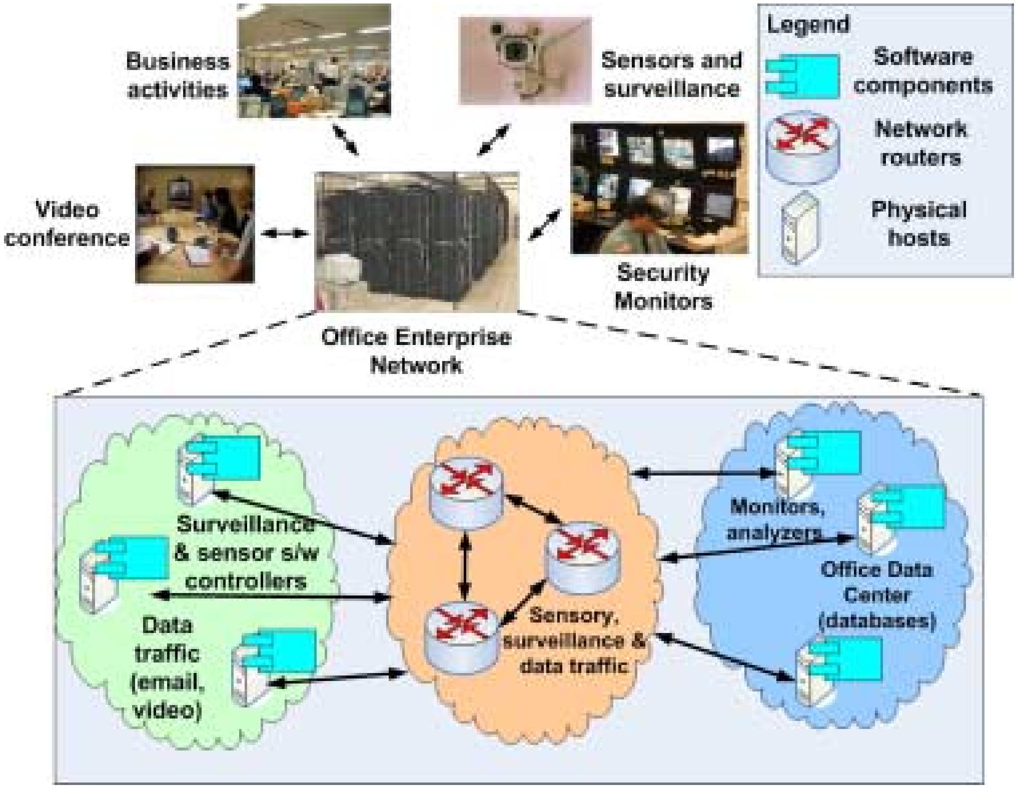
- Use case drawn from DARPA ARMS program
- Dynamic resource management for mission critical applications
- Multiple application workflows replicated in multiple data centers





- SLICE scenario depicts an application workflow

DRE System Example 5: Modern Office

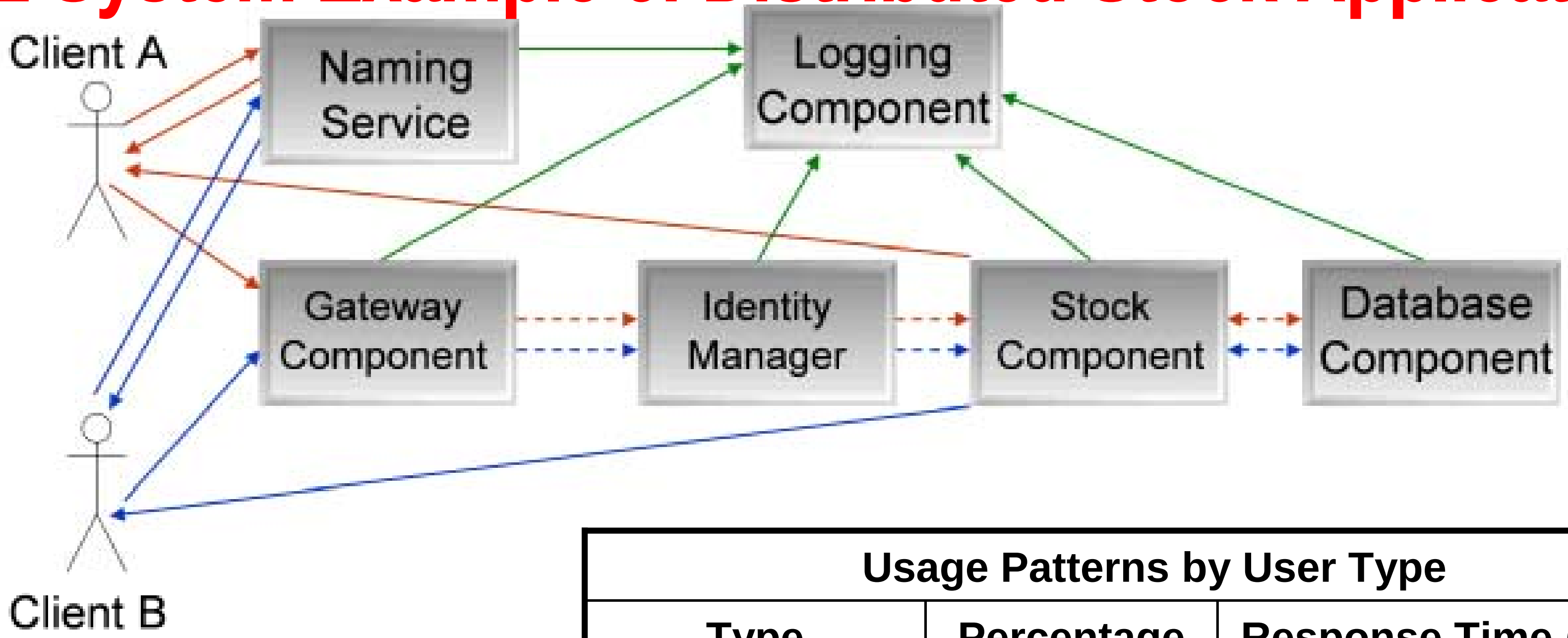


- Office traffic operates over IP networks & Fast ethernets
- Multiple application flows:
  - Email
  - Videoconferencing
  - Sensory (e.g., fire alarms)
- Differing network QoS requirements
  - Fire alarm – highest priority
  - Surveillance – multimedia
  - Temperature sensing – best effort
- QoS provisioned using DiffServ

Network QoS Provisioning Steps

1. Specify network QoS requirements for each application flow
2. Allocate network-level resources and DiffServ Code Points (DSCP) for every application flow joining two end points
3. Mark outgoing packet with the right DSCP values

DRE System Example 6: Distributed Stock Application



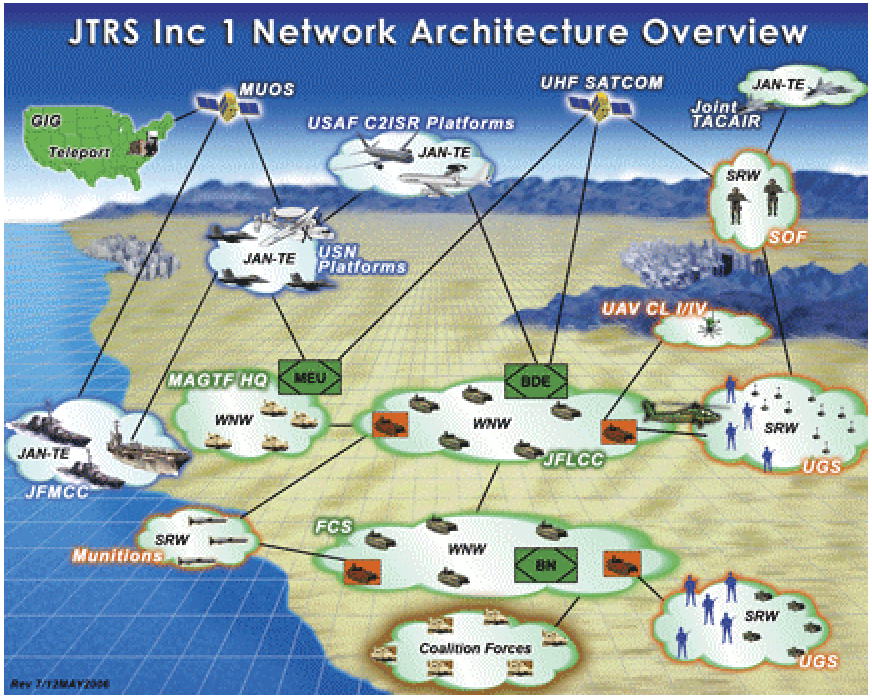
Usage Patterns by User Type		
Type	**Percentage**	**Response Time (msec)**
Basic (Client A)	65%	300
Gold (Client B)	35%	150

Design & Implementation

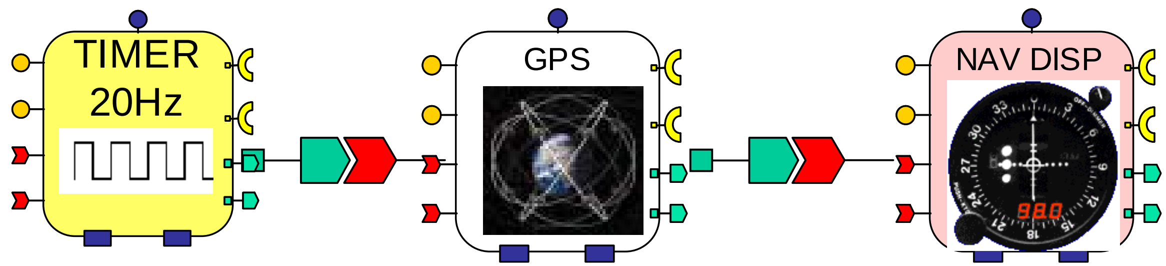
- Functionality developed as components
- Target architectures include multiple different nodes.
- Each component in the DSA is scheduled to complete its development at different times in development lifecycle

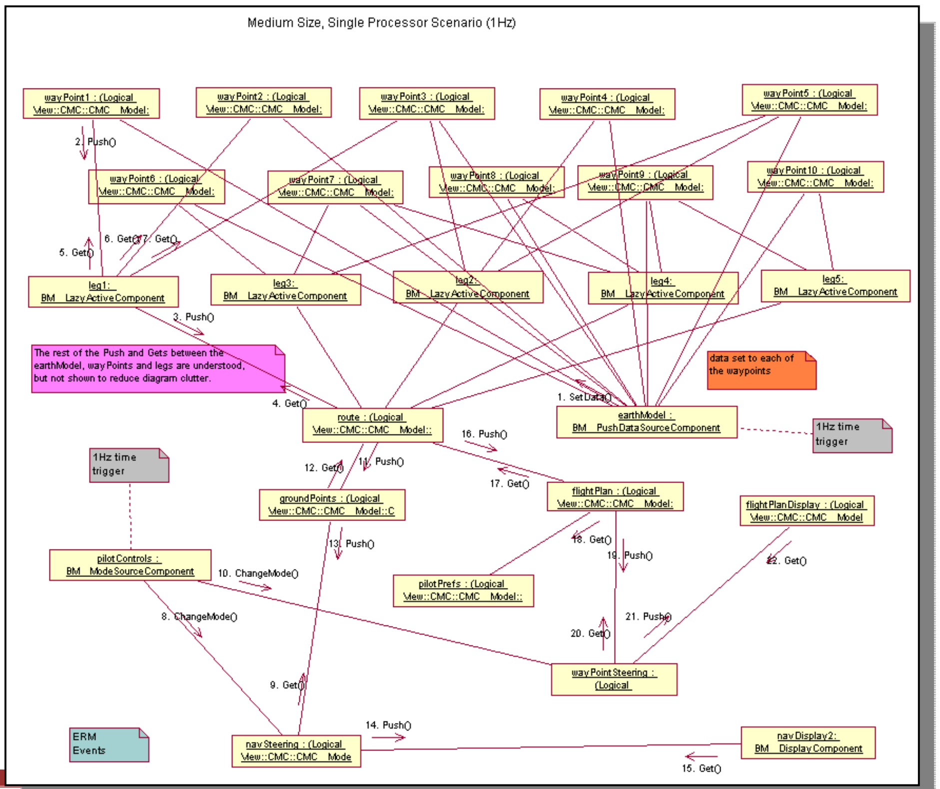
Part 3
Modeling DRE Application Workflows

Assembly & Packaging



Assembly & Packaging Problem

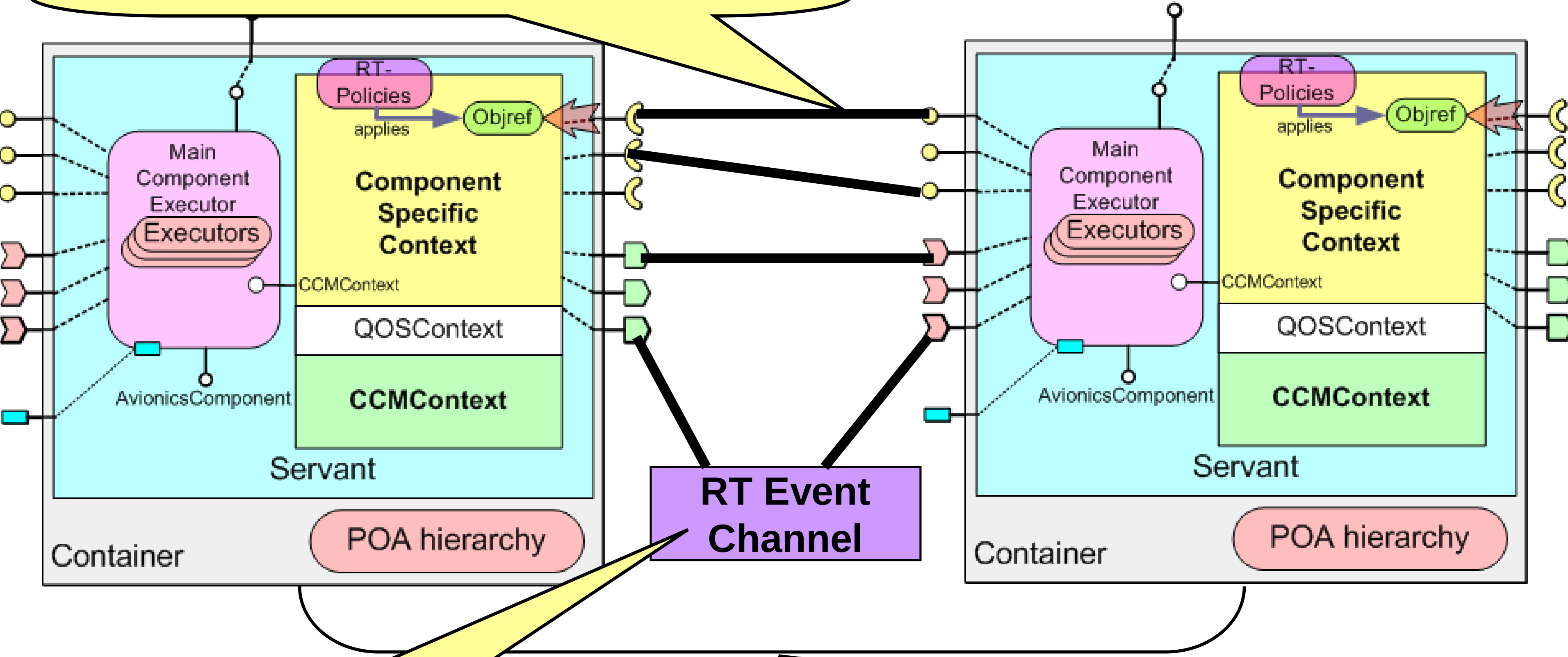




- Application components are bundled together into *assemblies*
- Several different assemblies tailored towards delivering different end-to-end QoS and/or using different algorithms can be part of the package
  - e.g., large-scale DRE systems require 100s-1,000s of components
- Packages describing the components & assemblies can be scripted via XML descriptors

Assembly & Packaging Challenges (1/2)

Ad hoc techniques for ensuring component syntactic & semantic compatibility



Ad hoc means to determine event notification support

Distribution & deployment done in ad hoc manner

Assembly & Packaging Challenges (2/2)

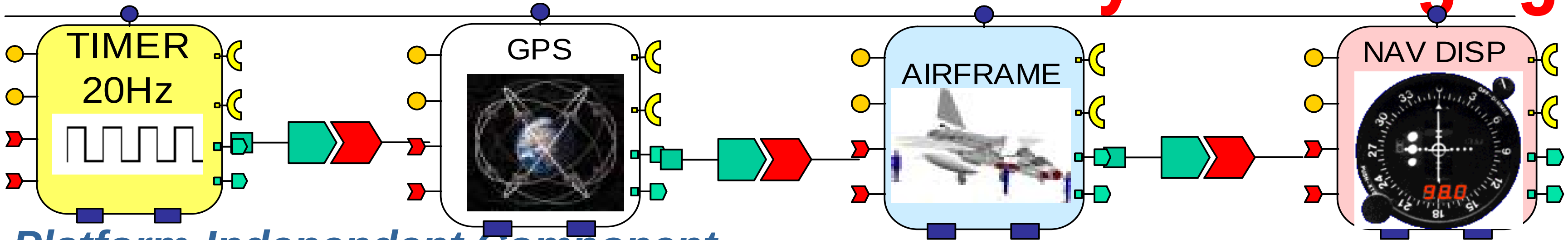
Existing practices involve handcrafting XML descriptors

```
<!– Associate components with impls -->
<componentfiles>
    <componentfile id=“RateGenerator">
      <fileinarchive name=“HouseRateGen.csd"/>
    </componentfile>

    <componentfile id=“HiResGPS">
      <fileinarchive name=“aGPS.csd"/>
    </componentfile>

    <componentfile id=“cockpitDisplay">
      <fileinarchive name=“navDisplay-if.csd"/>
    </componentfile>

 </componentfiles>
```

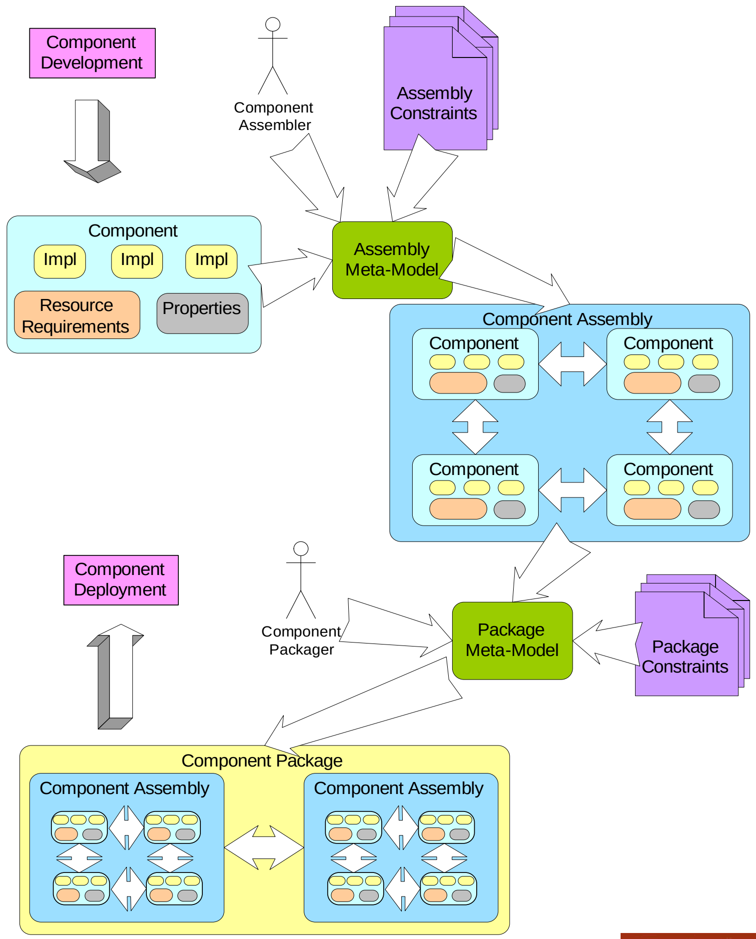
XML file in excess of 3,000 lines, even for medium sized scenarios

Modifications to the assemblies requires modifying XML file

CoSMIC MDE Solution for Assembly & Packaging

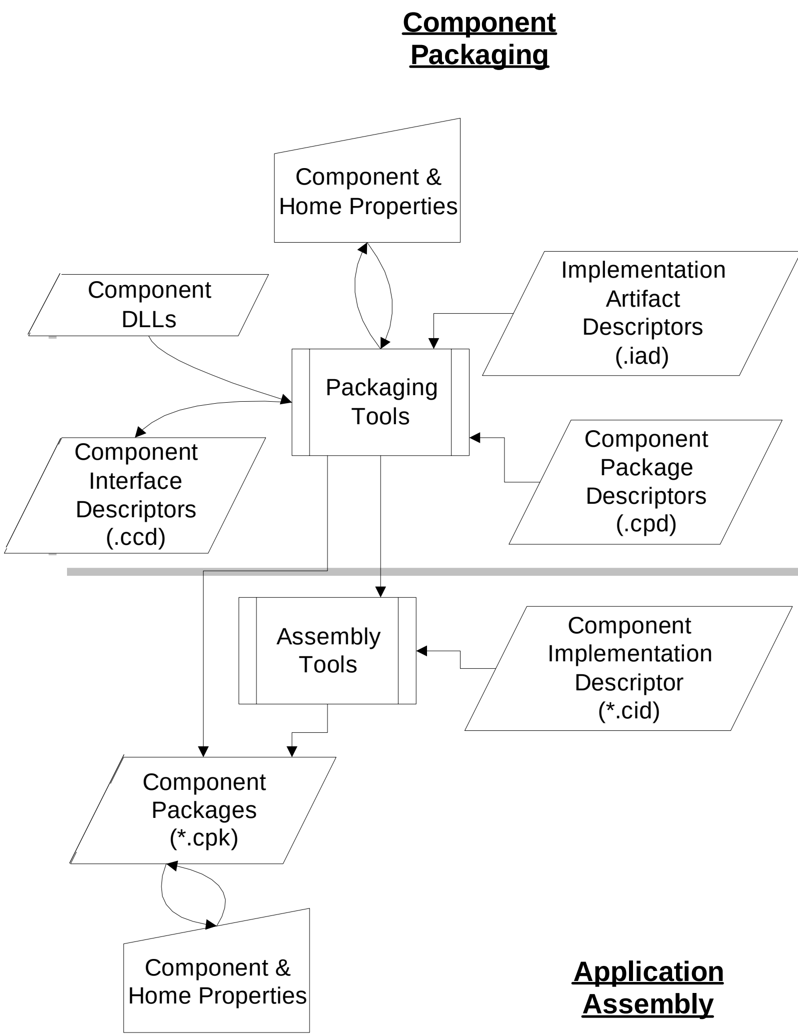


- ***Platform-Independent Component Modeling Language* (PICML)**
- Developed in Generic Modeling Environment (GME)
- Core of Component Synthesis using Model-Integrated Computing (CoSMIC) toolchain
- Capture elements & dependencies visually
- Define "static semantics" using Object Constraint Language (OCL)
- Define "dynamic semantics" via model interpreters
  - Also used for generating domain specific meta-data
- "Correct-by-construction"



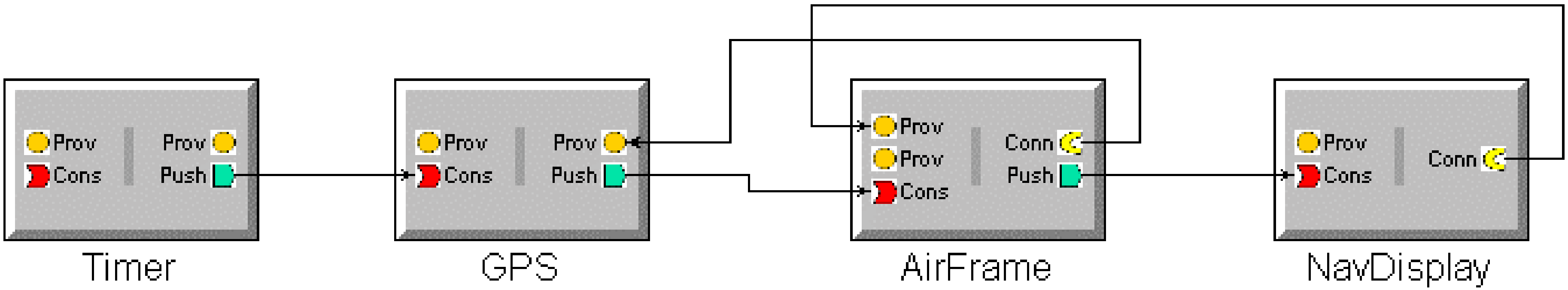
Example Metadata Generated by PICML

- **Component Interface Descriptor (.ccd)**
  - Describes the interface, ports, properties of a single component
- **Implementation Artifact Descriptor (.iad)**
  - Describes the implementation artifacts (e.g., DLLs, OS, etc.) of one component
- **Component Package Descriptor (.cpd)**
  - Describes multiple alternative implementations of a single component
- **Package Configuration Descriptor (.pcd)**
  - Describes a configuration of a component package
- **Top-level Package Descriptor (package.tpd)**
  - Describes the top-level component package in a package (.cpk)
- **Component Implementation Descriptor (.cid)**
  - Describes a specific implementation of a component interface
  - Implementation can be either monolithic- or assembly-based
  - Contains sub-component instantiations in case of assembly based implementations
  - Contains inter-connection information between components
- **Component Packages (.cpk)**
  - A component package can contain a single component
  - A component package can also contain an assembly



Based on OMG (D&C) specification (ptc/03-06-03)

Example Output from PICML



A Component Implementation Descriptor (*.cid) file

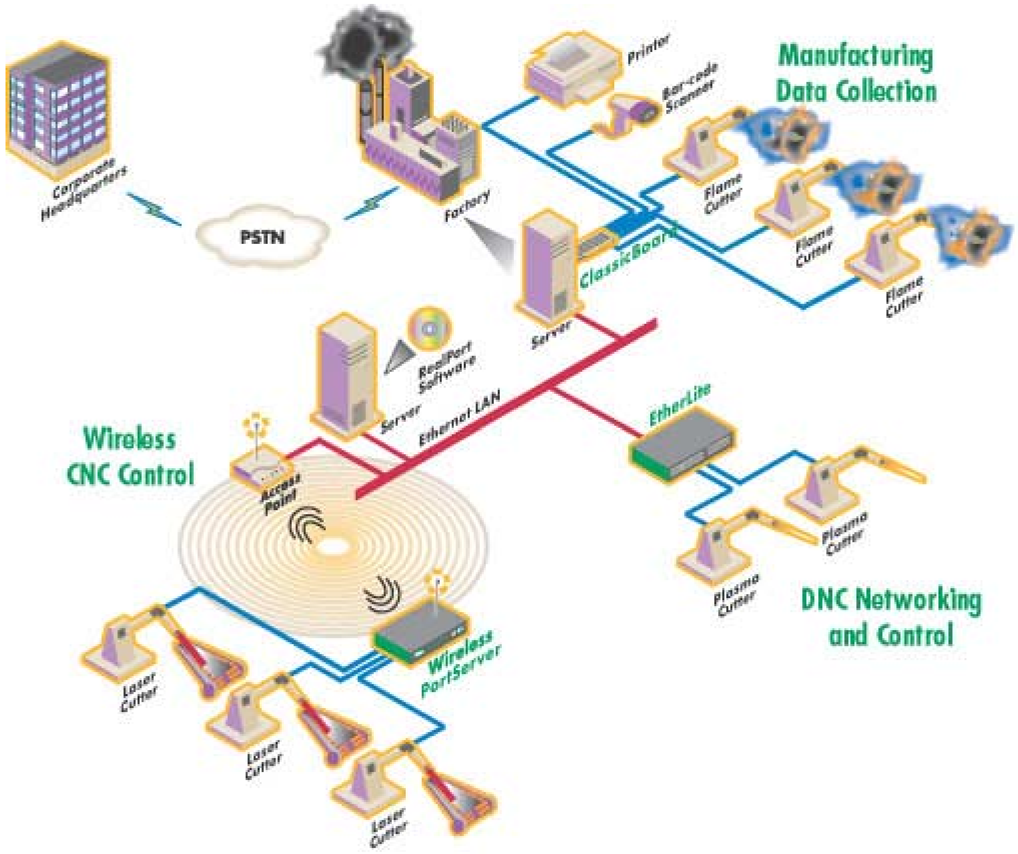
- Describes a specific implementation of a component interface
- Contains inter-connection information between components

```
<!--Component Implementation Descriptor(.cid) associates components
with impl. artifacts-->
<Deployment:ComponentImplementationDescription>
  <label>GPS Implementation</label>
  <UUID>154cf3cd-1770-4e92-b19b-8c2c921fea38</UUID>
  <implements href="GPS.ccd"/>
  <monolithicImpl>
     <primaryArtifact>
       <name>GPS Implementation artifacts</name>
       <referencedArtifact href="GPS.iad"/>
     </primaryArtifact>
  </monolithicImpl>
</Deployment:ComponentImplementationDescription>
```

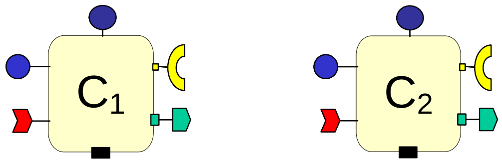
```
<ComponentAssemblyDescription id="a_HUDDisplay">
...
 <connection>
   <name>GPS-RateGen</name>
   <internalEndPoint>
      <portName>Refresh</portName>
      <instance>a_GPS</instance>
   </internalEndPoint>
   <internalEndPoint>
      <portName>Pulse</portName>
      <instance>a_RateGen</instance>
   </internalEndPoint>
  </connection>
  <connection>
    <name>NavDisplay-GPS</name>
    <internalEndPoint>
       <portName>Refresh</portName>
       <instance>a_NavDisplay</instance>
    </internalEndPoint>
    <internalEndPoint>
       <portName>Ready</portName>
       <instance>a_GPS</instance>
    </internalEndPoint>
  </connection>
...
</ComponentAssemblyDescription>
```

Part 4
DRE System QoS Provisioning

QoS Modeling



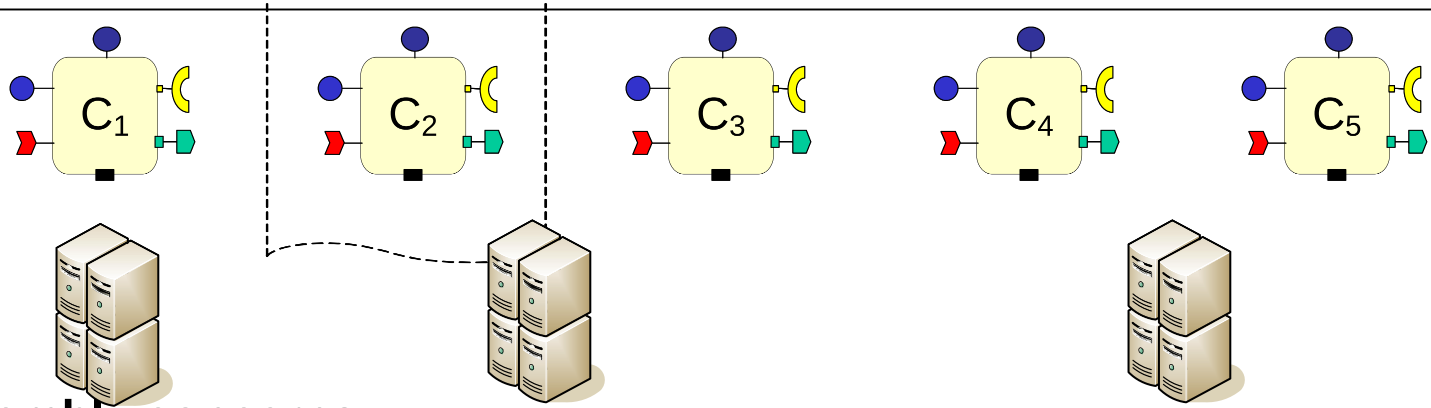
Sources of Variability Impacting QoS (1/3)



- **Understand the sources of variability in the problem/solution space that impacts QoS**
- **Find solutions to automate the mapping of the problem space to solution space**

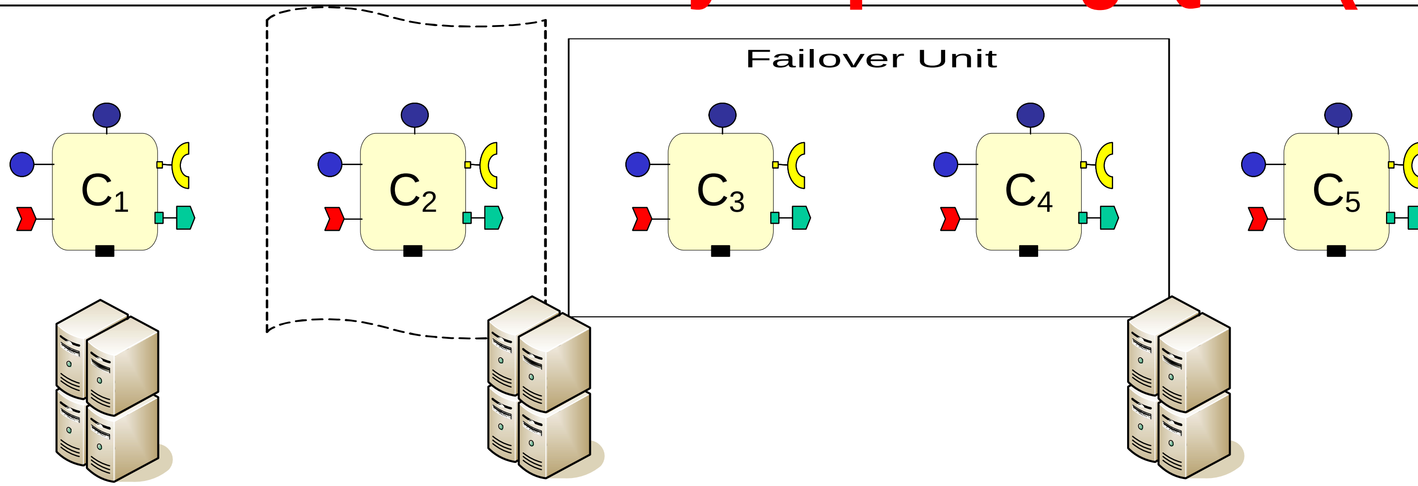
- **Per-component concerns**
  - *Problem space*: Functionality often supplied as third party COTS
  - *Solution space*: Implementations for different platforms and languages
  - *P->S mapping challenges*:
    - Implementations must match platform and language choice
    - Choice of implementation impacts end-to-end QoS
- **Communication concerns**
  - *Problem space*: Choice of communication paradigms (pub-sub, RPC, MOM) and requirements on QoS
  - *Solution space*: Diversity in communication mechanisms e.g, CORBA IIOP, Java RMI, DDS, Event channels, and QoS mechanisms e.g., DiffServ, IntServ, MPLS
  - *P->S mapping challenges:*
    - Decoupling application logic from provisioning communication QoS
    - Right optimizations in communication paths needed to enhance QoS
    - Redundancy and mixed mode communications needed to support QoS

Sources of Variability Impacting QoS (2/3)



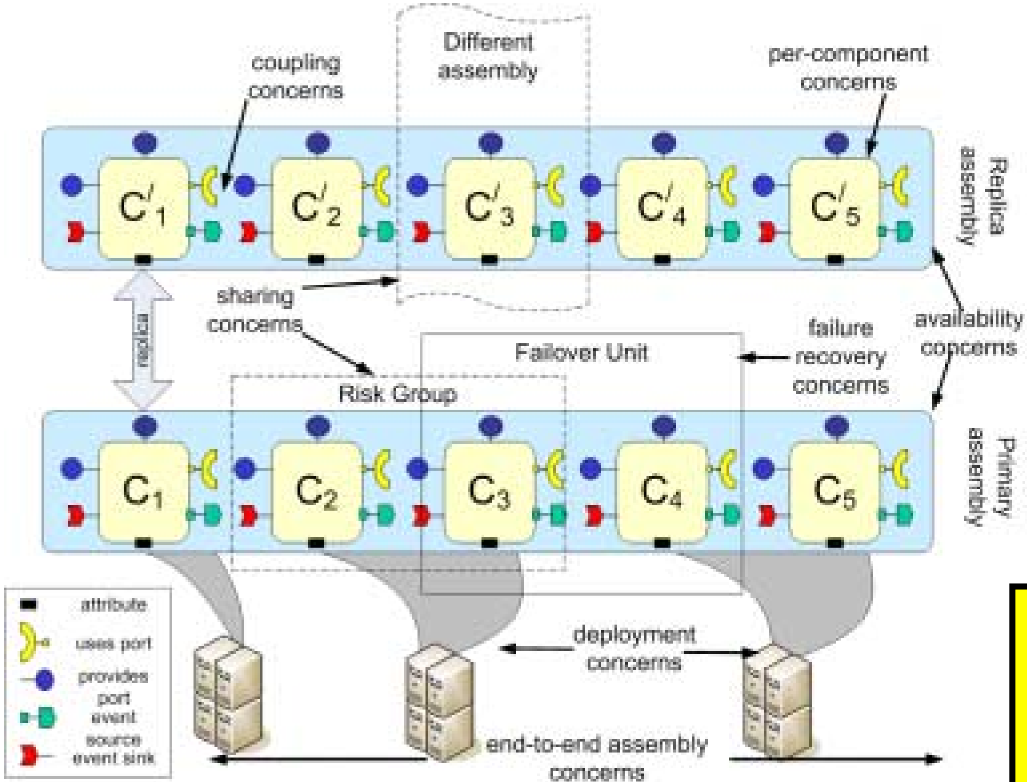
- **Assembly concerns**
  - *Problem space*: Discovering services and composing them together
  - *Solution space*: Interfaces and their semantics must match
  - *P->S mapping challenges*:
    - Must ensure functional and systemic compatibility in composition
    - Heterogeneity in platforms and languages in composition impacts QoS
- **Deployment concerns**
  - *Problem space*: Need resources e.g., CPU, memory, bandwidth, storage
  - *Solution space*: Diversity in resource types and their configurations
  - *P->S mapping challenges*:
    - How to select and provision resources such that end-to-end QoS is realized?
    - How to (re)allocate resources to maintain end-to-end QoS?
    - How to place components so that near optimal QoS tradeoffs are made?
    - How to allow sharing of components or assemblies?

Sources of Variability Impacting QoS (3/3)



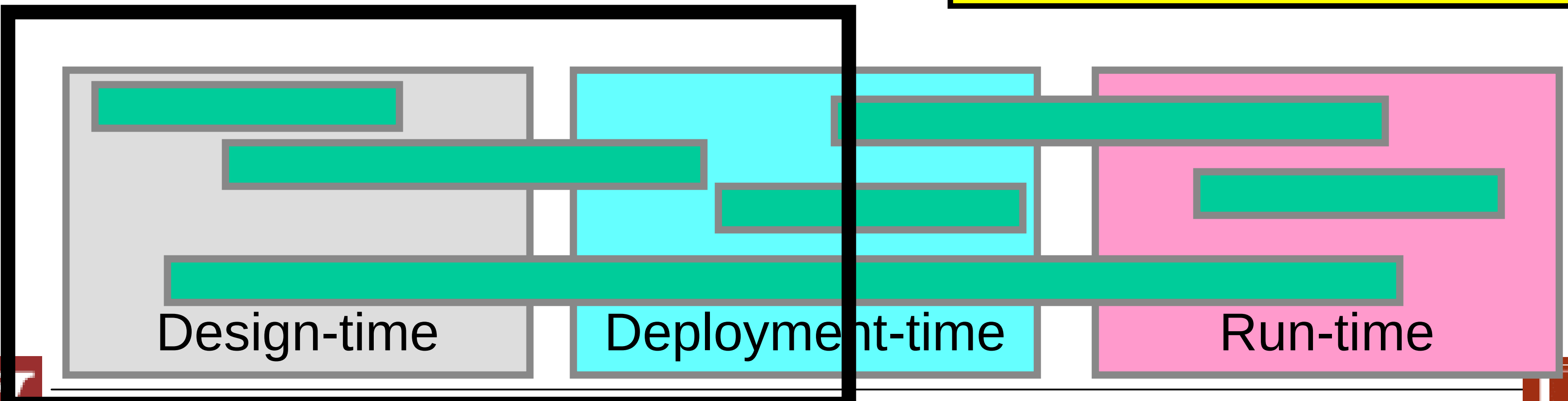
- **Configuration concerns**
  - *Problem space*: Multiple QoS requires right set of configurations
    - What is the unit of failover? What is the replication degree? Are entire assemblies replicated?
    - How to define access control rights?
    - Determining system task partitioning, schedulability
    - Defining network resource needs
  - *Solution space*: Platforms provide high degree of configuration flexibility
  - *P->S mapping challenges:*
    - What configuration options control a specific dimension of QoS?
    - How do different configuration options interact with each other?
    - How to configure heterogeneous platforms?
    - What impact deployment decisions have on configurations?

A Single Perspective: Tangling of QoS Issues

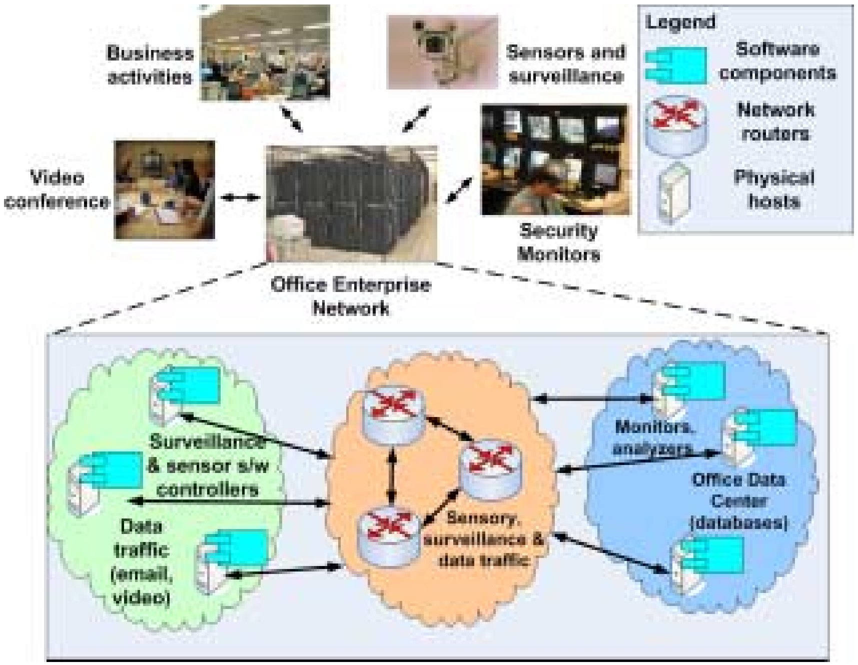


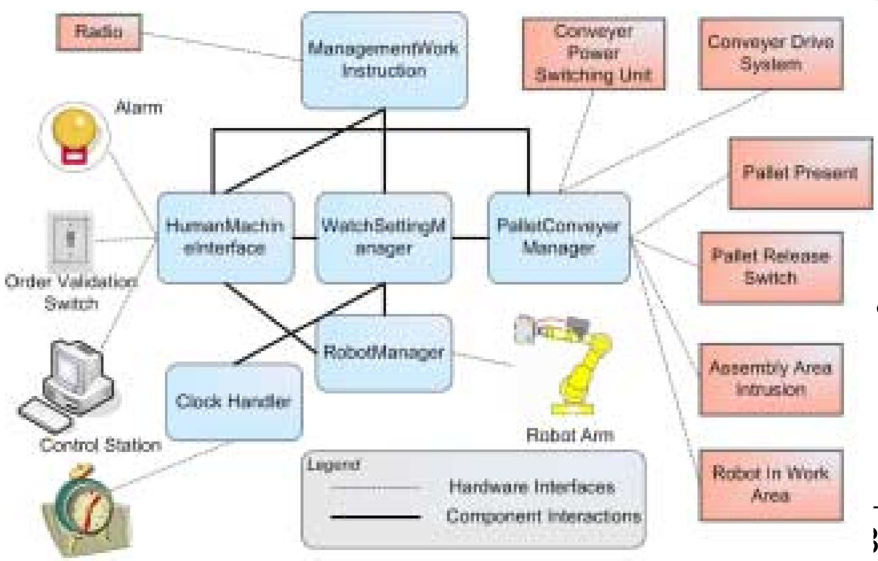
- Demonstrates numerous tangled para-functional concerns
- Significant sources of variability that affect end-to-end QoS
- QoS concerns tangled across system lifecycle

Lifecycle-wide separation of concerns (SoC) & variability management is the key



Requirements of a Solution Approach



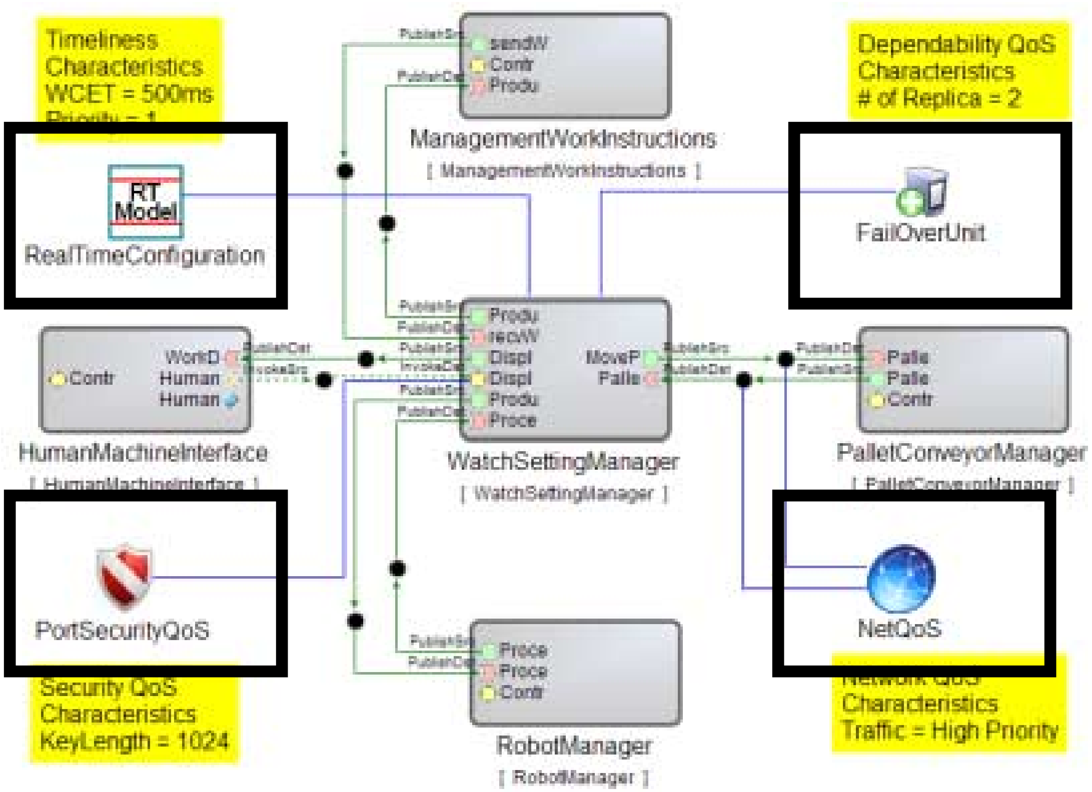


- Need expressive power to define QoS intent in the *problem space*, and perform design-time analysis
    - e.g., network QoS needs of application flows or end-to-end latencies
- Need to decouple DRE system functionality from systemic capabilities in the *solution space*
    - e.g., decouple robot manager logic from fault tolerance configurations
- Need to automate the mapping from problem to solution space
    - e.g., automate the (re)deployment and (re)configuration of system functionality to maintain QoS
- Need to provide a hosting platform with dynamic adaptation capabilities
    - e.g., survivability management of robot manager

Component QoS Modeling Language (CQML)

Objectives:

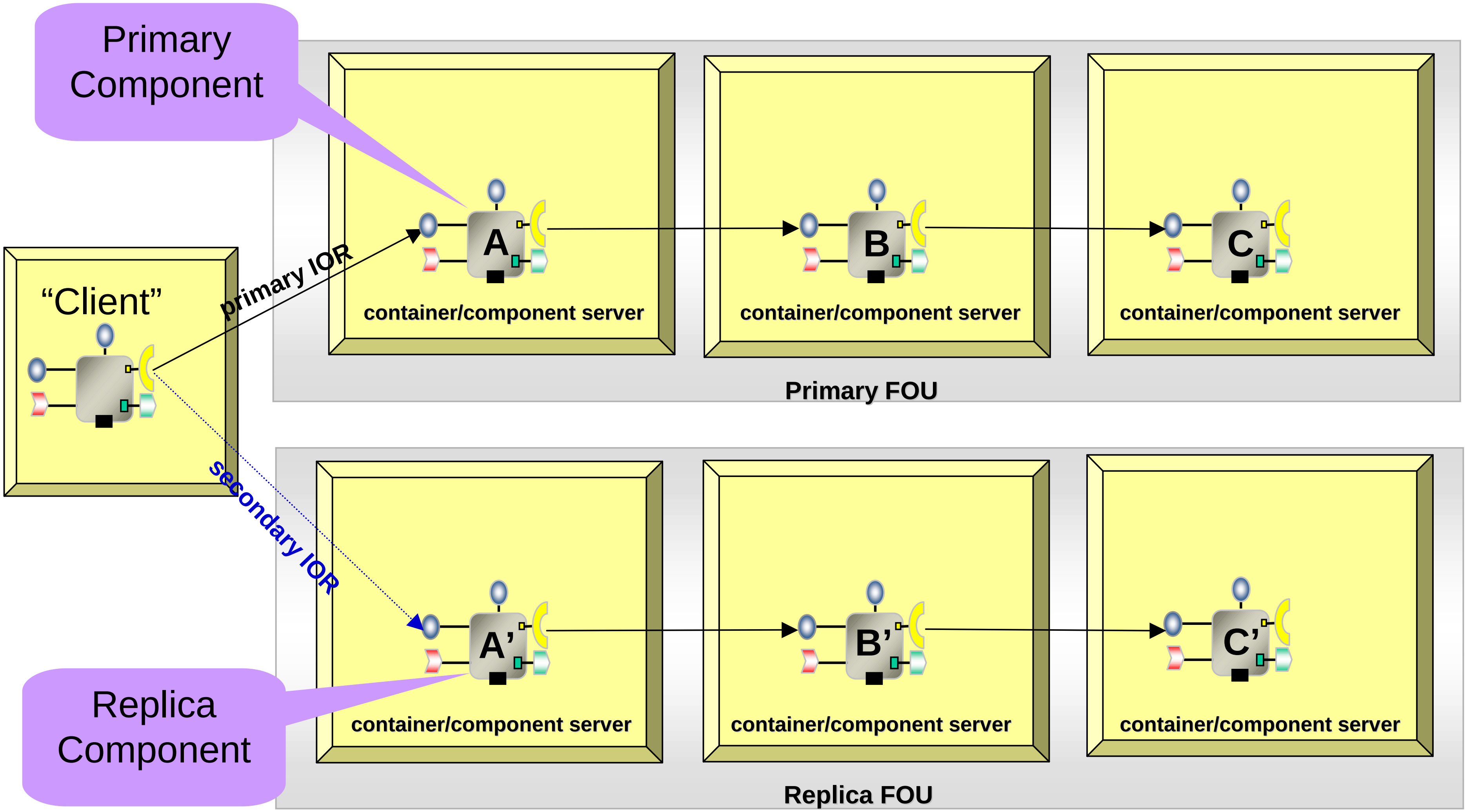
- Express QoS design intent
- Model crosscutting QoS concerns
- Perform design-time QoS tradeoff analysis



Challenges:

- Intuitive QoS abstractions
- Separation as well as unification of QoS concerns
  - 4 types of QoS modeling capabilities
    - Real-time
    - Fault tolerance
    - Security
    - Network QoS
  - Developed as overlay on a composition modeling language
  - Facility to integrate behavioral modeling
  - Unified internal mechanism for QoS integration
  - Pluggable back end analysis tools for QoS tradeoff analysis
    - e.g., Times tool

Example of Failover Unit Intent



CQML FT Modeling Abstractions

QoS Enhancements to PICML (Platform Independent Component Modeling Language) Metamodel

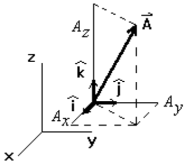
- **Fail-over Unit (FOU):** Abstracts away details of granularity of protection (e.g. Component, Assembly, App-string)
- **Replica Group (RPG):** Abstracts away fault-tolerance policy details (e.g. Active/passive replication, state-synchronization, topology of replica)
- **Shared Risk Group (SRG):** Captures associations related to risk. (e.g. shared power supply among processors, shared LAN)

Model Interpreter (component placement constraint solver): Encapsulates an algorithm for component-node assignment based on replica distance metric

Protection granularity concerns

State-synchronization concerns

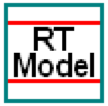
Component Placement constraints

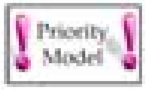


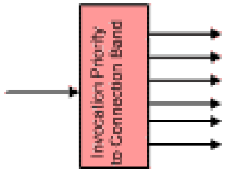
Replica Distance Metric

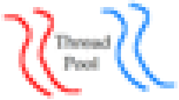
CQML RT & NetQoS Modeling Abstractions

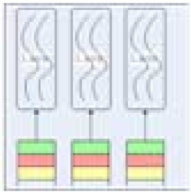
- Real-time modeling provided by the RT-Model element
  - Leverages known patterns of real-time programming usage:
    - e.g., real-time CORBA
  - Enables modeling of:
    - Priority models
    - Banded connections
    - Thread pools
    - Thread pools with Lanes
    - Resources e.g., CPU, memory
- Network QoS modeling allows modeling QoS per application flow
  - Classification into high priority, high reliability, multimedia and best effort classes
  - Enables bandwidth reservation in both directions
  - Client propagated or server declared models

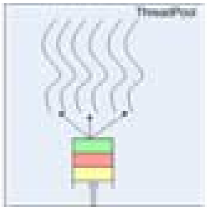






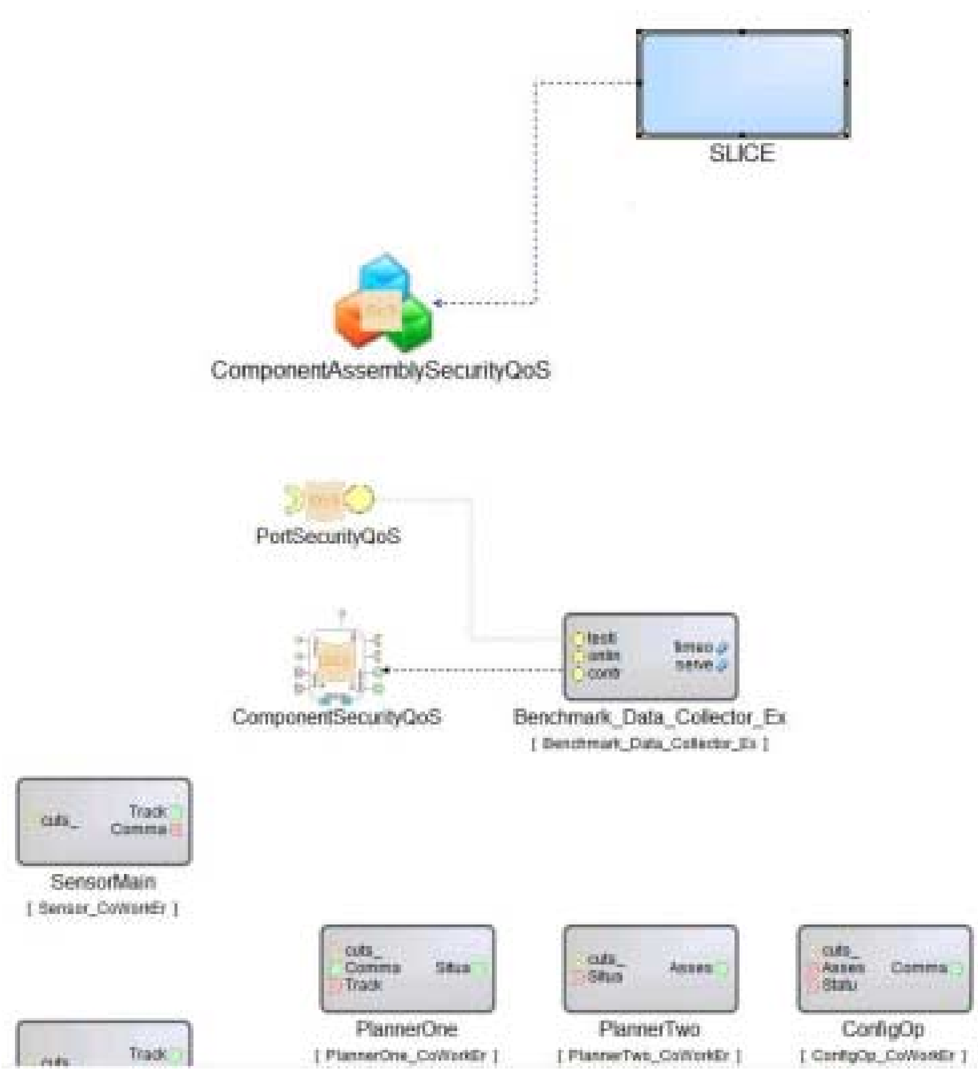






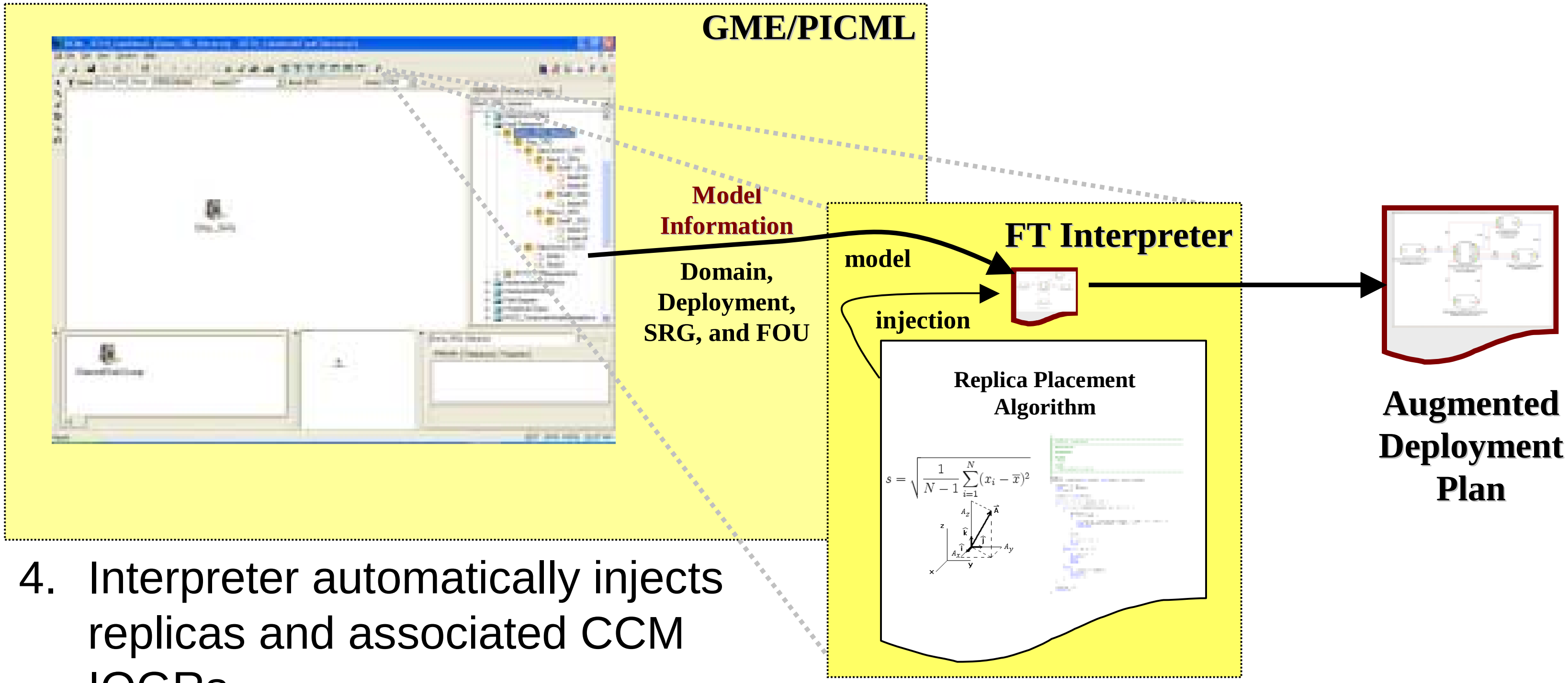
CQML Security Modeling Abstractions

- Focus on role based access control
- Fine-grained:
  - Interface Operation
  - Assembly Property
  - Component Attribute
- Coarse-grained:
  - Interface
  - Set of Operations
  - Class of Operations (based on Required Rights - corba:gsum)
  - Inter-Component Execution Flow (Path in an Assembly)



Modeling & Generative Steps in CQML

1. Model components and application strings in PICML
2. e.g., Model Fail Over Units (FOUs) and Shared Risk Groups (SRGs) using CQML
3. Generate deployment plan



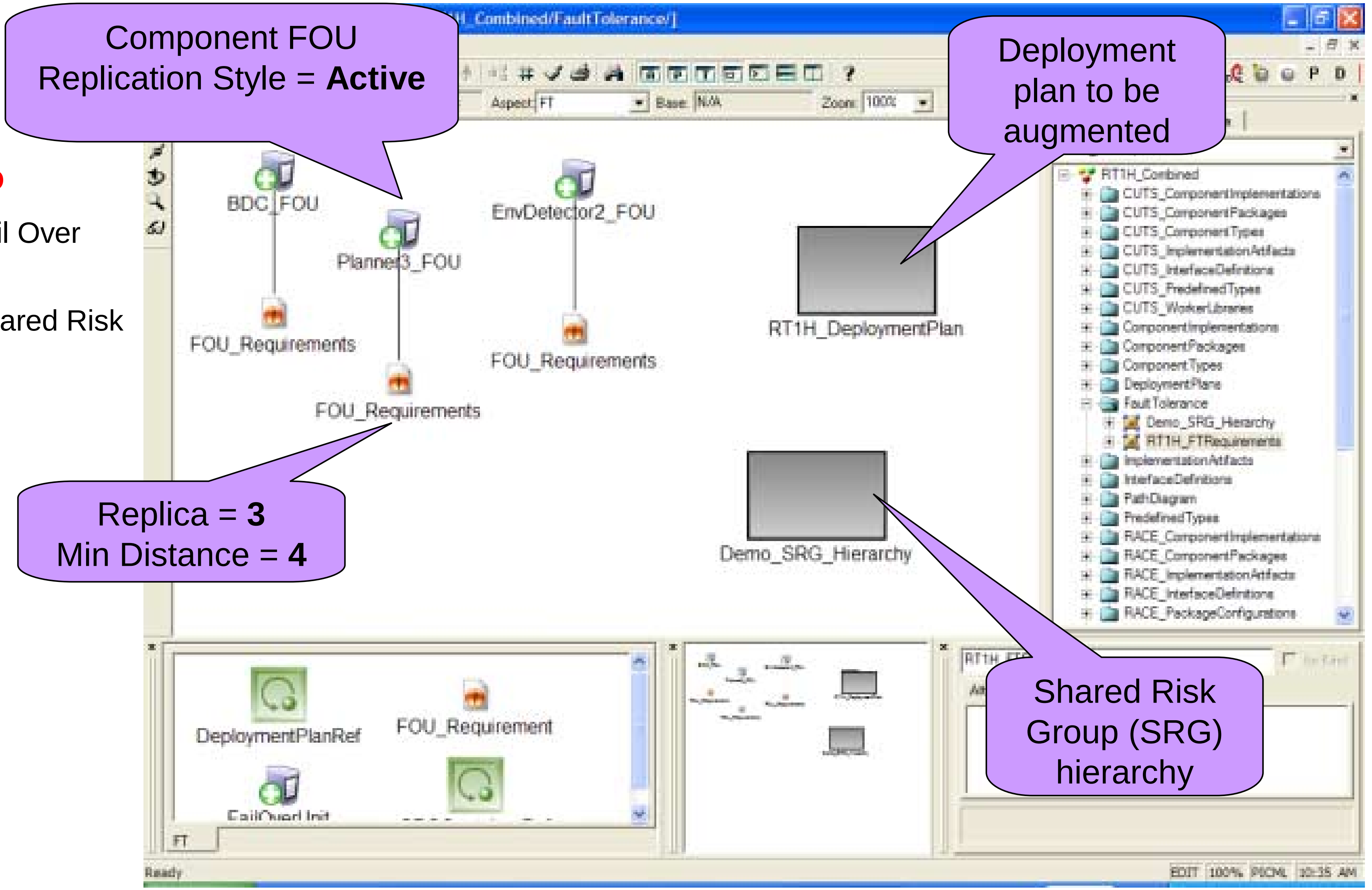
4. Interpreter automatically injects replicas and associated CCM IOGRs
5. Distance-based constraint algorithm determines replica placement in deployment descriptors.

Example: FT Requirements Modeling in CQML

LEGEND

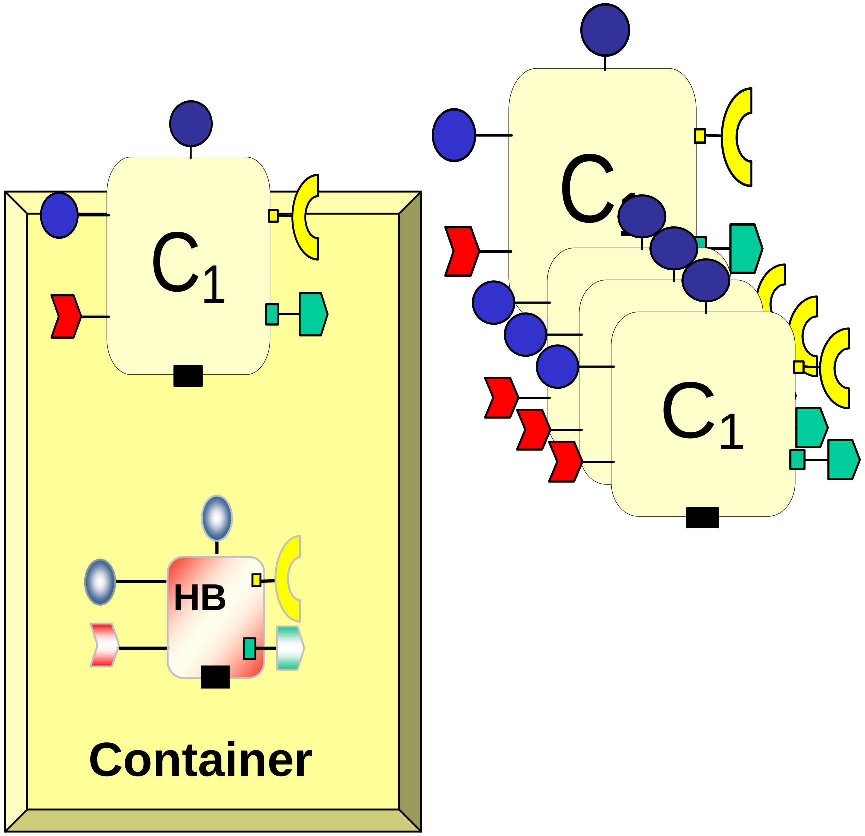
FOU: Fail Over Unit

SRG: Shared Risk Group

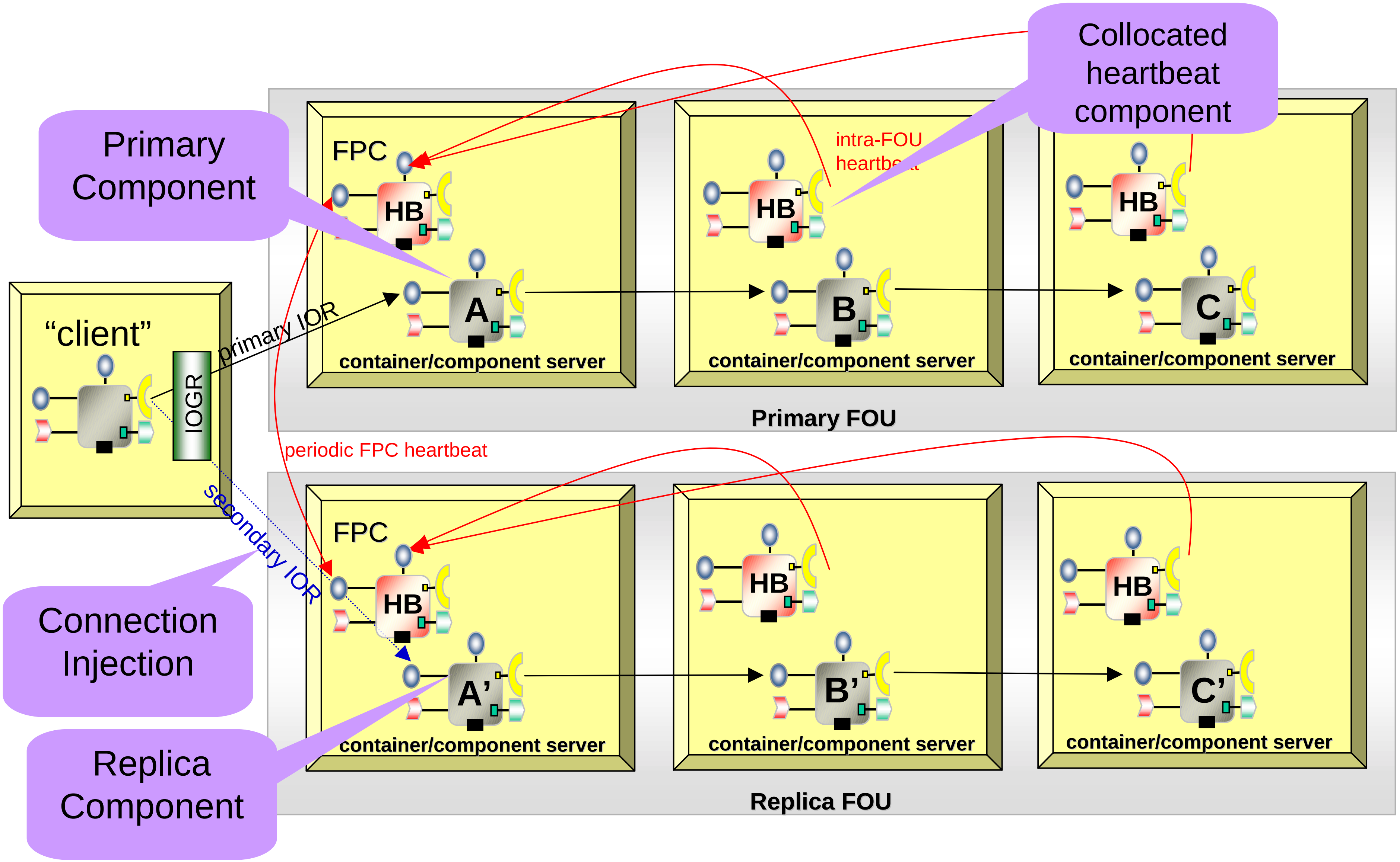


Automated Sample FT QoS Provisioning

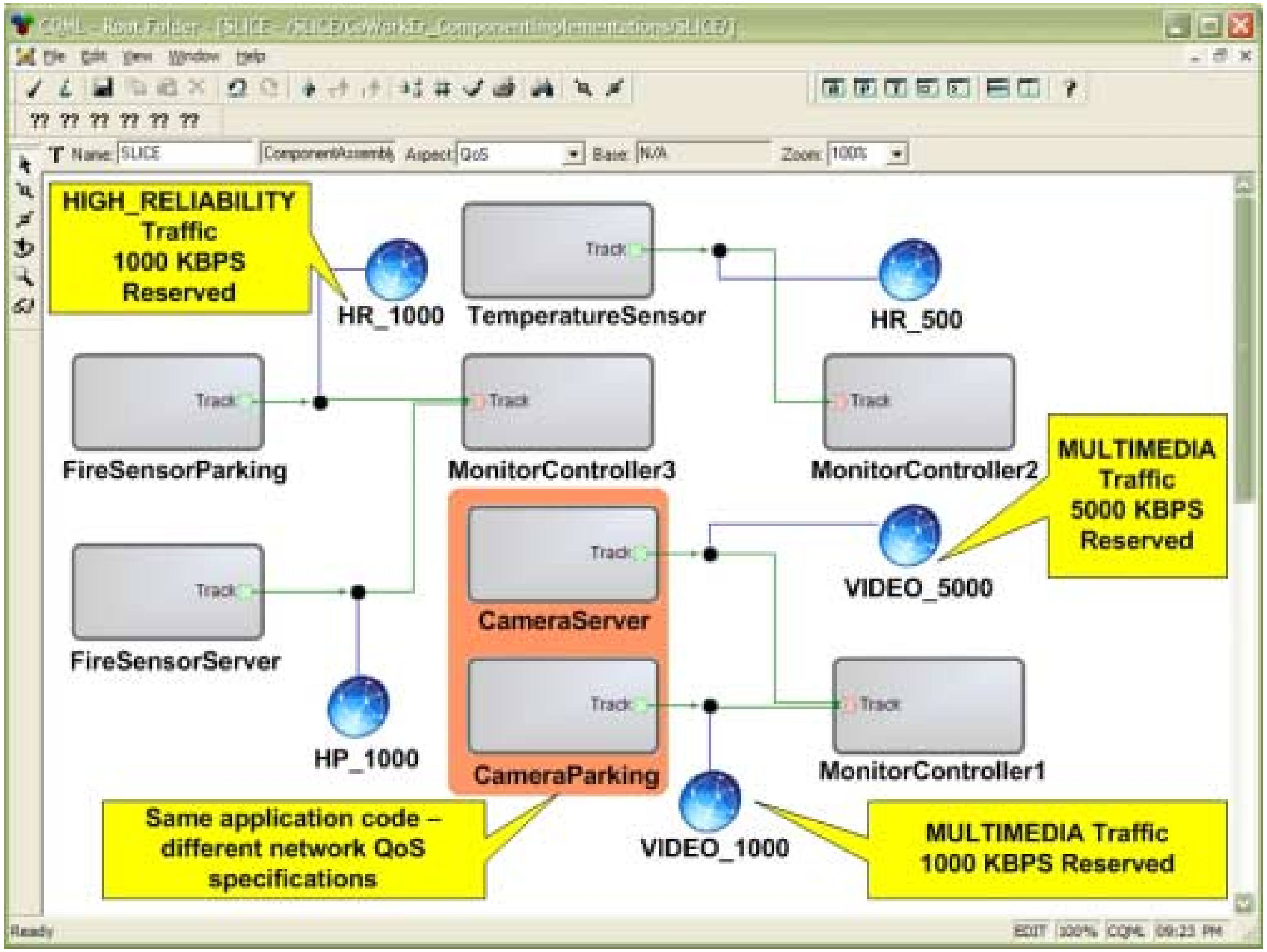
- Automatic Injection of replicas
  - Augmentation of deployment plan based on number of replicas
- Automatic Injection of FT infrastructure components
  - E.g. Collocated “heartbeat” (HB) component with every protected component.
- Automatic Injection of connection meta-data
  - Specialized connection setup for protected components (e.g. Interoperable Group References IOGR)



Automated Heartbeat Component Injection



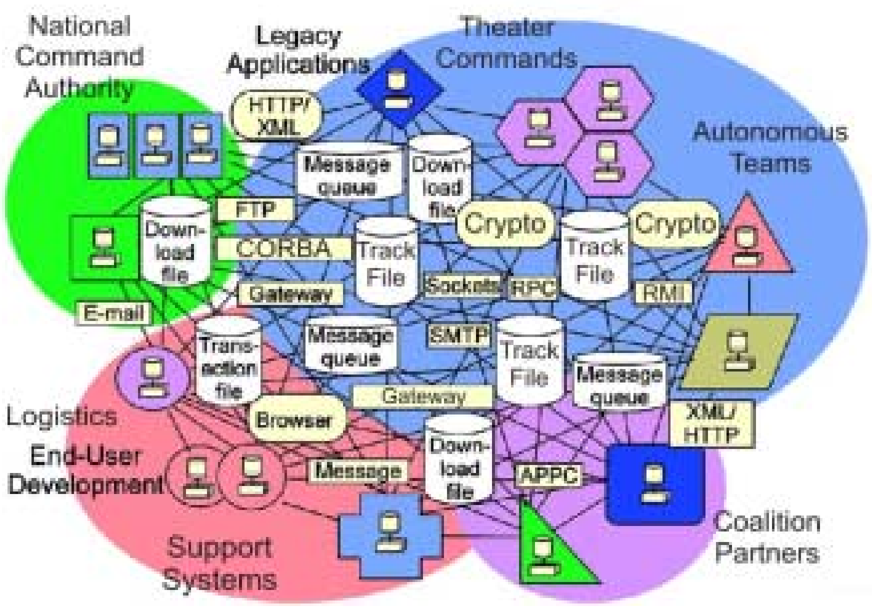
Example: NetQoS Modeling in CQML



- Expressing QoS intent for application flows
- Multiple connections sharing QoS can use same element

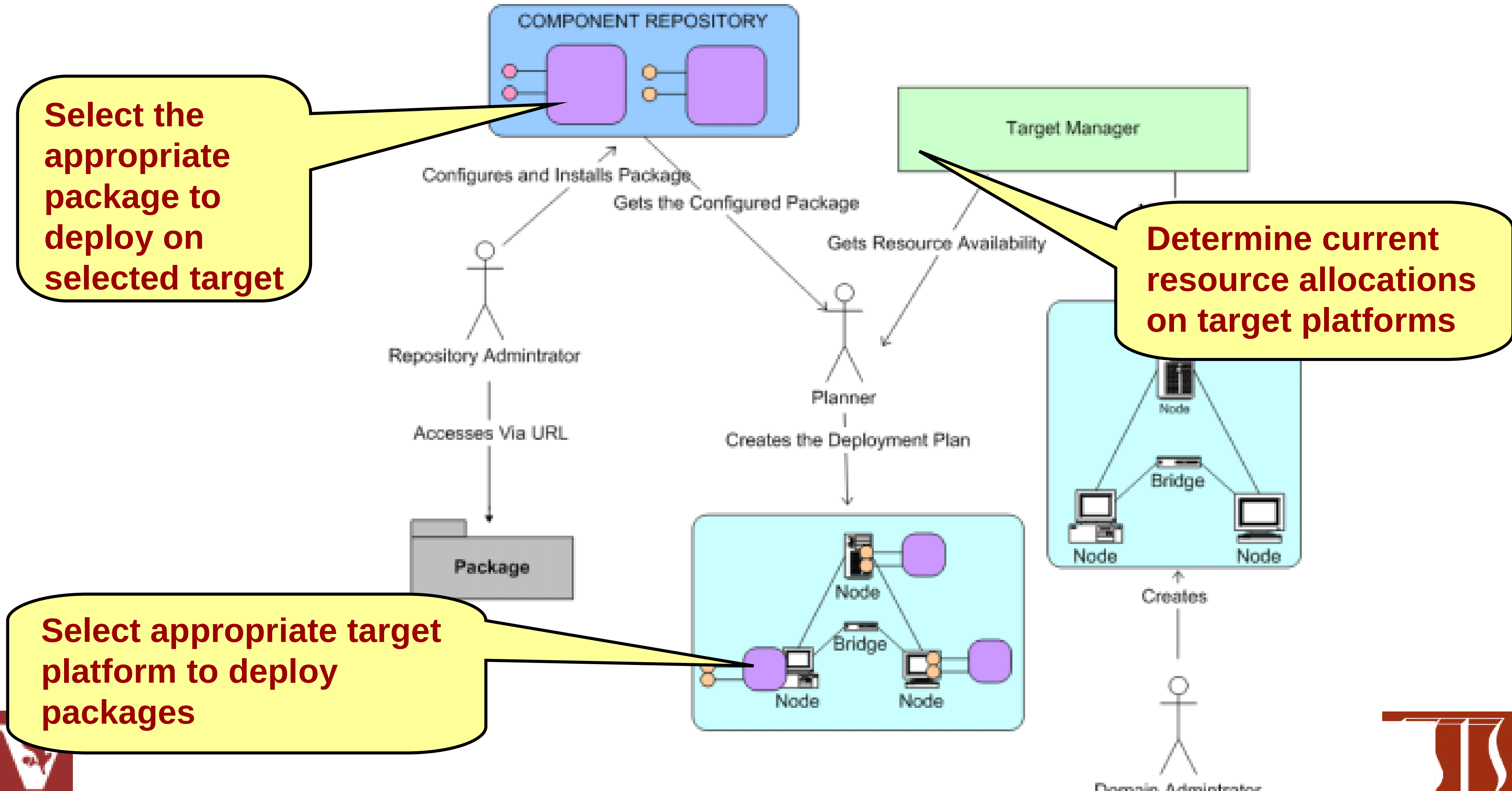
Part 5
DRE System Deployment Planning

Placement Algorithms

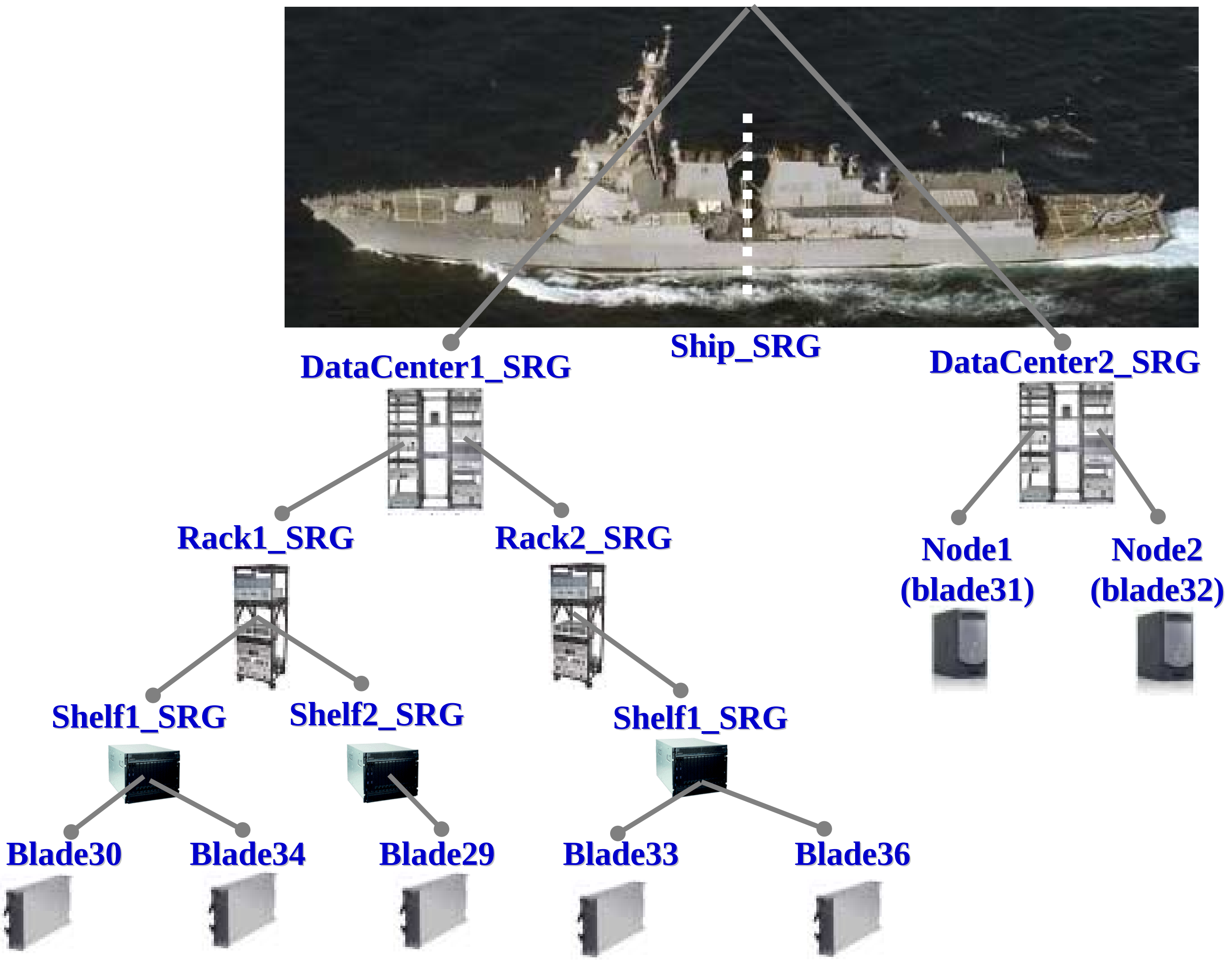


Deployment Planning Problem

Component integrators must make appropriate deployment decisions, including identifying the entities (e.g., CPUs) of the target environment where the packages will be deployed



Example of Shared Risk Group Intent

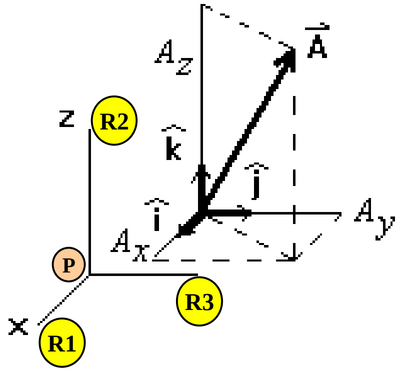


Example Replica Placement Algorithm

Define N orthogonal vectors, one for each of the distance values computed for the N components (with respect to a primary) and vector-sum these to obtain a resultant. Compute the magnitude of the resultant as a representation of the <u>composite distance</u> captured by the placement

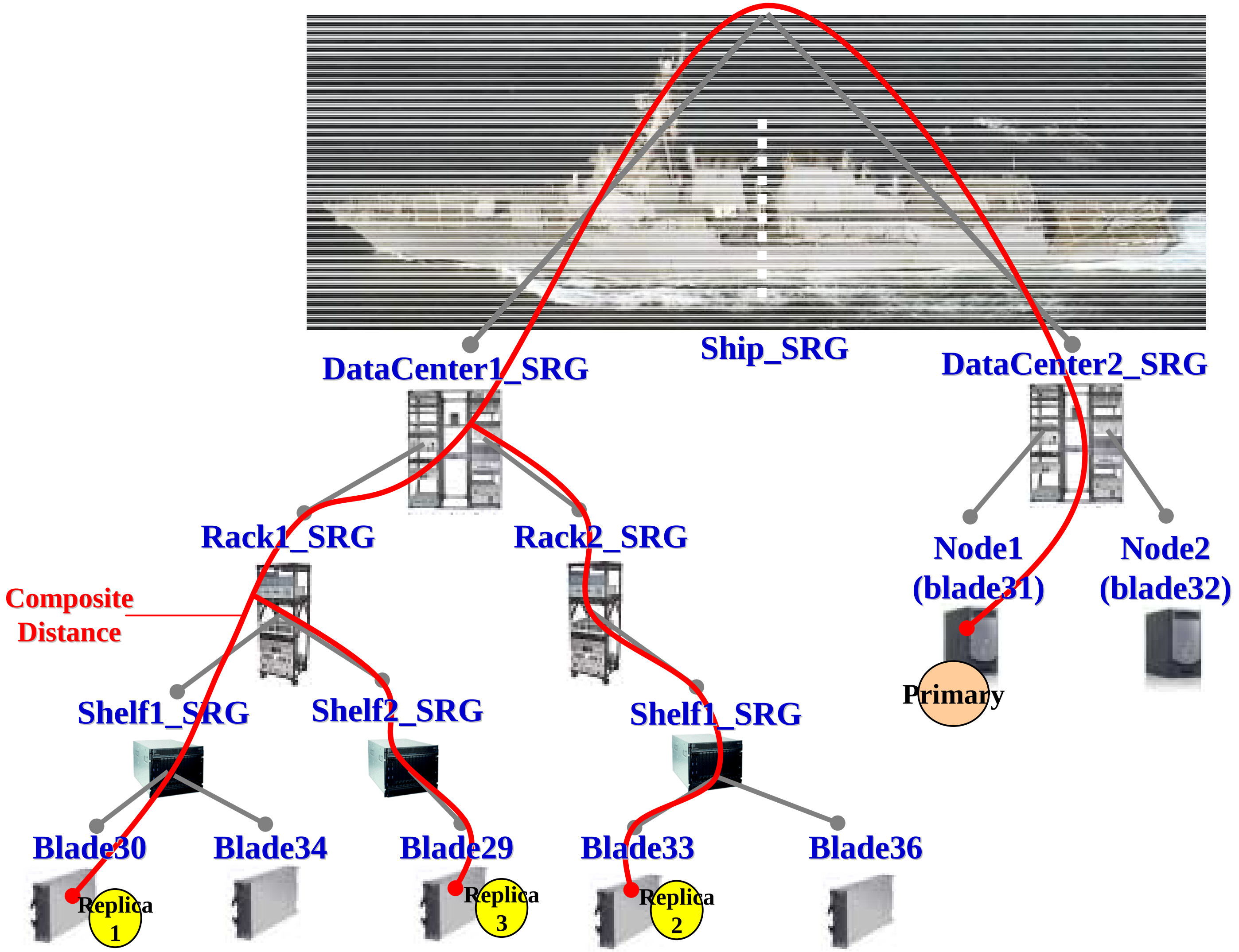
.

1. Compute the distance from each of the replicas to the primary for a placement.
2. Record ***each* distance as a vector, where** <u>all</u> vectors are orthogonal.
3. Add the vectors to obtain a resultant.
4. Compute the magnitude of the resultant.
5. Use the resultant in all comparisons (either among placements or against a threshold)
6. Apply a penalty function to the composite distance (e.g. pair wise replica distance or uniformity)



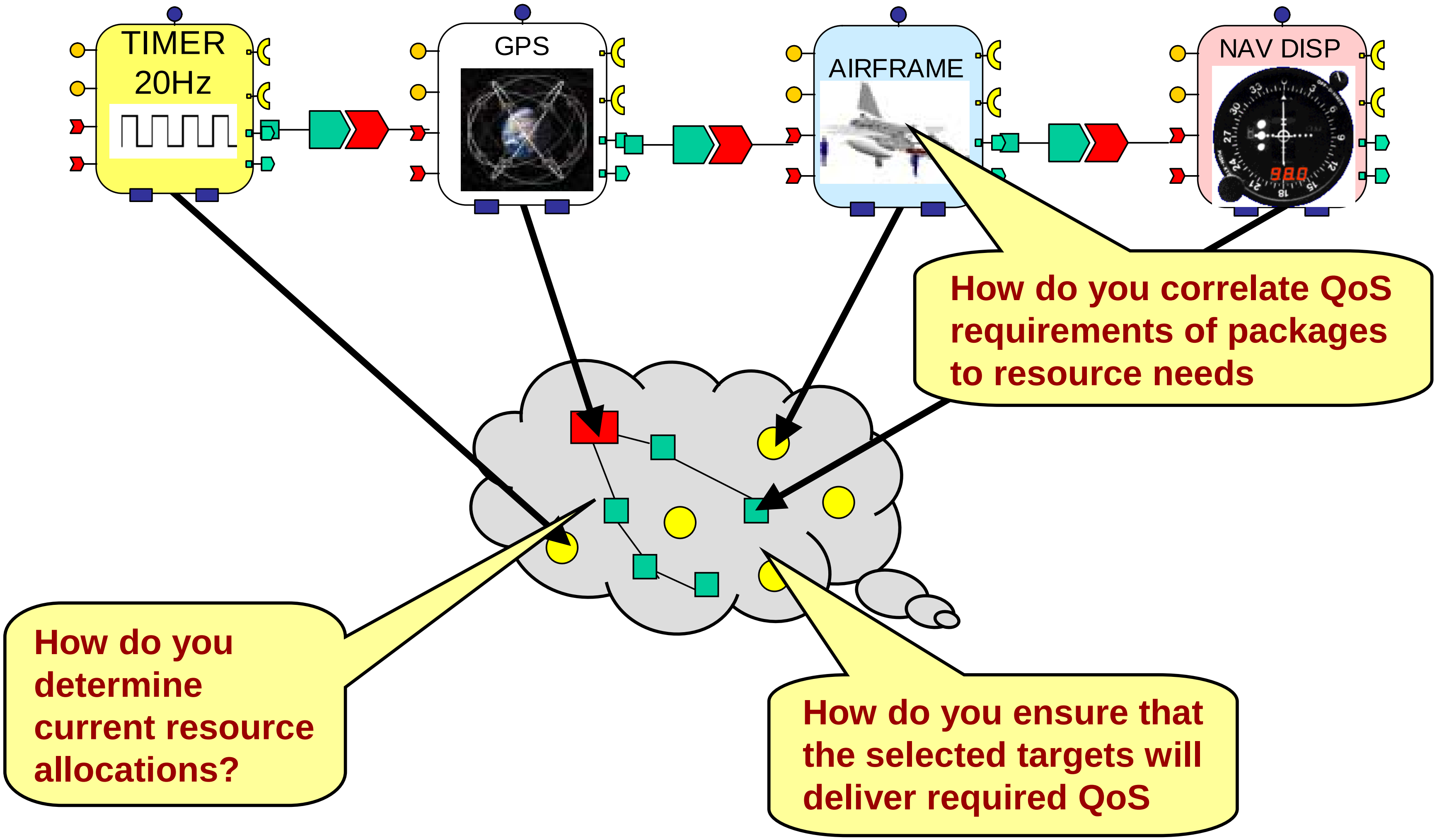
$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

Example: Automated Component Placement

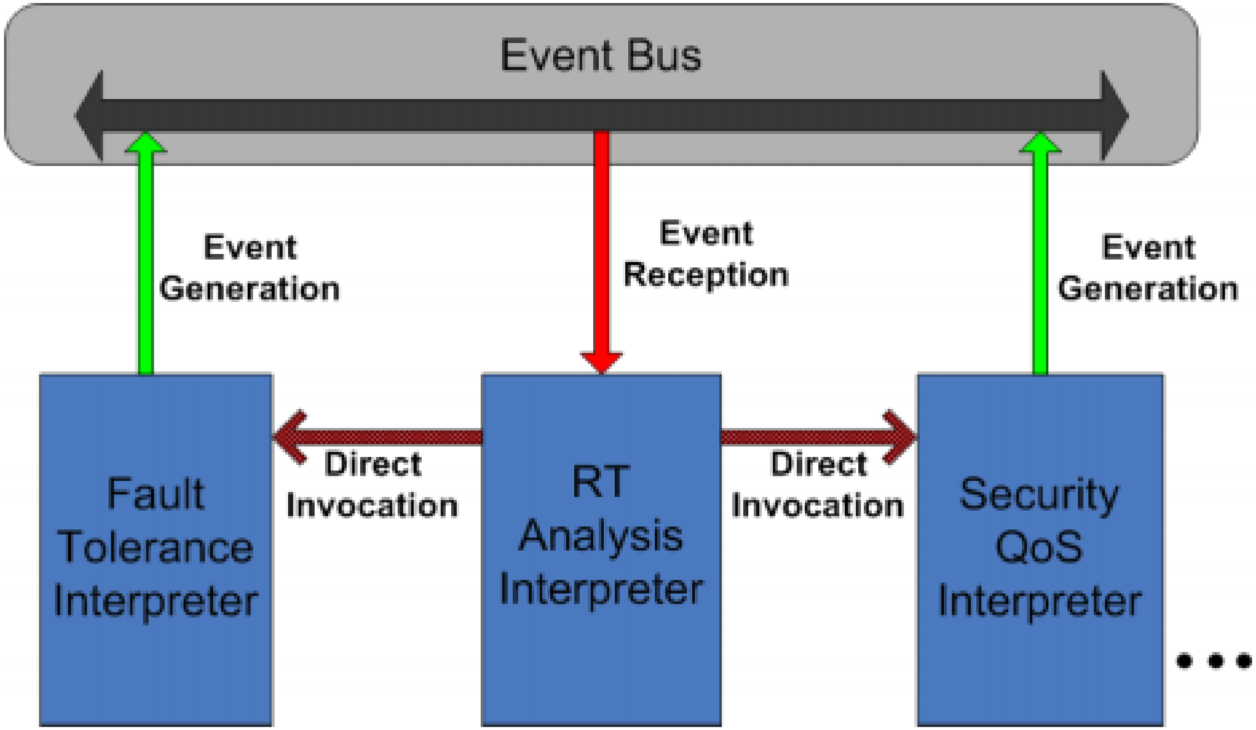


Deployment Planning Challenges

How to ensure a particular deployment configuration maximizes QoS

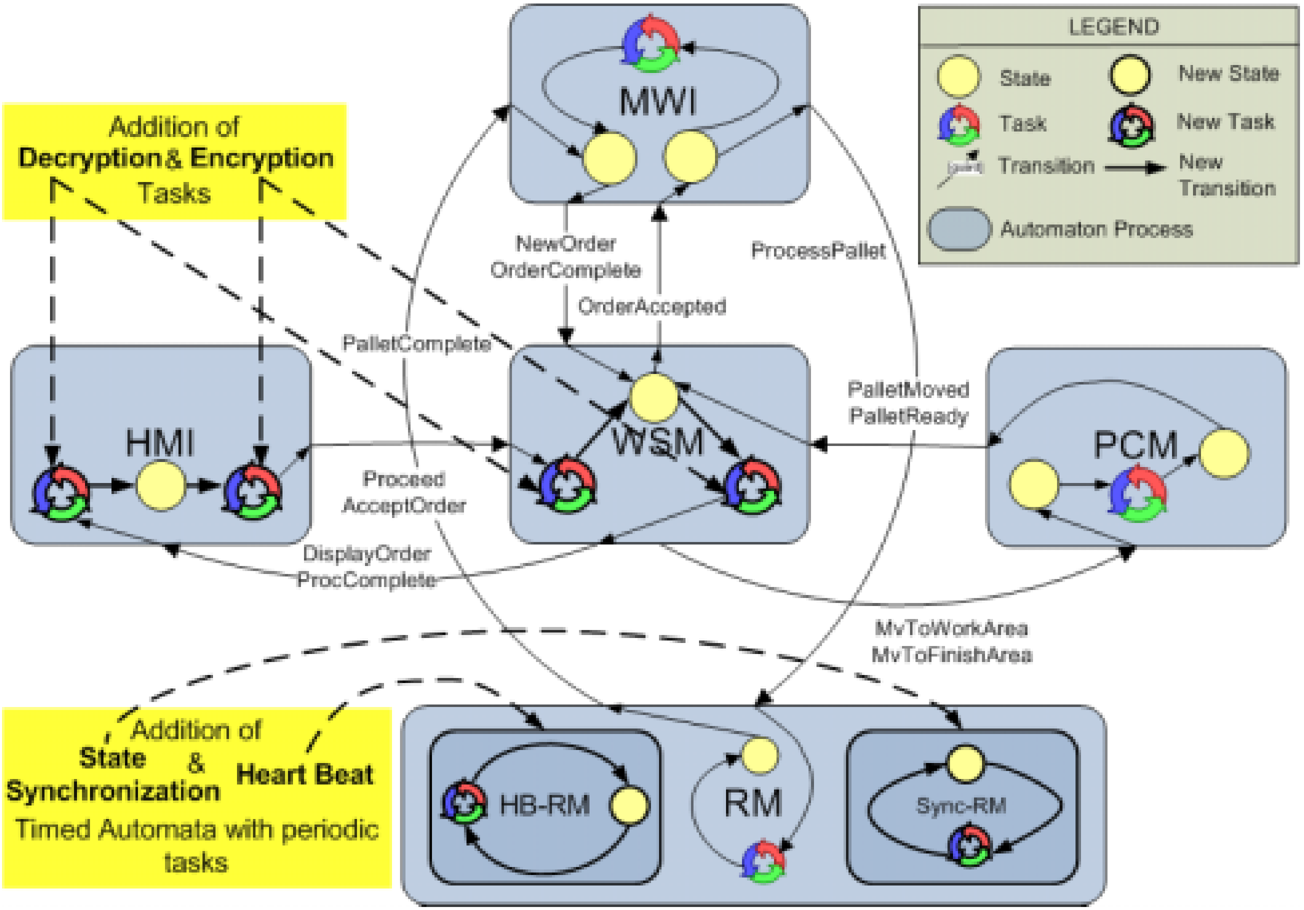


CQML QoS Unification & Tradeoff Analysis



- Leverages QoS and behavioral models
- Model interpretation using an EventBus framework
- Each QoS dimension publishes its modeled QoS requirements
- A particular QoS dimension is chosen as the driver e.g., real-time
  - Acts as subscriber of events generated by other dimensions
  - Driver interpreter integrates other QoS dimensions making tradeoffs on the way
- Pluggable back end analysis and generative tools

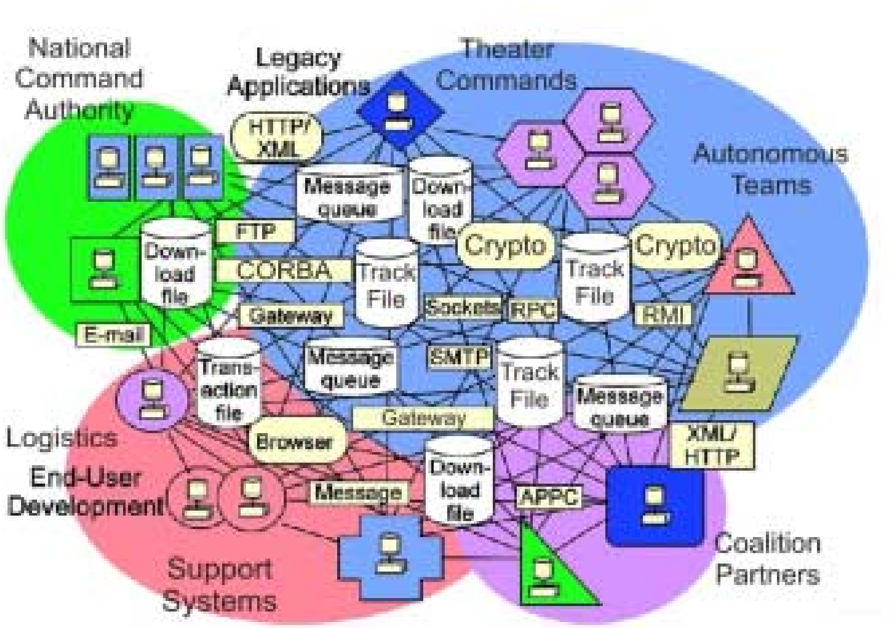
Example: QoS Tradeoff Analysis using TIMES tool



- CQML EventBus architecture weaves in FT and Security concerns into RT concerns
- Feeds to backend schedulability analysis tools e.g., Times

Part 6
Configuring Middleware for DRE Systems

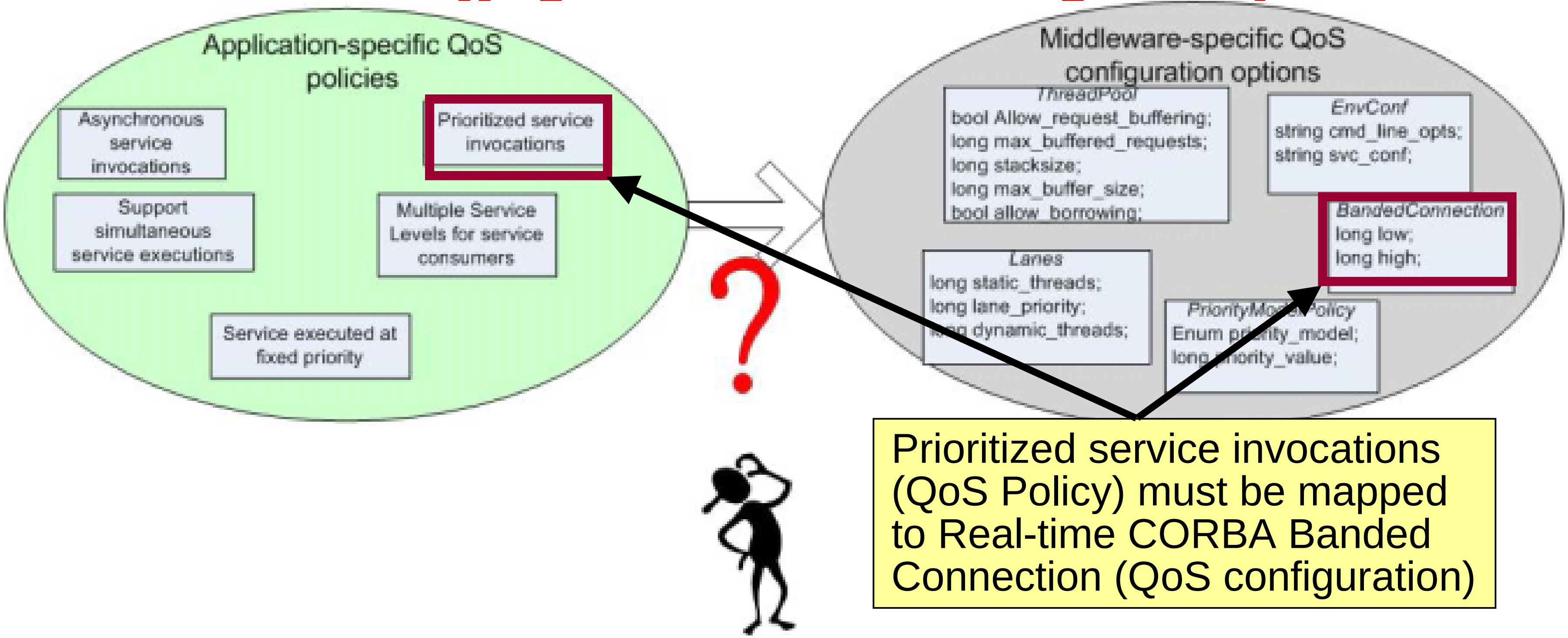
Middleware Configuration Options



Domain-specific Services
Common Services
Distribution Middleware
Infrastructure Middleware

Operating System & Network Stack

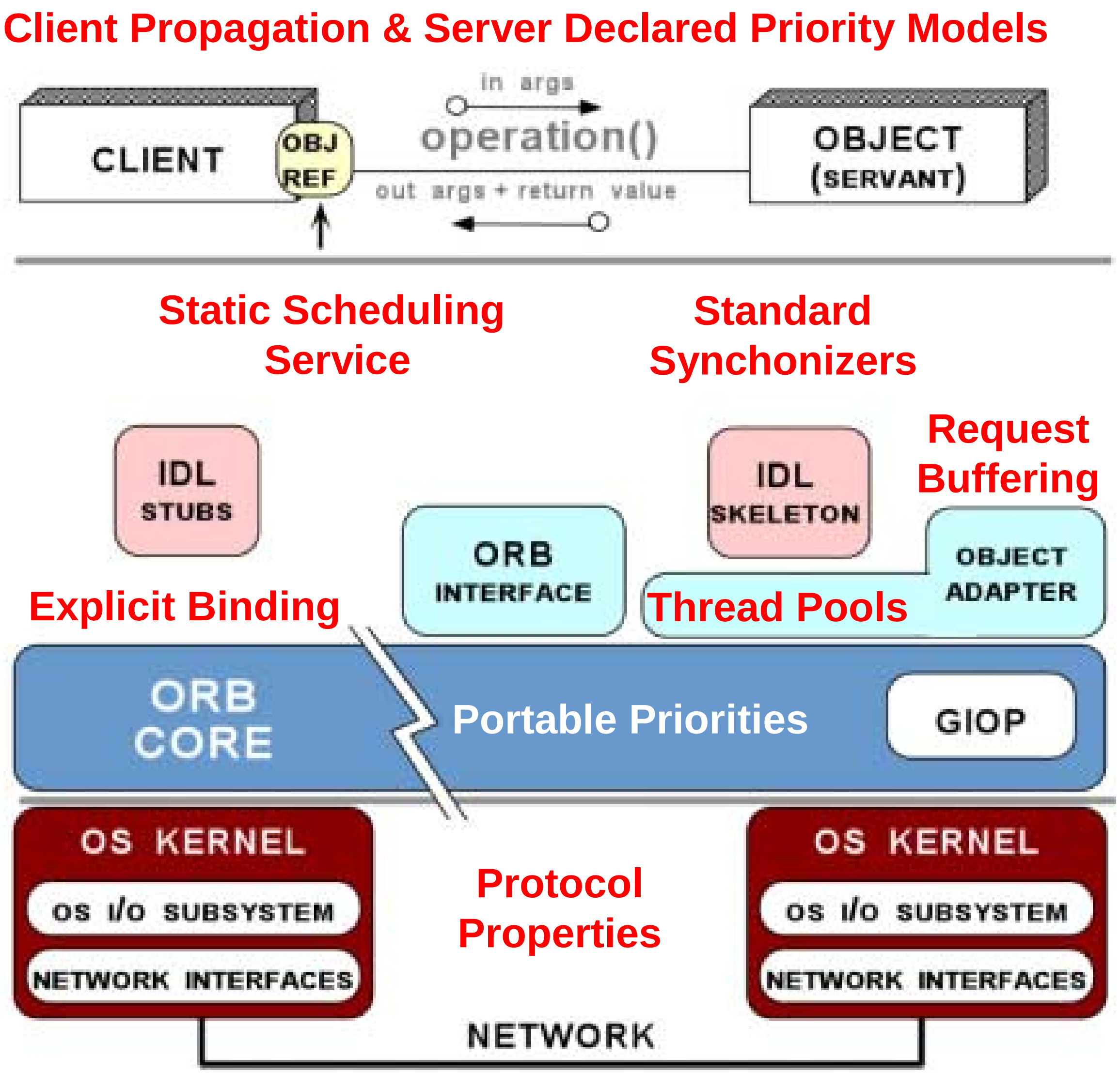
Translating QoS Policies to QoS Options



- Large gap between application QoS policies & middleware QoS configuration options
  - Bridging this gap is necessary to realize the desired QoS policies
- The mapping between application-specific QoS policies & middleware-specific QoS configuration options is non-trivial, particular for large systems

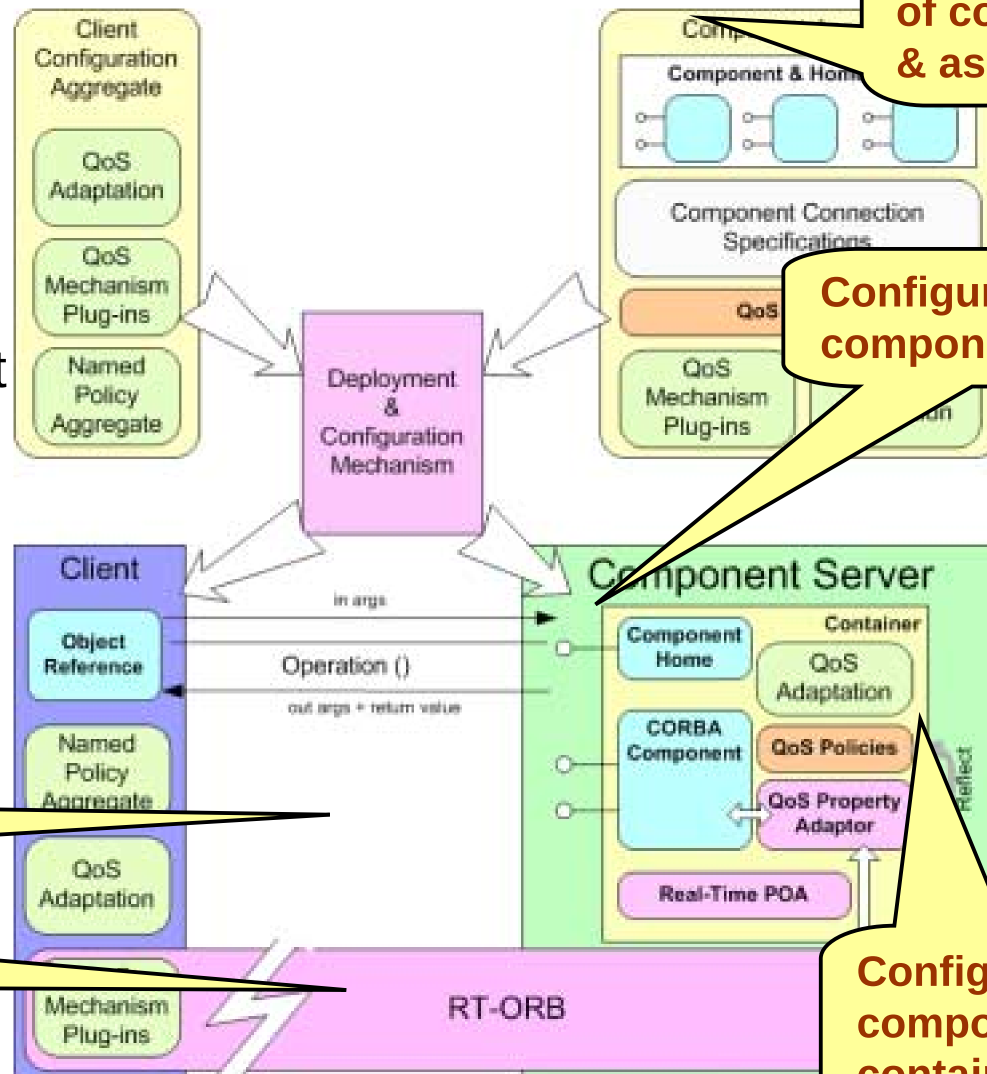
Challenge 1: Mapping QoS Policies to QoS Options

- Conventional mapping approach requires deep understanding of the middleware configuration space
  - e.g., multiple types/levels of QoS policies require configuring appropriate number of thread pools, threadpool lanes (server) & banded connections (client)



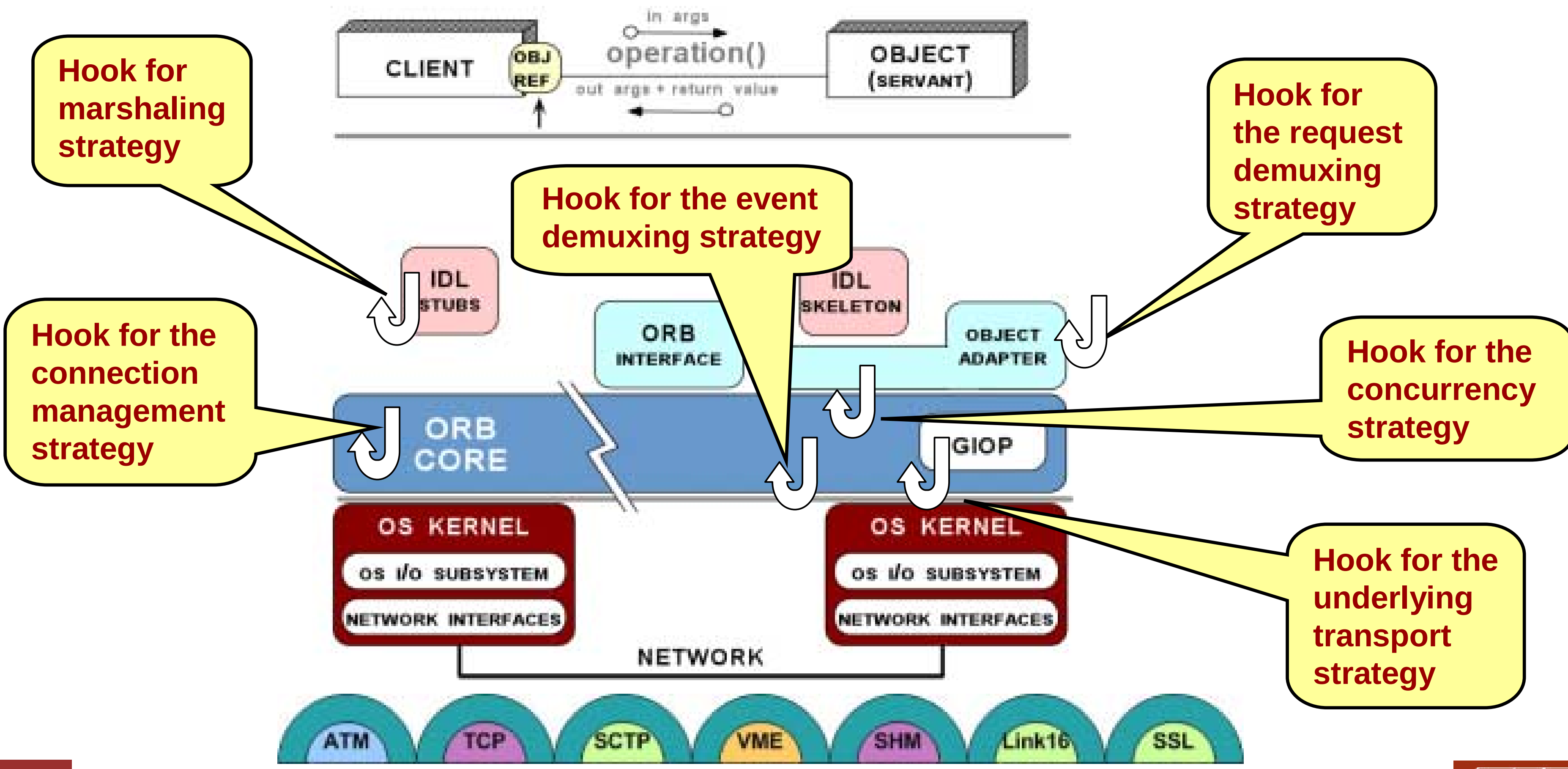
The Multilayer Configuration Problem

- Component-based software must be configurable at many levels
  - *e.g.*, application components & containers, component servers, & middleware services & infrastructure

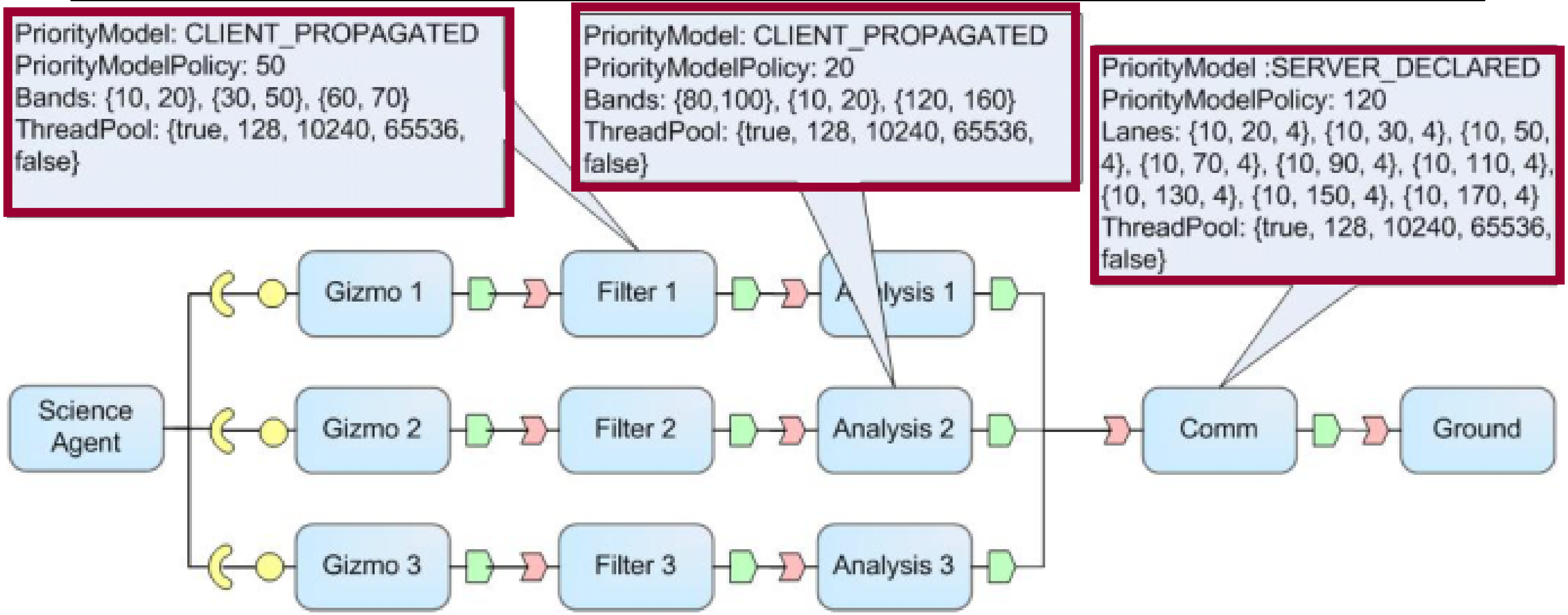


Example: The M/W Bus Configuration

Component middleware is characterized by a large *configuration space* that maps known variations in the application requirements space to known variations in the middleware solution space



Challenge 2: Choosing QoS Option Values

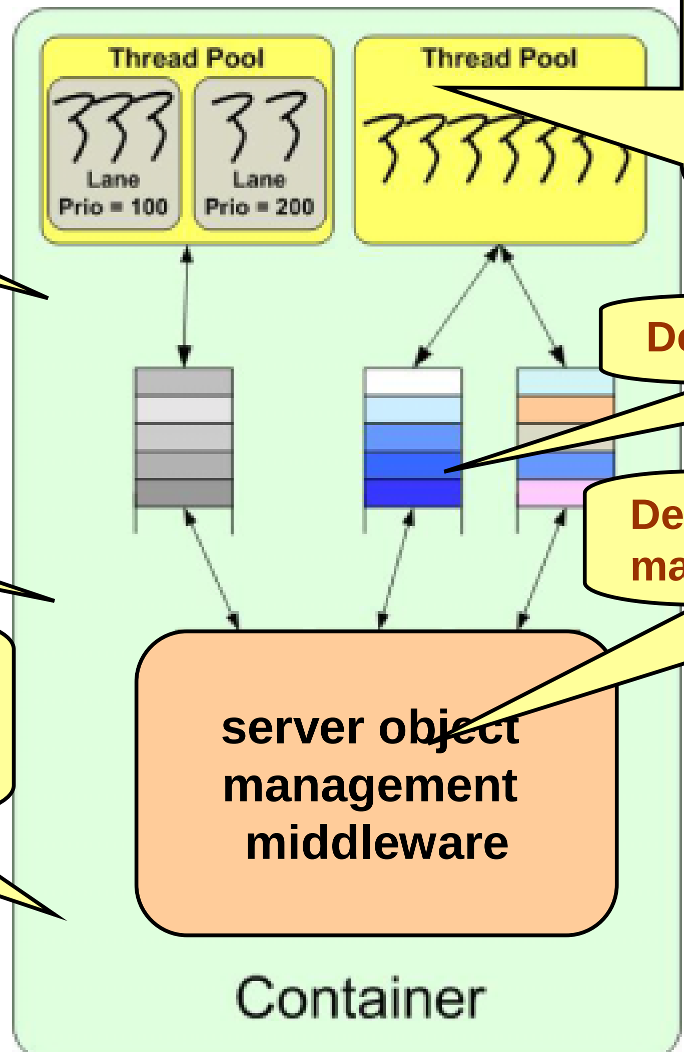


- Individually configuring component QoS options is tedious & error-prone
  - e.g., ~10 distinct QoS options per component & ~140 total QoS options for entire NASA MMS mission prototype
- Manually choosing valid values for QoS options does not scale as size & complexity of applications increase

Example: Configuring Container Policies

Determine various middleware policies for server objects e.g., security, lifetime, replication

Determine thread pool sizes; how are they shared; number of lanes and their priorities; if borrowing is enabled



Determine end-to-end priority propagation model to use

Determine right buffer sizes

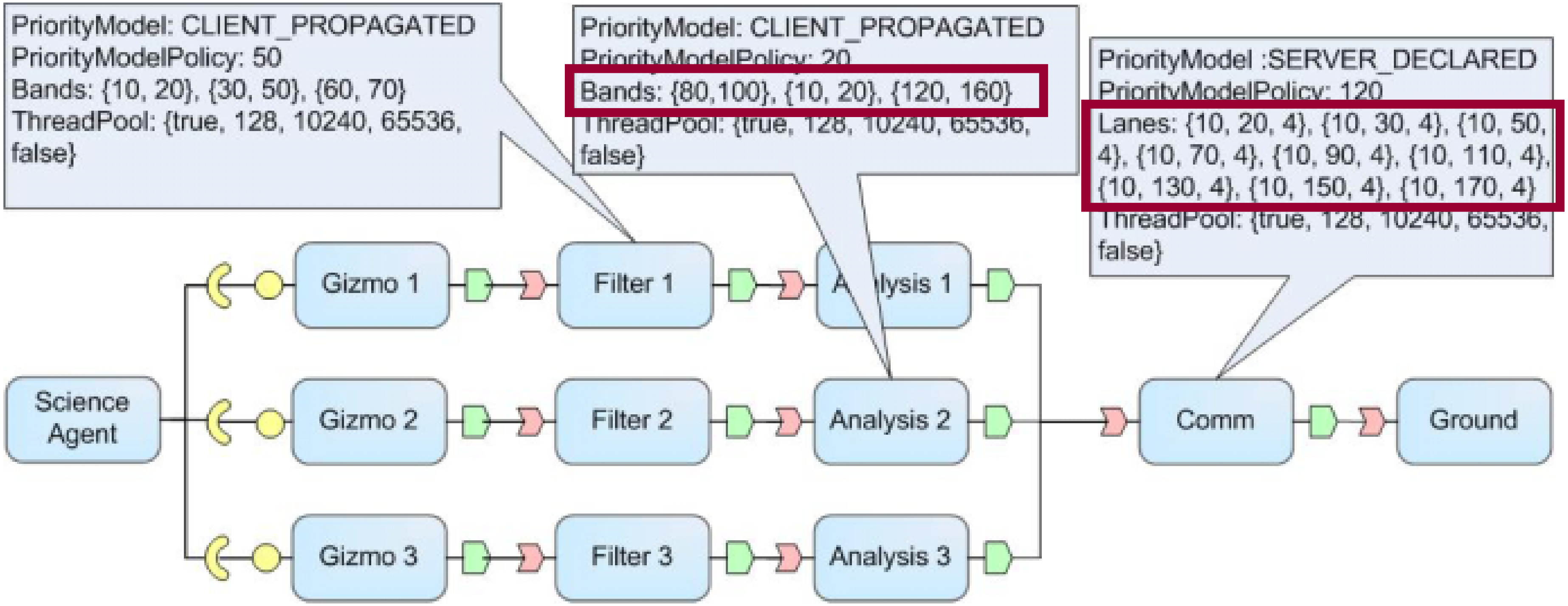
Determine the server object management policies

Ensure semantic compatibility among chosen configurations

- This "glue code" is traditionally handcrafted

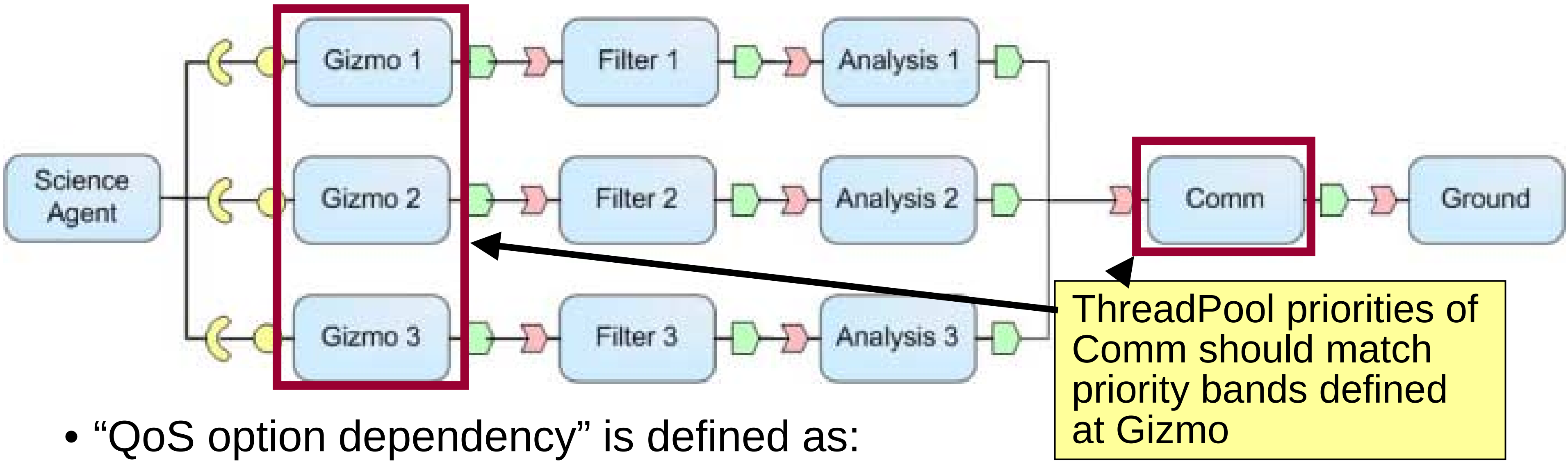
- Existing techniques for metadata configurations rely on *ad hoc* manual configurations e.g., CORBA server-side programming

Challenge 3: Validating QoS Options



- Each QoS option value chosen should be validated
  - e.g., Filter priority model is CLIENT_PROPAGATED, whereas Comm priority model is SERVER_DECLARED
- Each system reconfiguration (at design time) should be validated
  - e.g., reconfiguration of *bands* of Analysis should be validated such that the modified value corresponds to (some) *lane* priority of the Comm

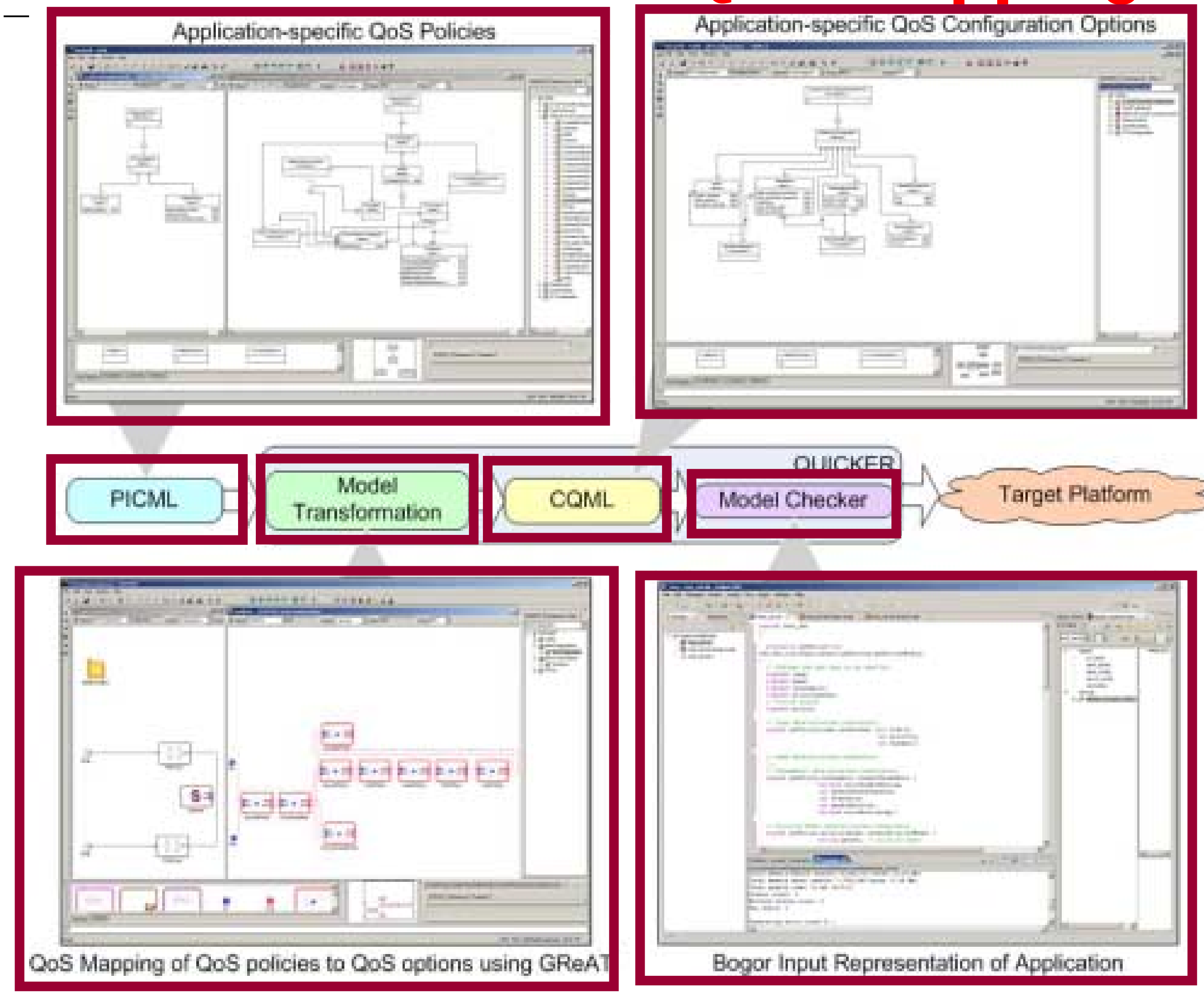
Challenge 4: Resolving QoS Option Dependencies



- "QoS option dependency" is defined as:
  - Dependency between QoS options of different components
- Manually tracking dependencies is hard – or in some cases infeasible
  - Dependent components may belong to more than one assembly
  - Dependency may span beyond immediate neighbors
    - e.g., dependency between Gizmo & Comm components
- Empirically validating configuration changes by hand is tedious, error-prone, & slows down development & QA process considerably
  - Several iterations before desired QoS is achieved (if at all)

Solution Approach: Model-Driven QoS Mapping

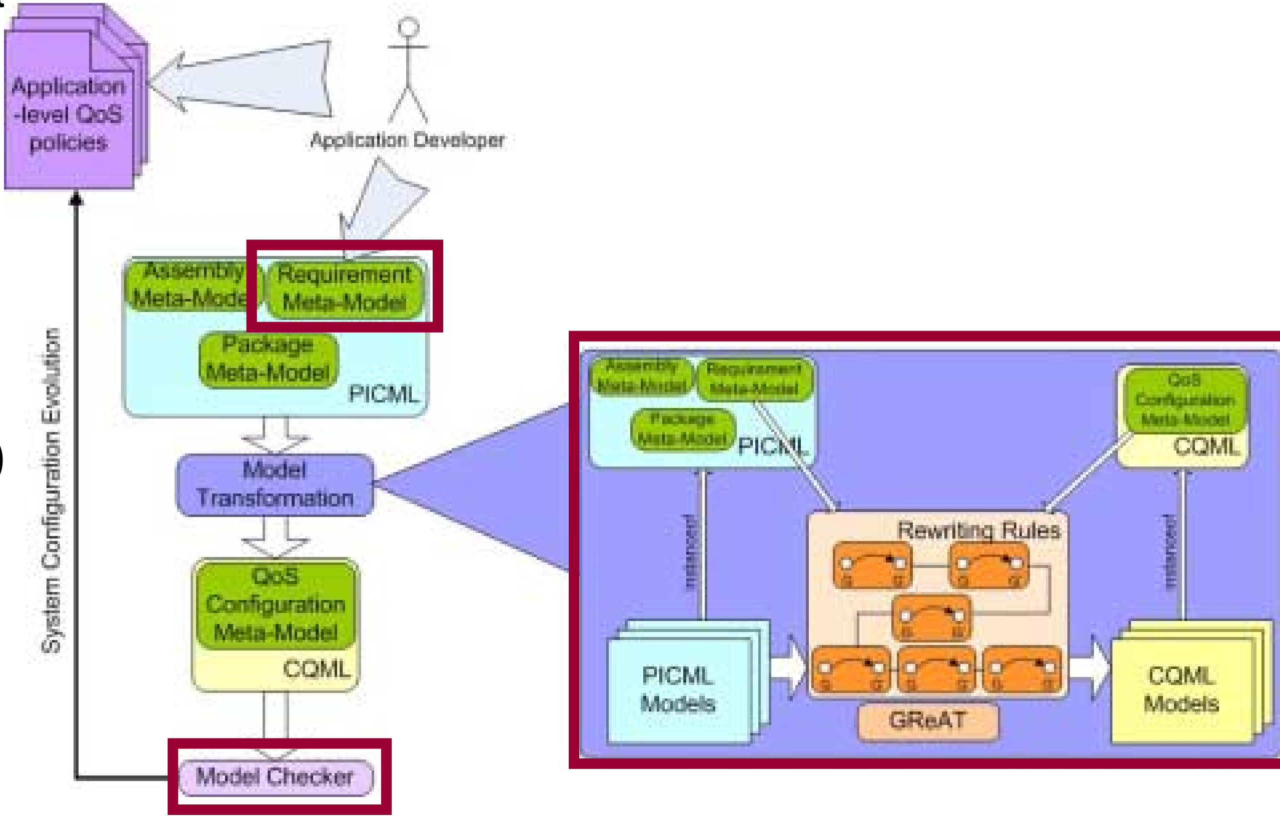
- QUality of service pICKER (QUICKER)
  - Model-driven engineering (MDE) tools model application QoS policies
  - Provides automatic mapping of QoS policies to QoS configuration options
  - Validates the generated QoS options



- Automated QoS mapping & validation tools can be used iteratively throughout the development process

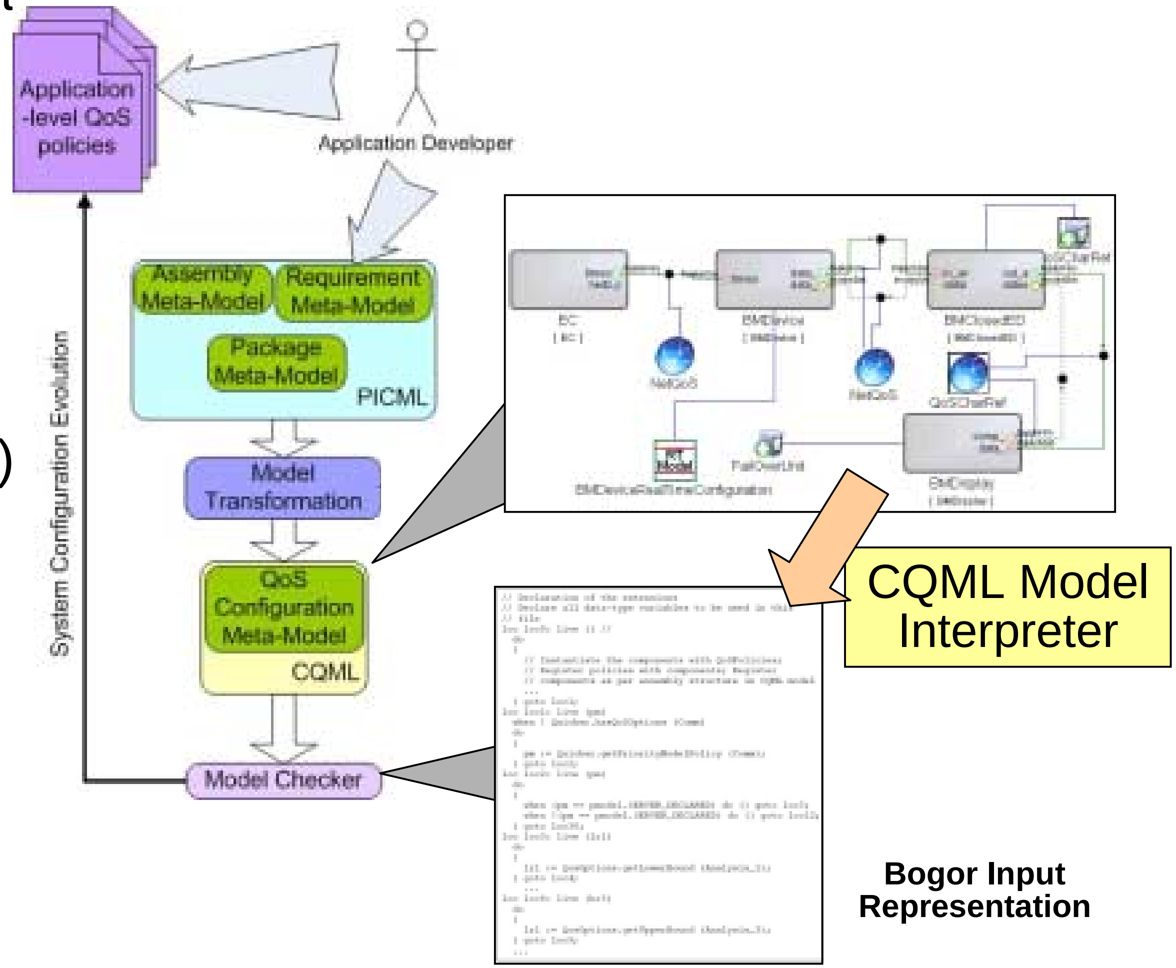
QUICKER Enabling MDE Technologies

- Enhanced Platform Independent Component Modeling Language (PICML), a DSML for modeling component-based applications
- QoS mapping uses Graph Rewriting & Transformation (GReAT) model transformation tool
- Customized Bogor model-checker used to define new types & primitives to validate QoS options



QUICKER Enabling MDE Technologies

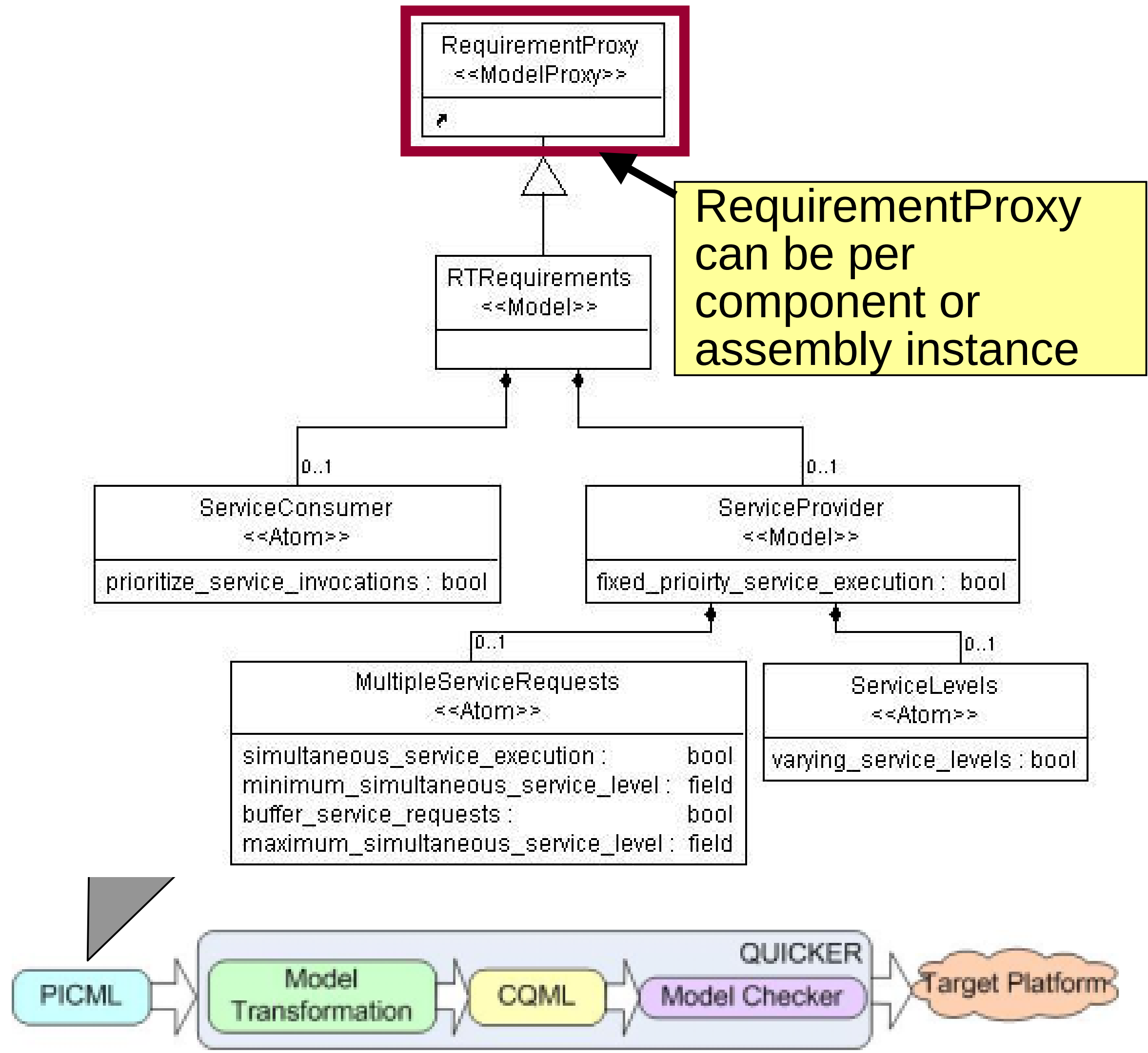
- Enhanced Platform Independent Component Modeling Language (PICML), a DSML for modeling component-based applications
- QoS mapping uses Graph Rewriting & Transformation (GReAT) model transformation tool
- Customized Bogor model-checker used to define new types & primitives to validate QoS options



- CQML Model interpreter generates Bogor Input Representation (BIR) of DRE system from its CQML model

QUICKER: Transformation of QoS policies(1/2)

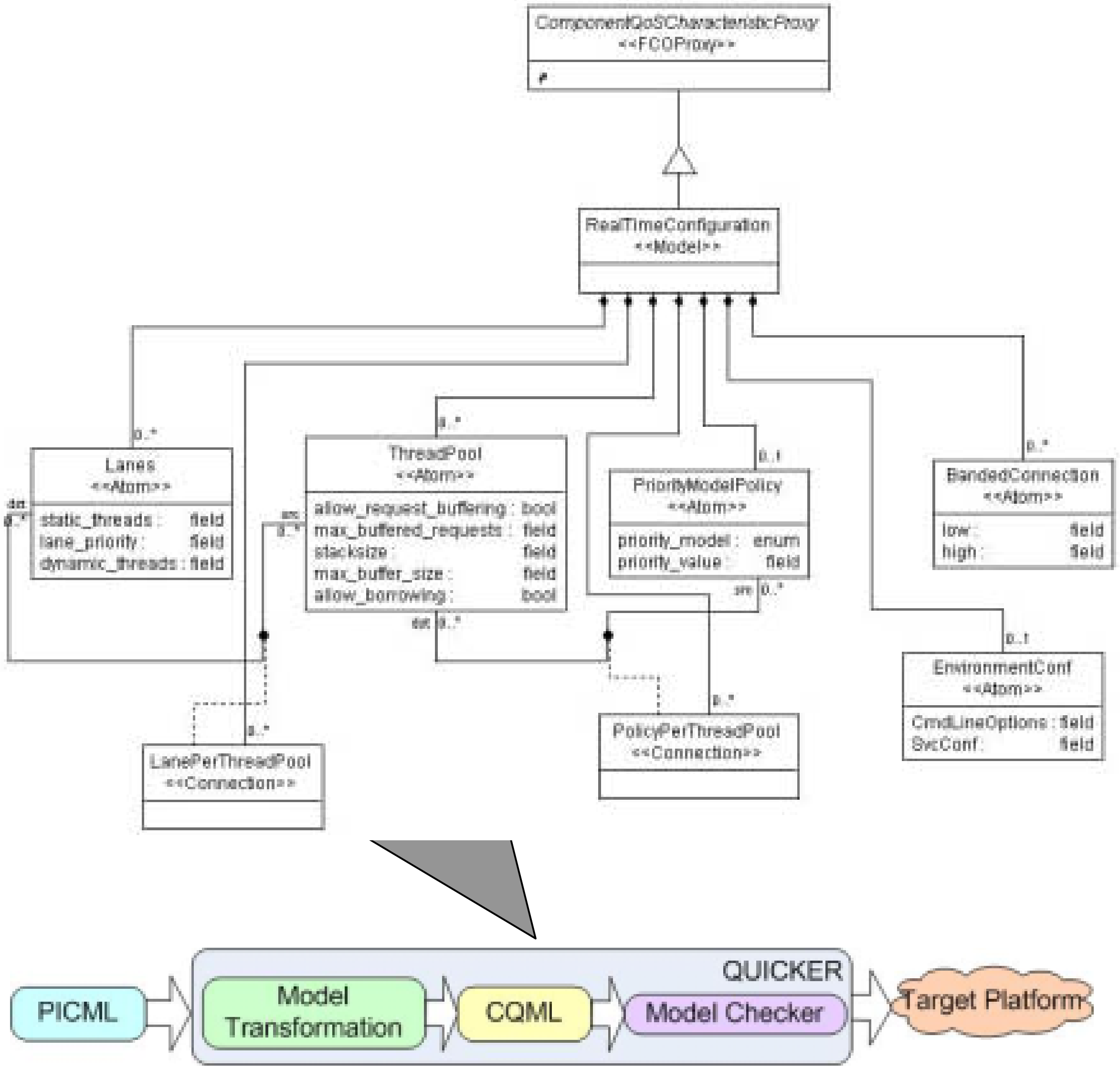
1. Platform-Independent Modeling Language (PICML) represents application QoS policies
   - PICML captures policies in a platform-independent manner
   - Representation at multiple levels
     - e.g., component- or assembly-level



69

QUICKER: Transformation of QoS policies(1/2)

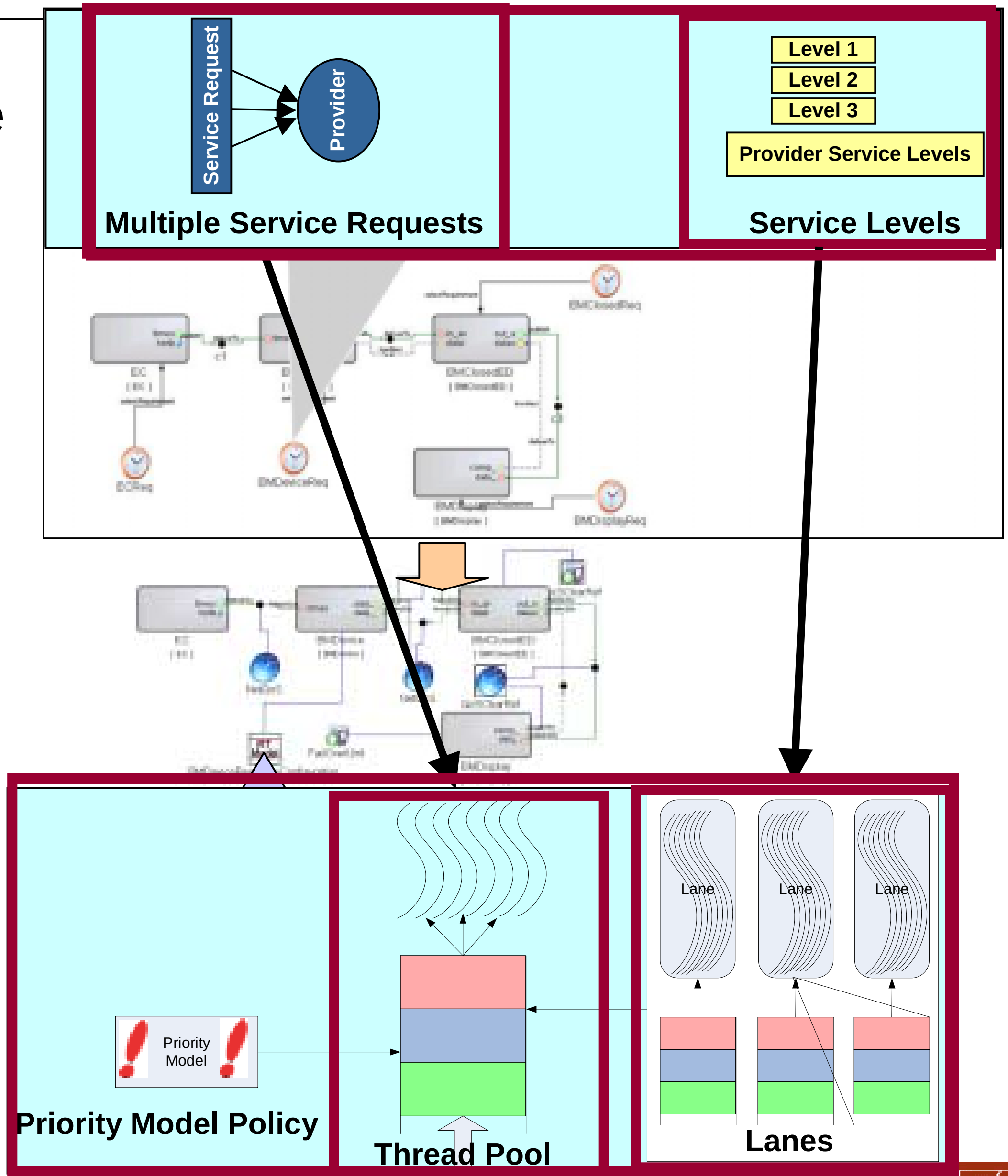
1. Platform-Independent Modeling Language (PICML) represents application QoS policies
   - PICML captures policies in a platform-independent manner
   - Representation at multiple levels
     - e.g., component- or assembly-level
2. Component QoS Modeling Language (CQML) represents QoS options
   - CQML captures QoS configuration options in a platform-specific manner



70

QUICKER: Transformations of QoS policies(2/2)

3. Translation of application QoS policies into middleware QoS options
   - Semantic translation rules specified in terms of input (PICML) & output (CQML) type graph
     - e.g., rules that translate multiple application service requests & service level policies to corresponding middle-ware QoS options
   - QUICKER transformation engine maps QoS policies (in PICML) to QoS configuration options (in CQML)



QUICKER: Validation of QoS Options (1/2)

1. Representation of middleware QoS options in Bogor model-checker
    - BIR extensions allow representing domain-level concepts in a system model
    - QUICKER defines new BIR extensions for QoS options
        - Allows representing QoS options & domain entities directly in a Bogor input model
            - e.g., CCM components, Real-time CORBA lanes/bands are first-class Bogor data types
    - Reduces size of system model by avoiding multiple low-level variables to represent domain concepts & QoS options

```
extension QoSOptions for
edu.ksu.cis.bogor.module.QoSOptions.QoSOptionsModule
{
  // Defines the new type to be used for
  typedef lane;
  typedef band;
  typedef threadpool;
  typedef prioritymodel;
  typedef policy;
  // Lane constructor.
  expdef QoSOptions.lane createLane (
   int static, int priority, int dynamic);
  // ThreadPool constructor.
  expdef QoSOptions.threadpool
  createThreadPool (boolean allowreqbuffering,
   int maxbufferedrequests, int stacksize, int
   maxbuffersize, boolean allowborrowing);
  // Set the band(s) for QoS policy.
  actiondef registerBands (QoSOptions.policy
   policy, QoSOptions.band ...);
  // Set the lane(s) for QoS policy.
  actiondef registerLanes (QoSOptions.policy
   policy, QoSOptions.lane ...);
  ...
}
extension Quicker for
edu.ksu.cis.bogor.module.Quicker
{
  // Defines the new type.
  typedef Component;
  // Component Constructor.
  expdef Quicker.Component
  createComponent (string component);
  // Set the QoS policy for the component.
  actiondef registerQoSOptions (Quicker.Component
    component,QoSOptions.policy policy);
  // Make connections between components.
  actiondef connectComponents (Quicker.Component
    server,Quicker.Component client);
  ...
}
```

QUICKER: Validation of QoS Options (2/2)

2. Representation of properties (that a system should satisfy) in Bogor
   - BIR primitives define language constructs to access & manipulate domain-level data types, e.g.:
     - Used to define rules that validate QoS options & check if property is satisfied
3. Automatic generation of BIR of DRE system from CQML models

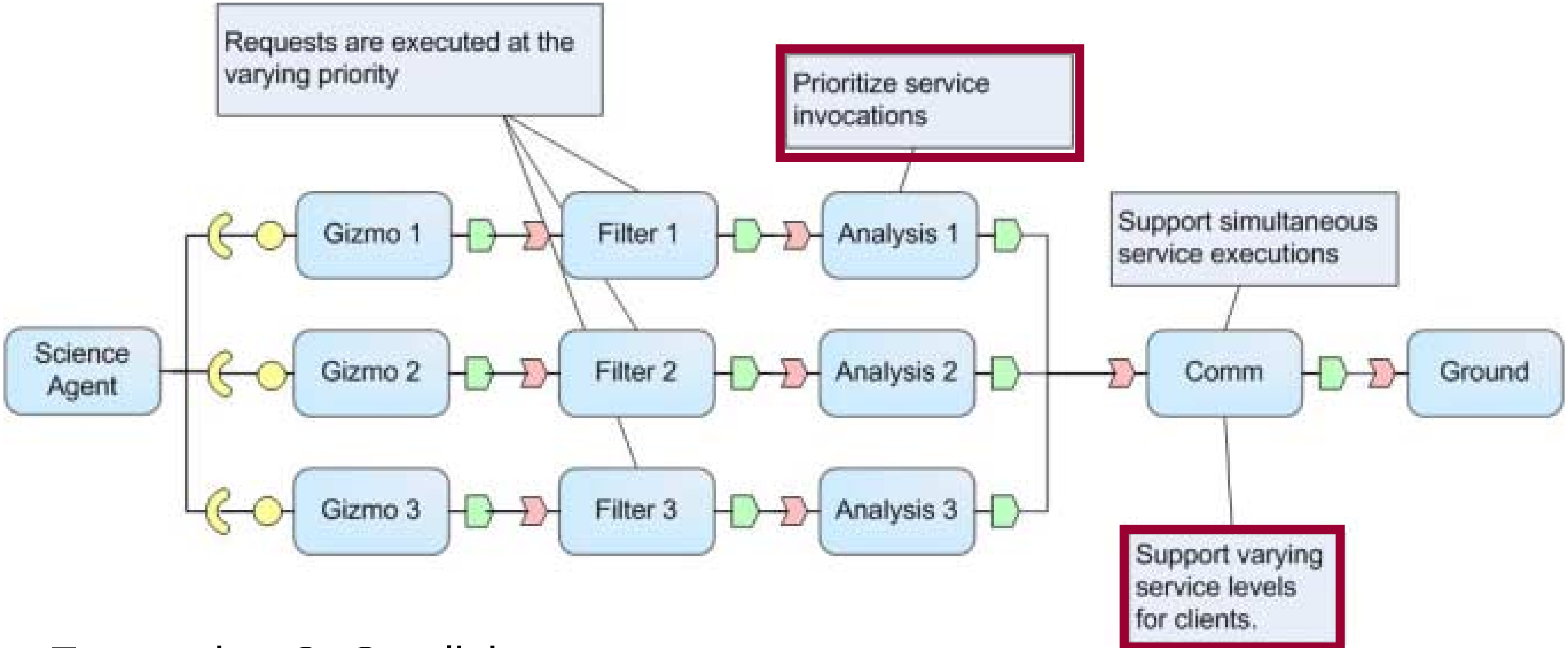
```
// Declaration of the extensions
// Declare all data-type variables to be used in this
// file
loc loc0: live {} //
  do
  {
    // Instantiate the components with QoSPolicies;
    // Register policies with components; Register
    // components as per assembly structure in CQML model
    ...
  } goto loc1;
loc loc1: live {pm}
  when ! Quicker.hasQoSOptions (Comm)
  do
  {
    pm := Quicker.getPriorityModelPolicy (Comm);
  } goto loc2;
loc loc2: live {pm}
  do
  {
    when (pm == pmodel.SERVER_DECLARED) do {} goto loc3;
    when !(pm == pmodel.SERVER_DECLARED) do {} goto loc12
  } goto loc30;
loc loc3: live {lr1}
  do
  {
    lr1 := QosOptions.getLowerBound (Analysis_1);
  } goto loc4;
  ...
loc loc8: live {hr3}
  do
  {
    lr1 := QosOptions.getUpperBound (Analysis_3);
  } goto loc9;
  ...
```

Rule determines if ThreadPool priorities at Comm match priority bands at Analysis

Model interpreters auto-generate Bogor Input Representation of a system from its CQML model

73

Soln: Translating Policies to Options (1/2)

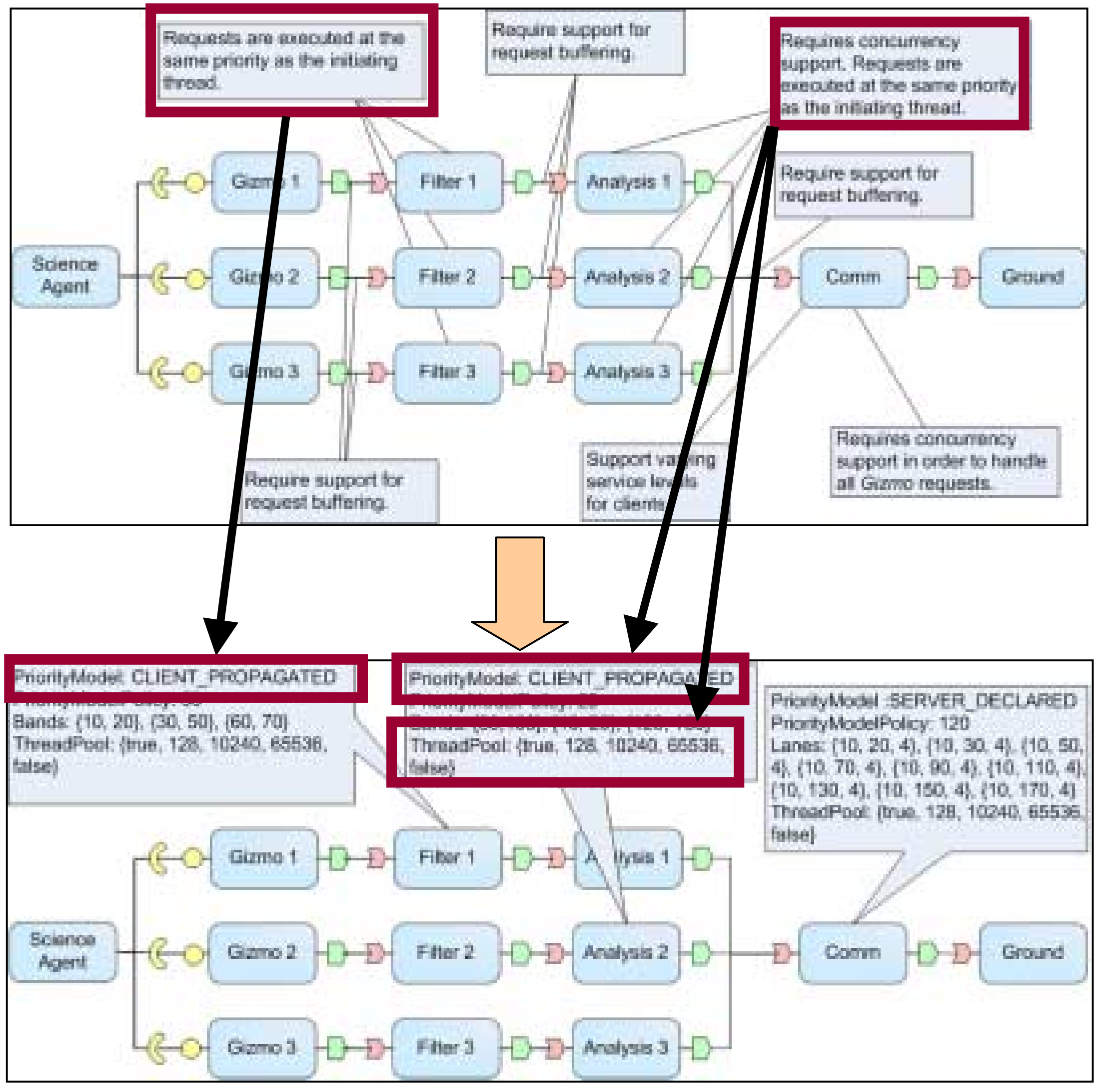


- Expressing QoS policies
  - PICML modes application-level QoS policies at high-level of abstraction
    - e.g., multiple service levels support for Comm component, service execution at varying priority for Analysis component
  - Reduces modeling effort
    - e.g., ~25 QoS policy elements for MMS mission vs. ~140 QoS options

74

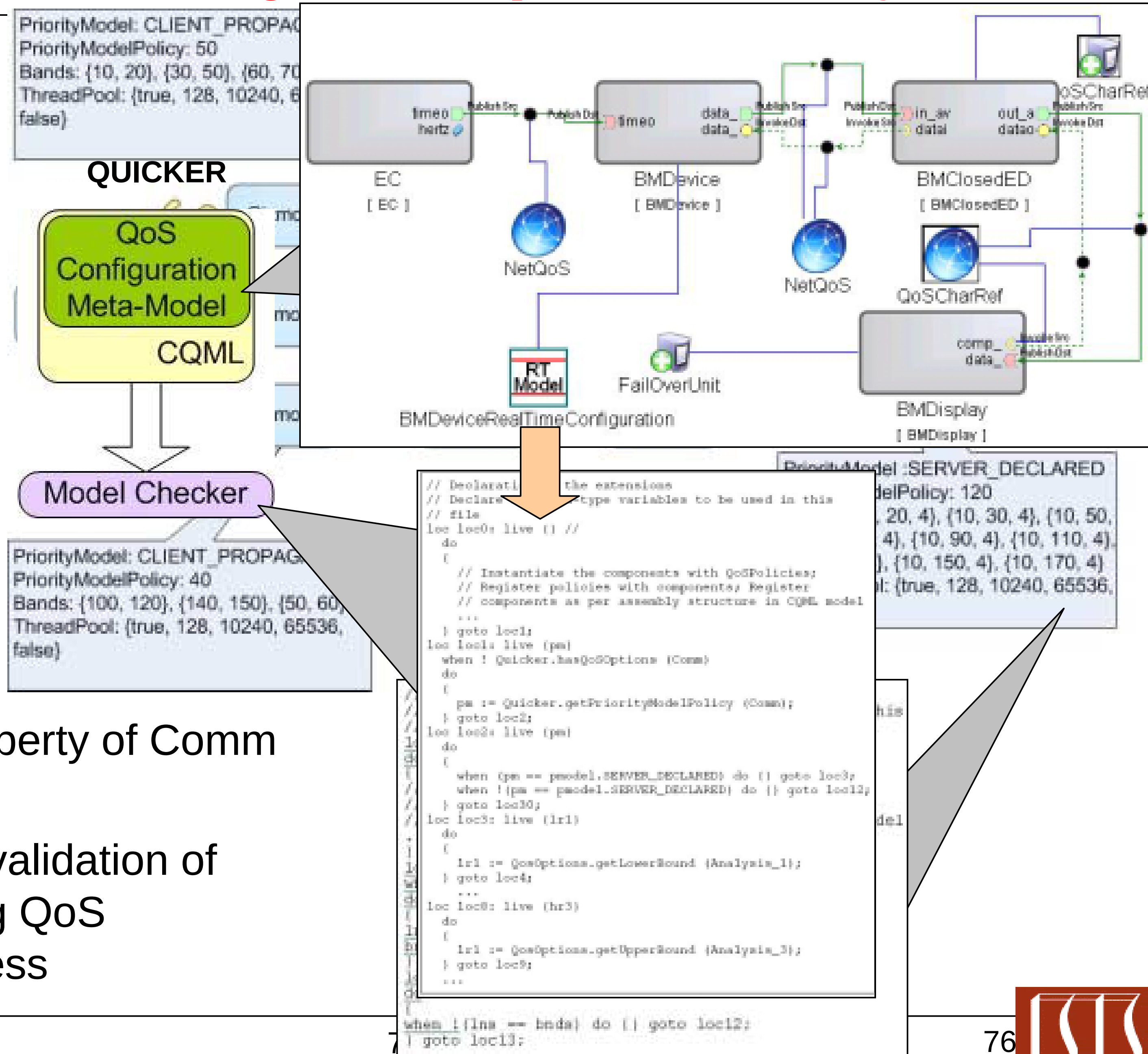
Soln: Translating Policies to Options (2/2)

- Mapping QoS policies to QoS options
  - GReAT model transformations automate the tedious & error-prone translation process
- Transformations generate QoS configuration options as CQML models
  - Allow further transformation by other tools
    - e.g., code optimizers & generators
  - Simplifies application development & enhances traceability



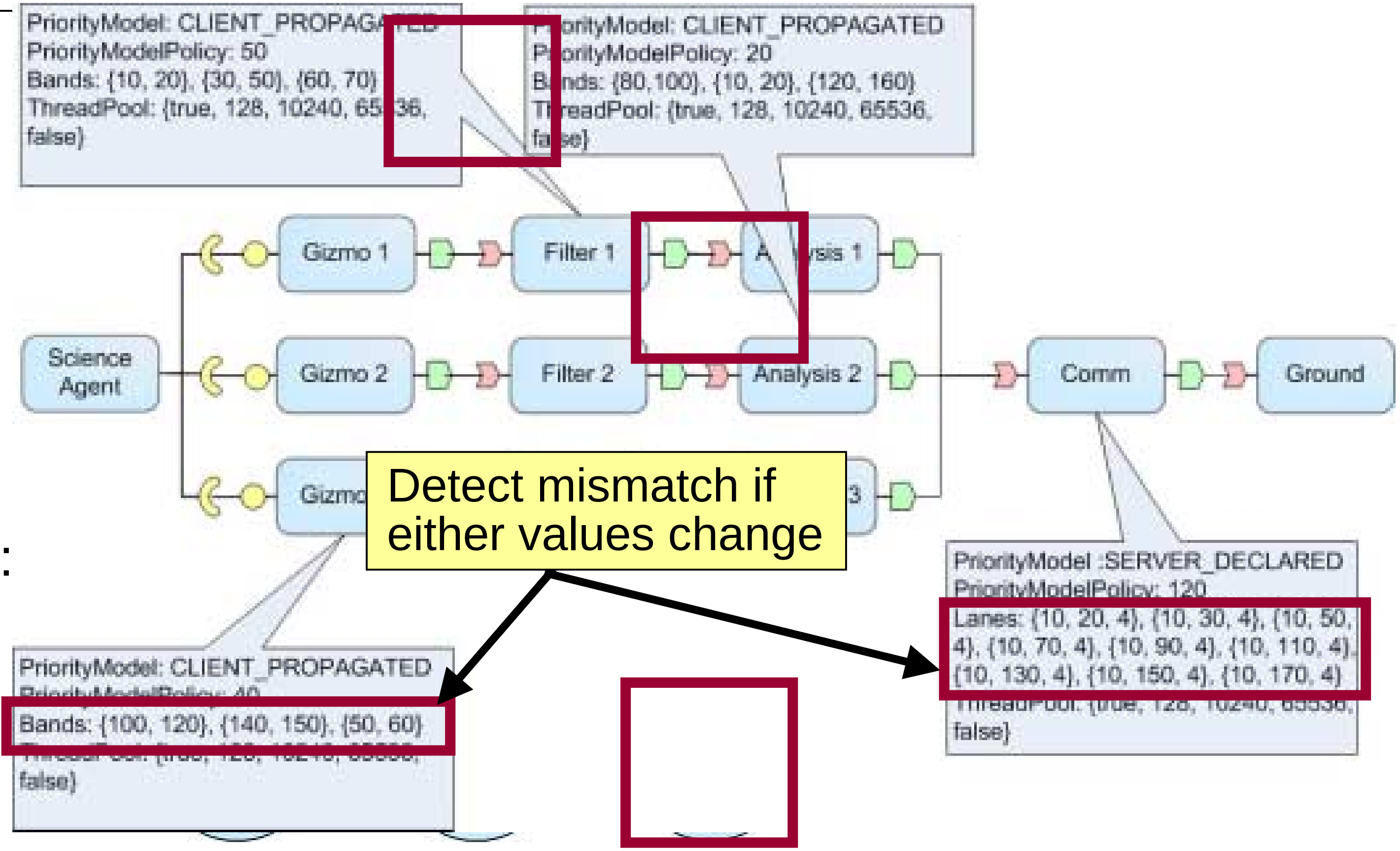
Soln: Ensuring QoS Option Validity

- CQML model interpreter generates BIR specification from CQML models
- BIR primitives used to check whether a given set of QoS options satisfies a system property
  - e.g., fixed priority service execution, a property of Comm component
- Supports iterative validation of QoS options during QoS configuration process



Soln: Resolving QoS Option Dependencies

- Dependency structure maintained in Bogor used to track dependencies between QoS options of components, e.g.:
  - Analysis & Comm are connected
  - Gizmo & Comm are dependent



Dependency Structure of MMS Mission Components

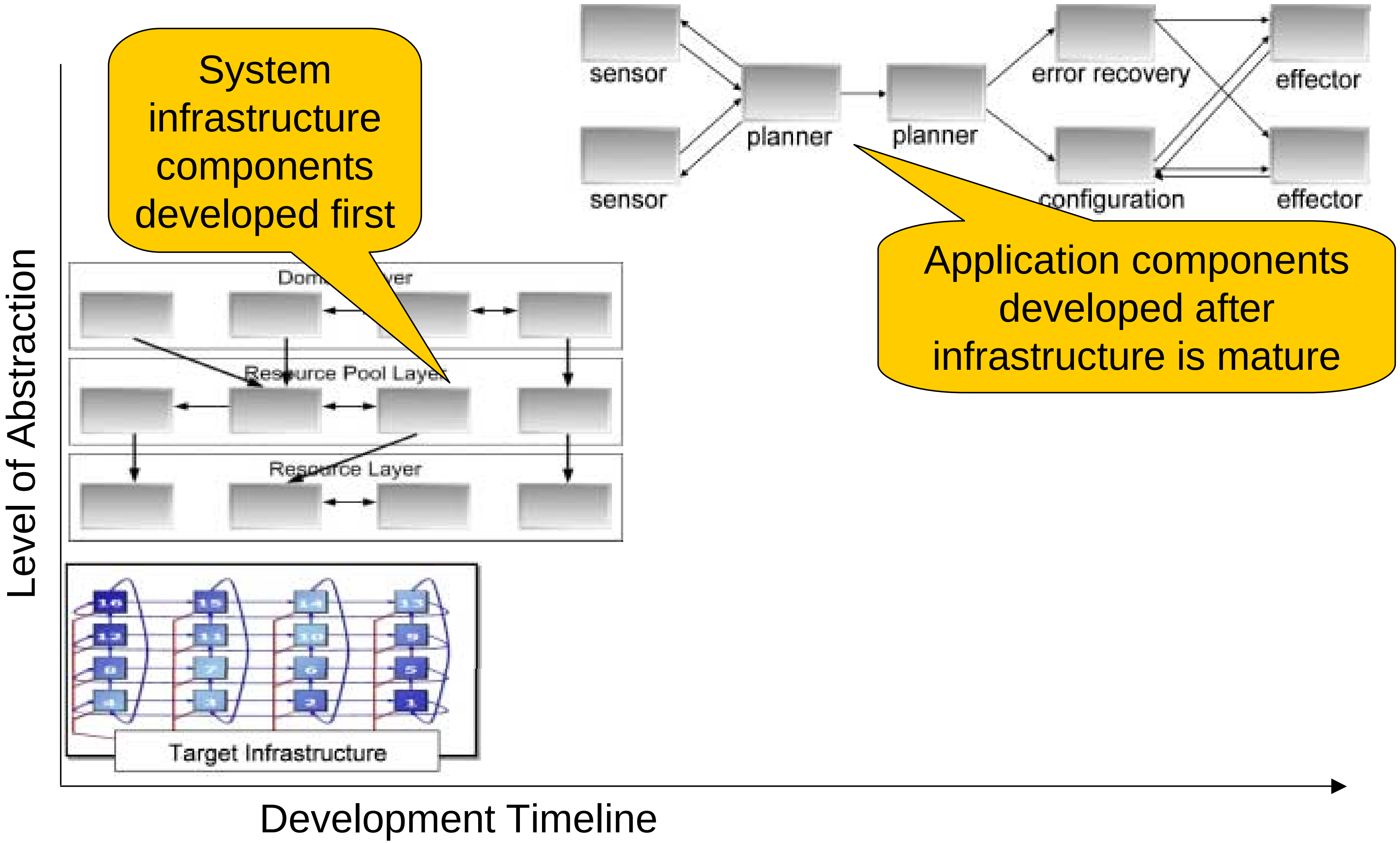
- Change(s) in QoS options of dependent component(s) triggers detection of potential mismatches
  - e.g., dependency between Gizmo invocation priority & Comm lane priority

77

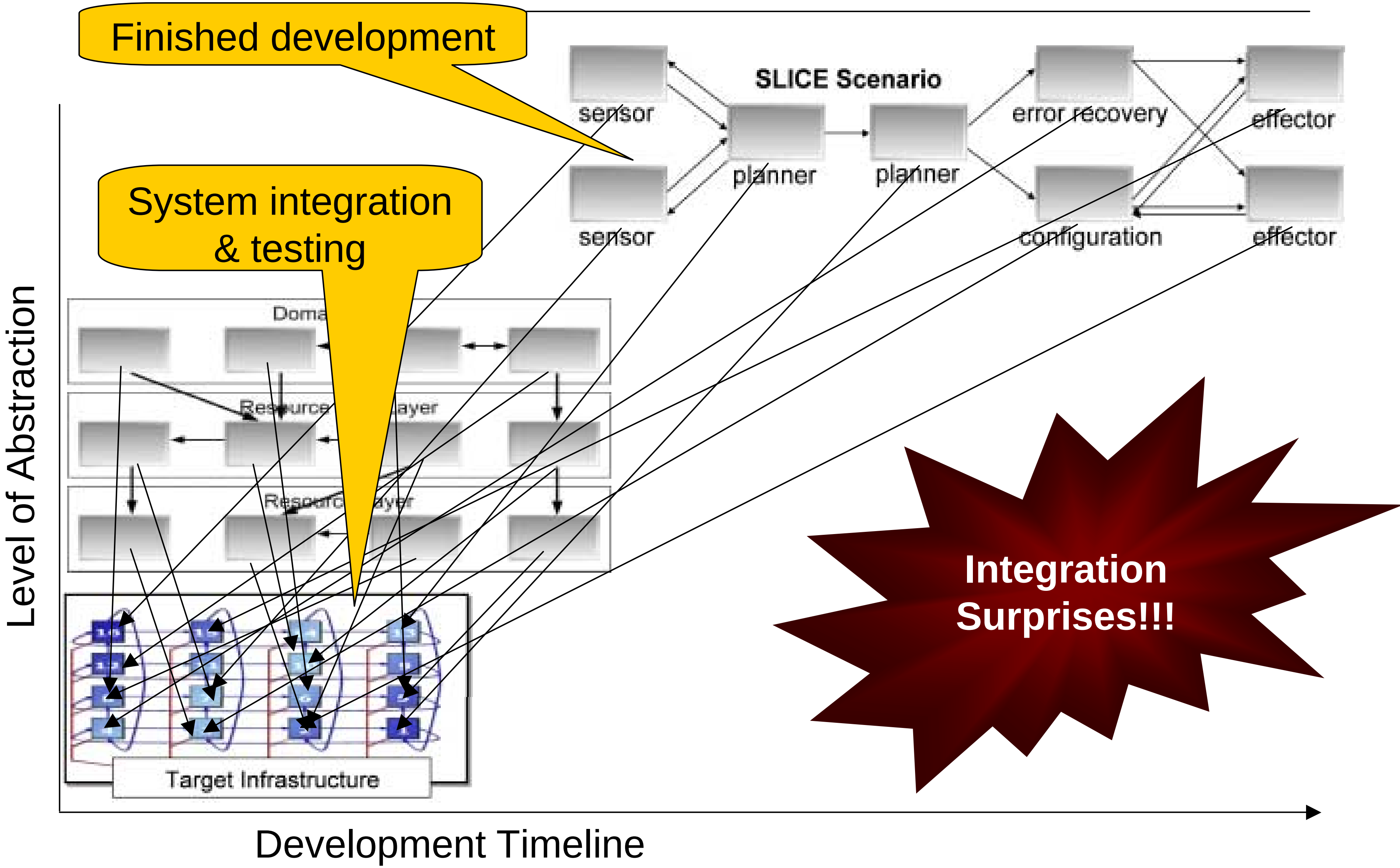
Part 7
Continuous System QoS Validation using "What If" Analysis

MDE for Emulation Techniques

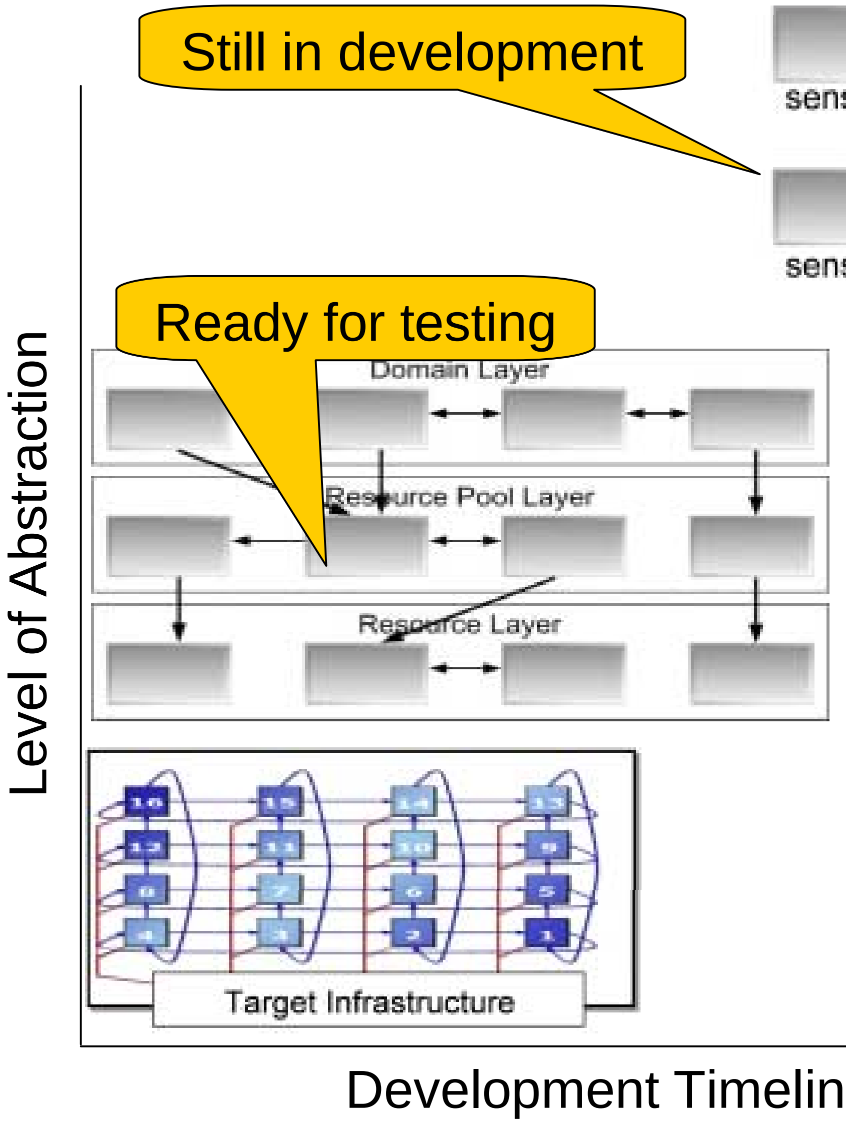
Serialized Phasing in DRE Systems (1/2)

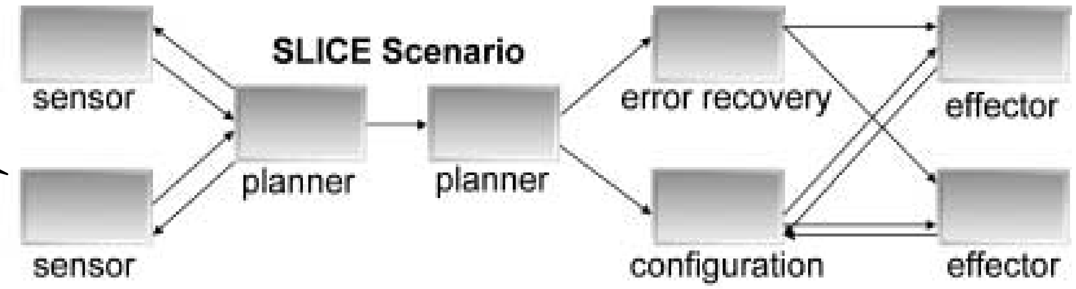


Serialized Phasing in DRE Systems (2/2)



Complexities of Serialized Phasing



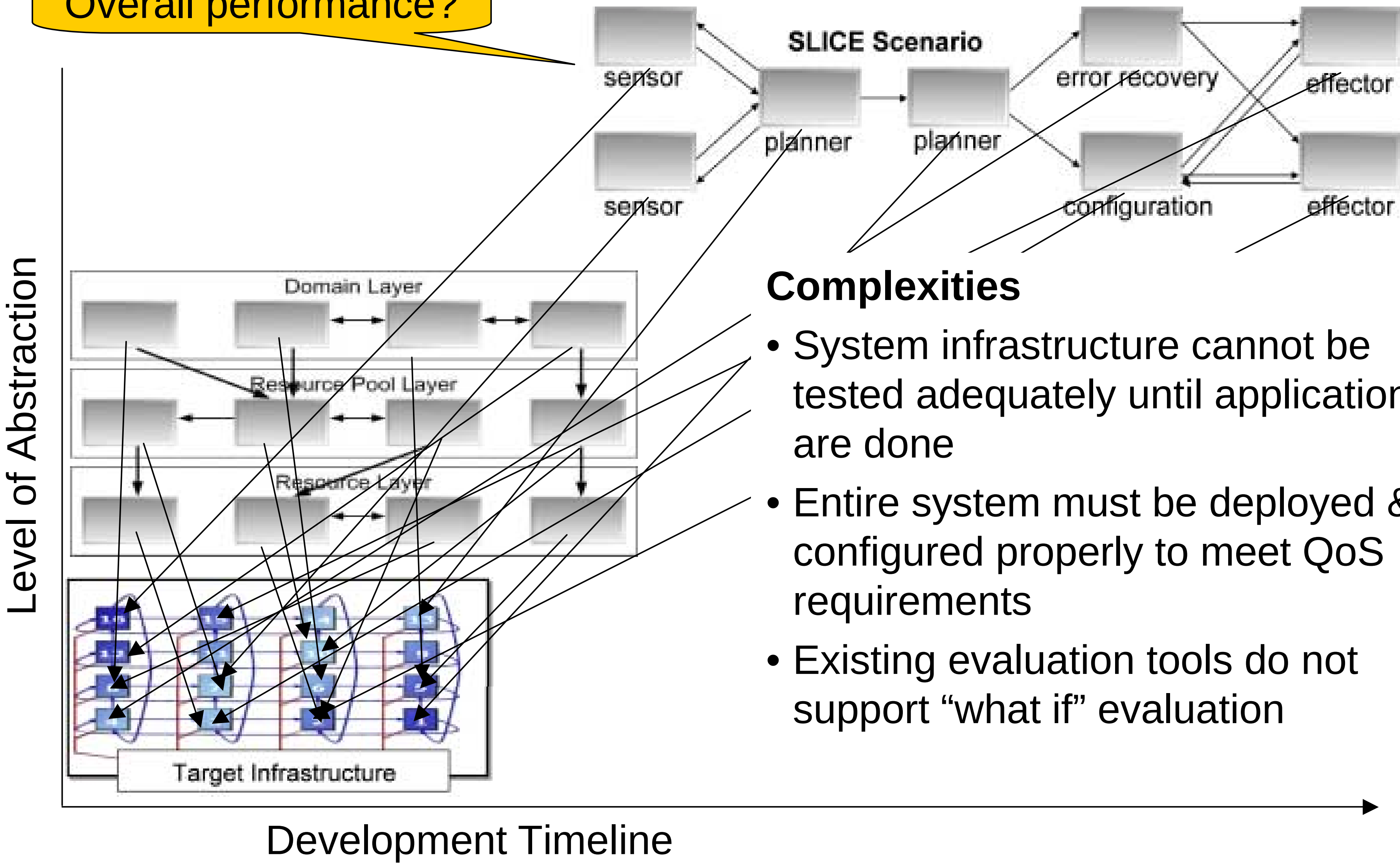


Complexities

- System infrastructure cannot be tested adequately until applications are done

Complexities of Serialized Phasing

Overall performance?

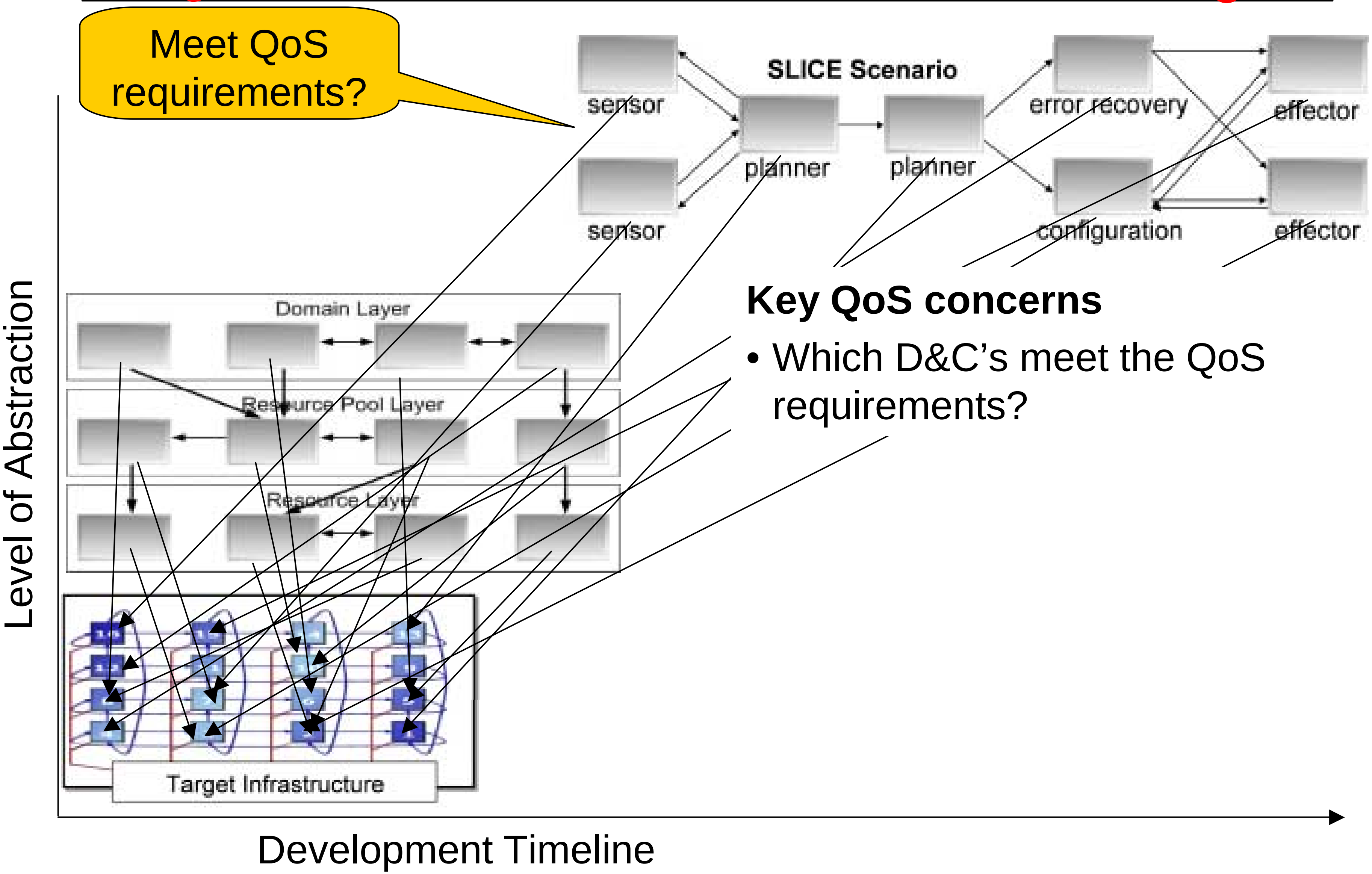


Complexities

- System infrastructure cannot be tested adequately until applications are done
- Entire system must be deployed & configured properly to meet QoS requirements
- Existing evaluation tools do not support "what if" evaluation

It is hard to address these concerns in processes that use serialized phasing

QoS Concerns with Serialized Phasing

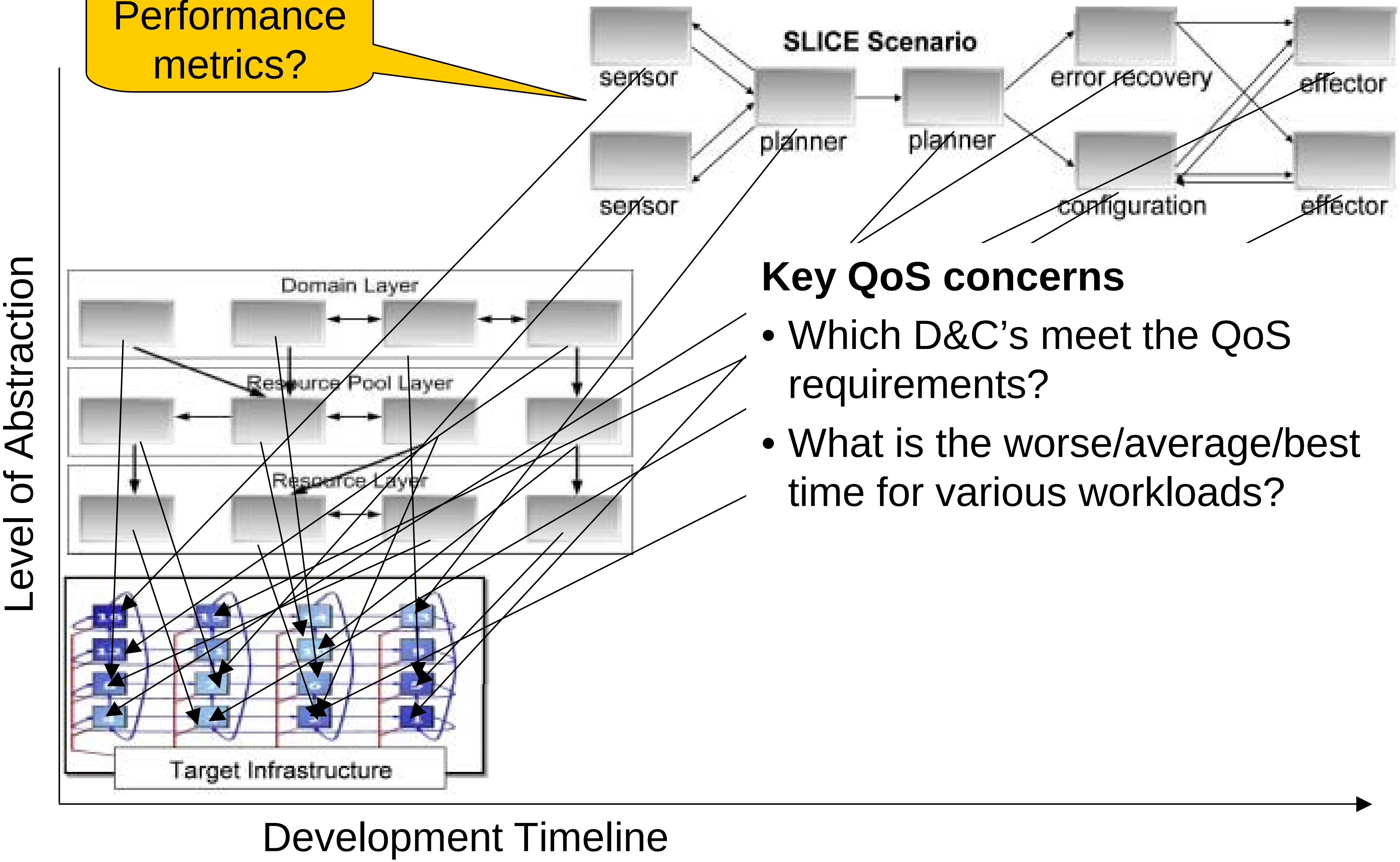


Key QoS concerns

- Which D&C's meet the QoS requirements?

QoS Concerns with Serialized Phasing

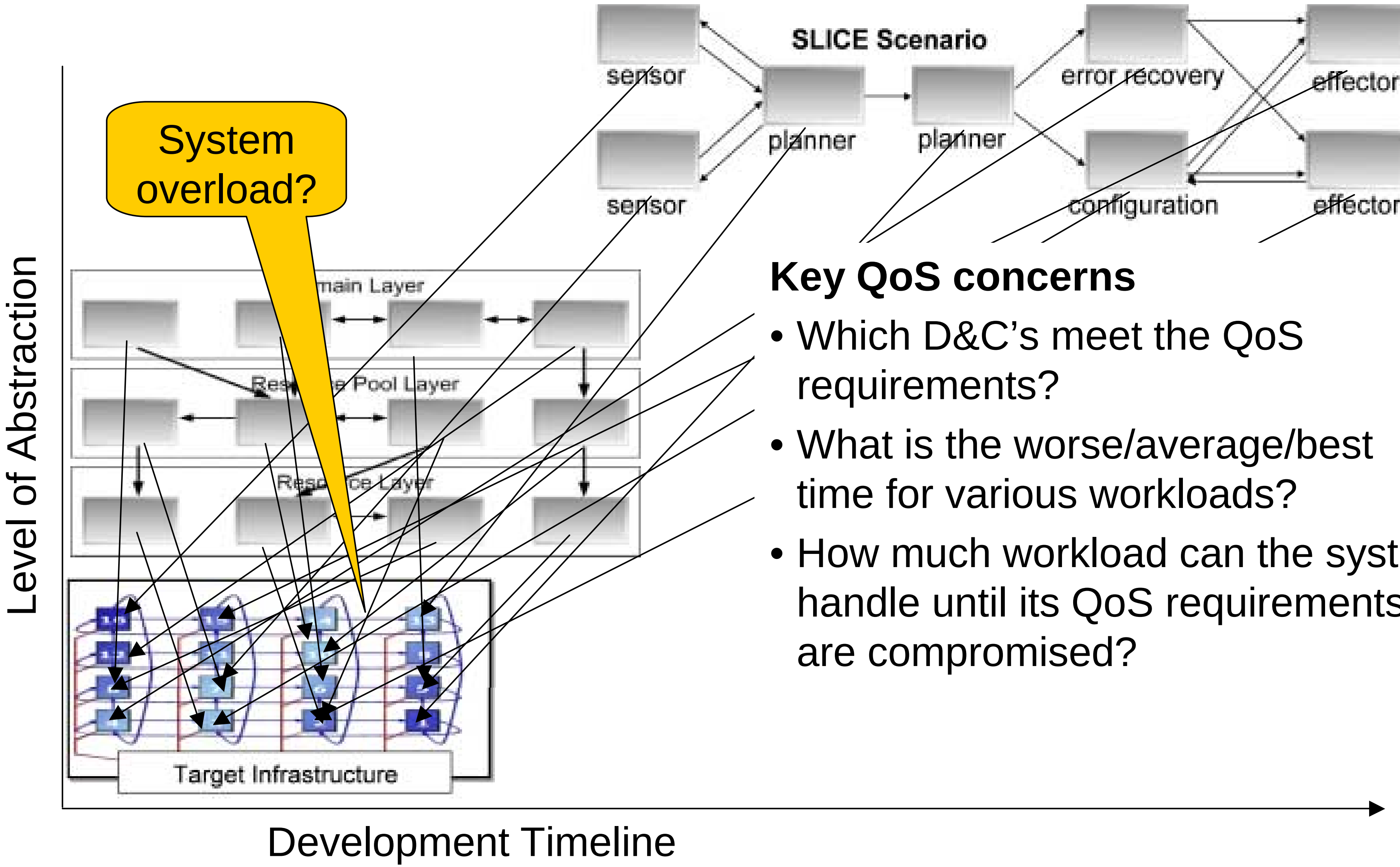
Performance metrics?



Key QoS concerns

- Which D&C's meet the QoS requirements?
- What is the worse/average/best time for various workloads?

QoS Concerns with Serialized Phasing



Key QoS concerns

- Which D&C's meet the QoS requirements?
- What is the worse/average/best time for various workloads?
- How much workload can the system handle until its QoS requirements are compromised?

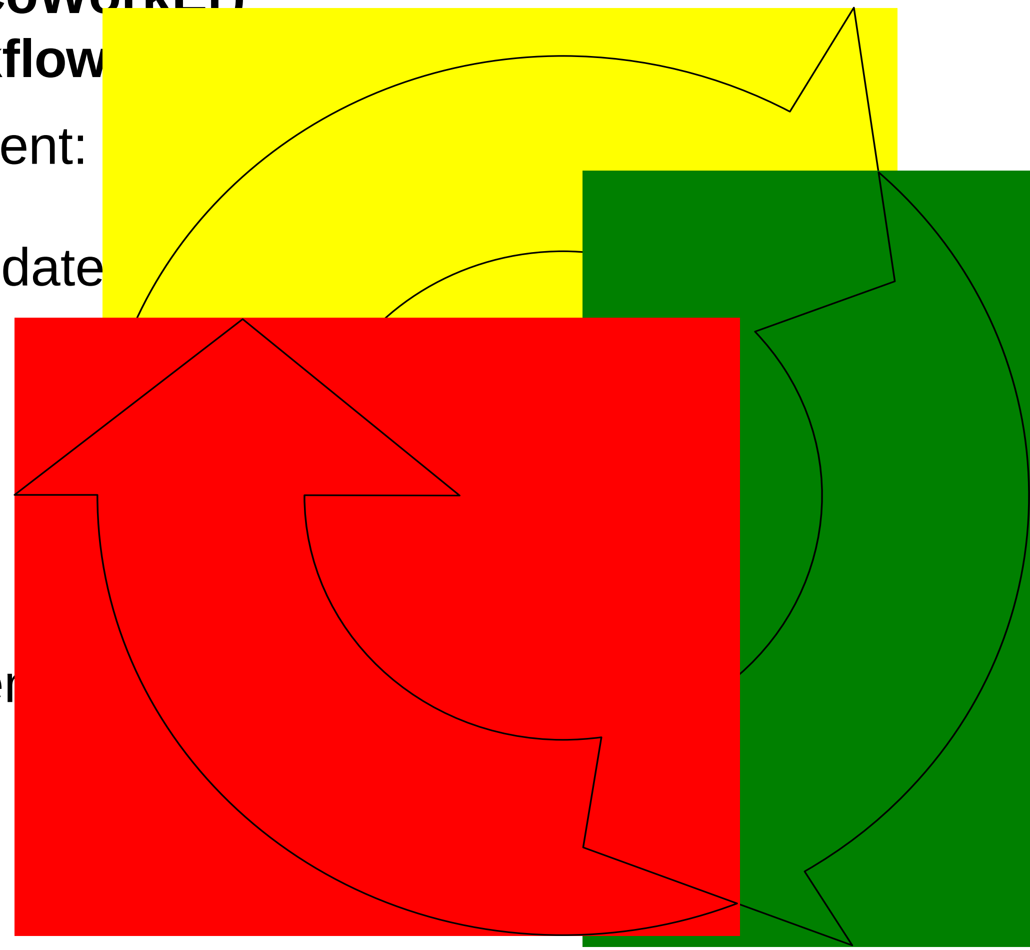
It is hard to address these concerns in processes that use serialized phasing

Approach: Emulate Behavior using Next Generation System Execution Modeling Tools

Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS) Workflow

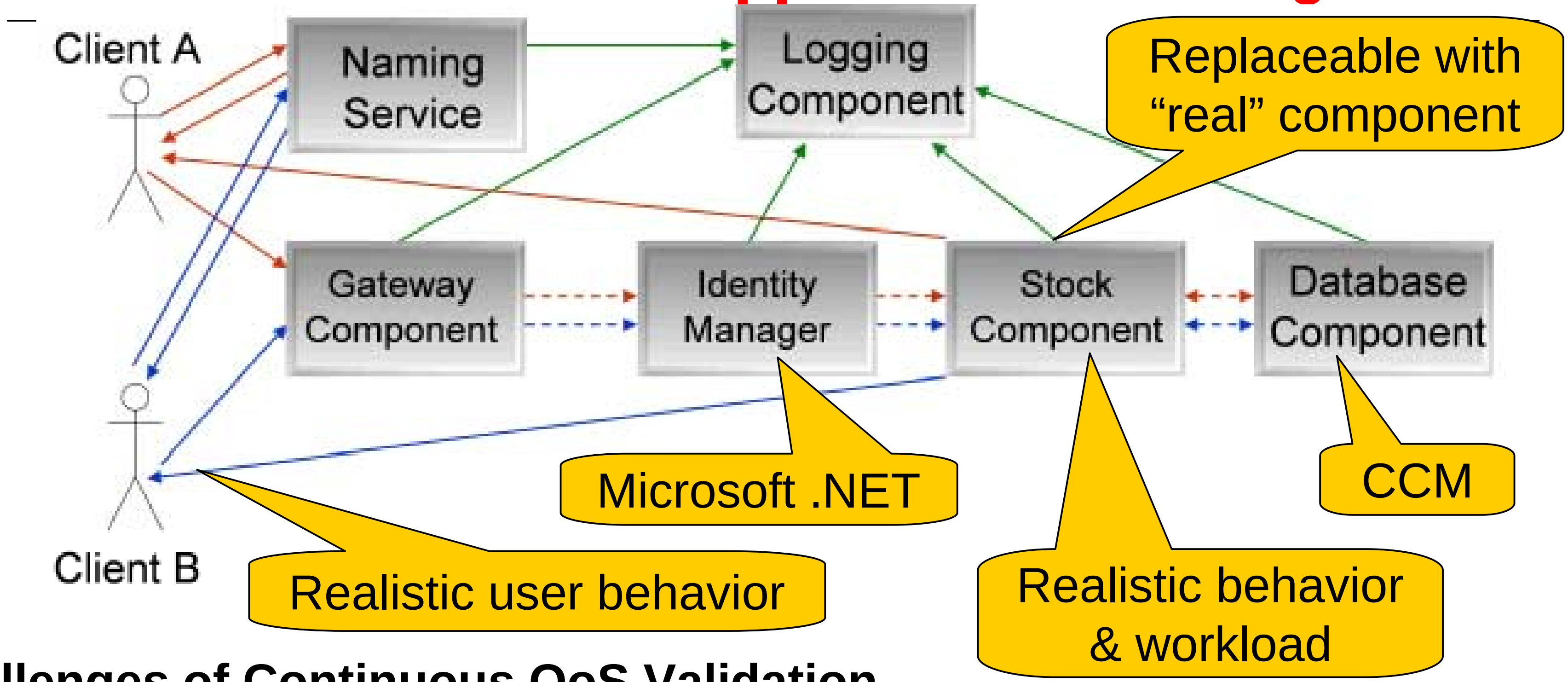
While target system under development:

1. Use a *domain-specific modeling language* (DSML) to define & validate infrastructure specifications & requirements
2. Use DSML to define & validate application specifications & requirements
3. Use middleware & MDE tools to generate D&C metadata so syster conforms to its specifications & requirements
4. Use analysis tools to evaluate & verify QoS performance
5. Redefine system D&C & repeat



Enables testing on target infrastructure throughout the development lifecycle
http://www.dre.vanderbilt.edu/~hillj/docs/publications/CUTS-RTCSA06.pdf

Distributed Stock Application Challenges



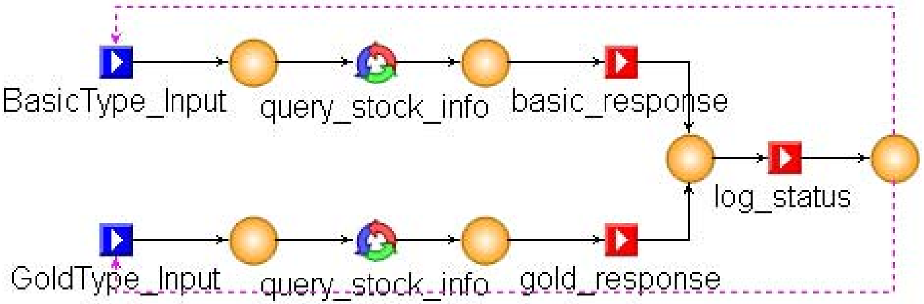
Challenges of Continuous QoS Validation

1. **Emulating Business Logic:** Emulated components must resemble their counterparts in both supported interfaces & behavior
2. **Realistic Mapping of Emulated Behavior:** Behavior specification should operate at a high-level of abstraction & map to realistic operations
3. **Technology Independence:** Behavior specification should not be tightly coupled to a programming language, middleware platform, hardware technology, or MDE tool

DSMLs for Continuous QoS Validation

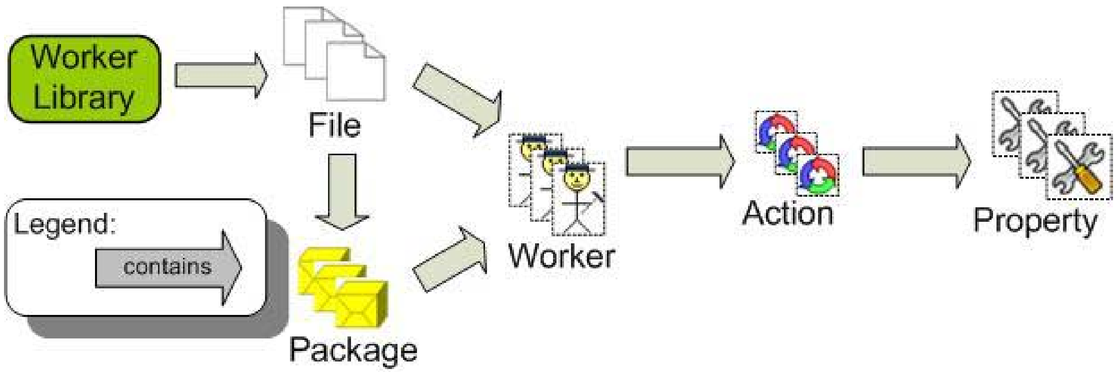
Component Behavior Modeling Language (CBML)

- a high-level domain-specific modeling language for capturing the behavior of components
  - e.g., its actions & states



Workload Modeling Language (WML)

- a domain-specific modeling language for capturing workload, & used to parameterize the actions on CBML



CBML & WML were both developed using GME (http://www.isis.vanderbilt.edu/projects/GME)

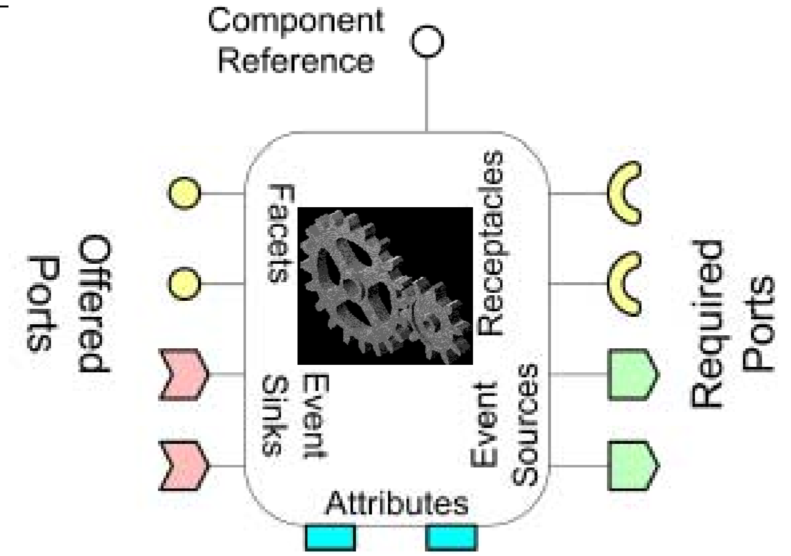
The Component Behavior Modeling Language

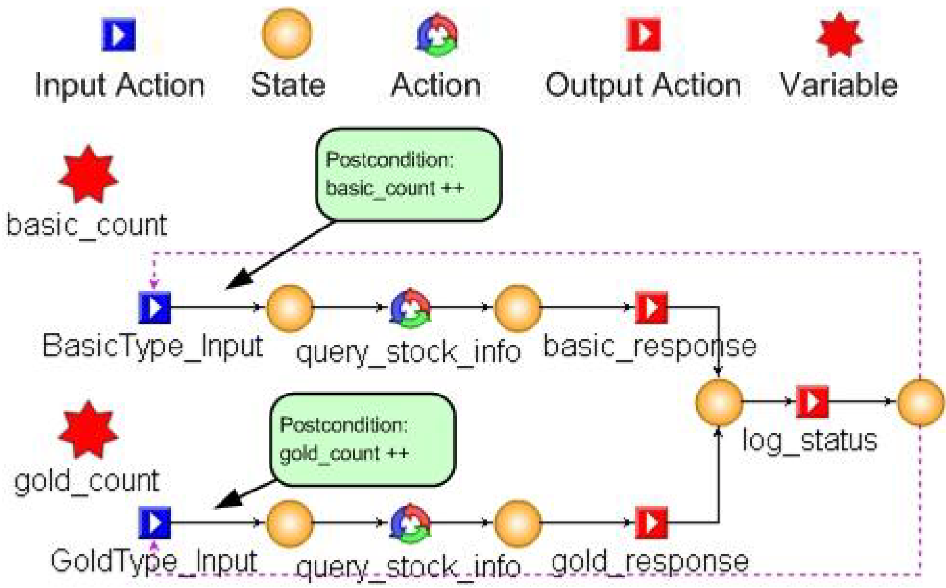
Context

- Component's behavior can be classified as: *communication* & *internal* actions
- Need to define these actions as close as possible to their real counterpart (i.e., Challenge 1)

Research Contributions of CBML

- Based on the semantics of Input/Output (I/O) Automata
    - i.e., contains representative elements
- Behavior is specified using a series of *action* to *state* transitions
- *Transitions* have *preconditions* that represent guards & *effects* have *postconditions* that determine the new state after an action occurs
- *Variables* are used to store state & can be used within pre & postconditions





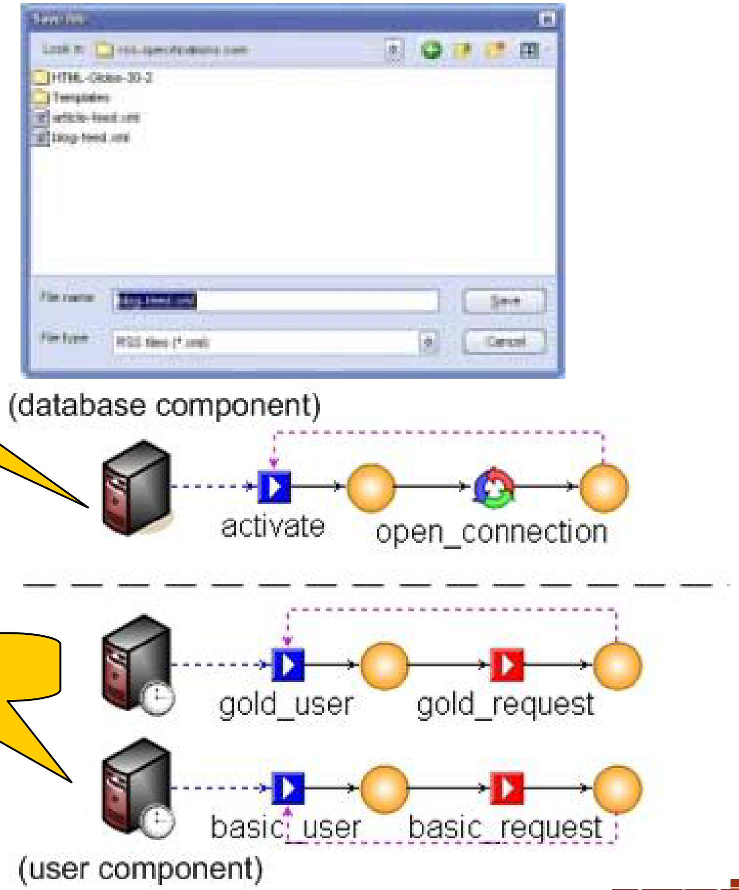
Domain-Specific Extensions to CBML

Context

- Some aspects of component-based systems are not first-class entities in I/O automata
  - e.g., lifecycle events & monitoring notification events
- Extended CBML (without affecting formal semantics) to support domain-specific extensions

Domain-Specific Extensions

- *Environment events* – input actions to a component that are triggered by the hosting system rather than another component
- *Periodic events* - input actions from the hosting environment that occur periodically



Ensuring Scalability of CBML

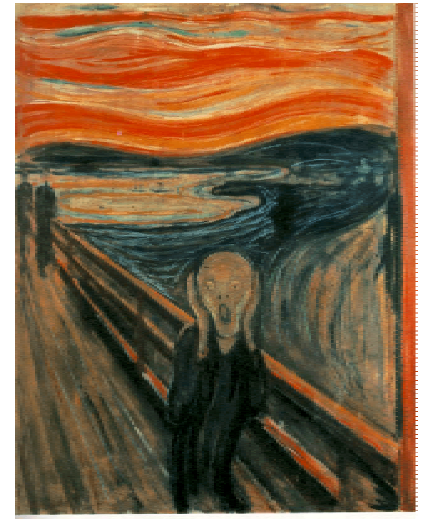
Context

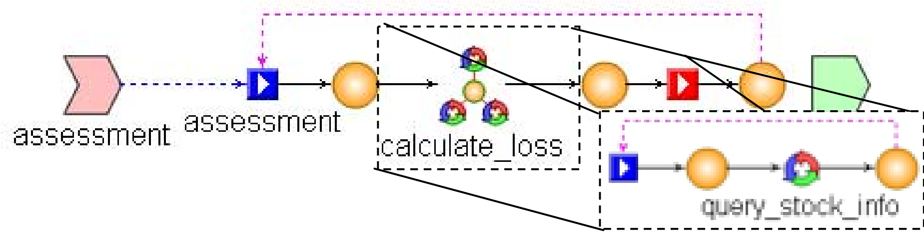
- One of the main goals of higher-level of abstraction is *simplicity* & *ease of use*
- It is known that one of the main drawbacks of automata languages is *scalability*

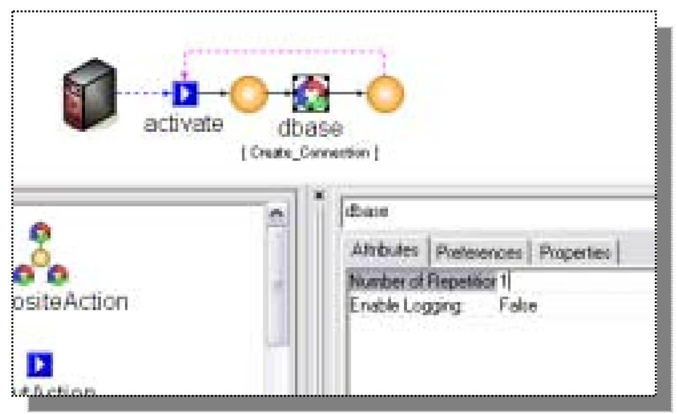
Usability Extensions

- *Composite Action* – contains other actions & helps reduce model clutter
- *Repetitions* - determines how many times to repeat an action to prevent duplicate sequential actions
- *Log Action* - an attribute of an Action element that determines if the action should be logged

Tool Specific – GME add-on that auto-generates required elements (e.g., states)







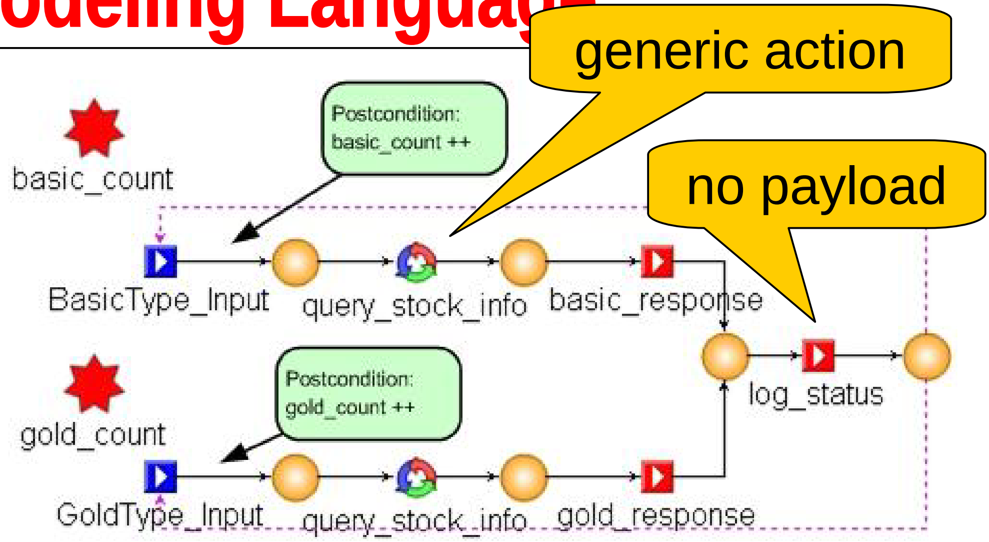
The Workload Modeling Language

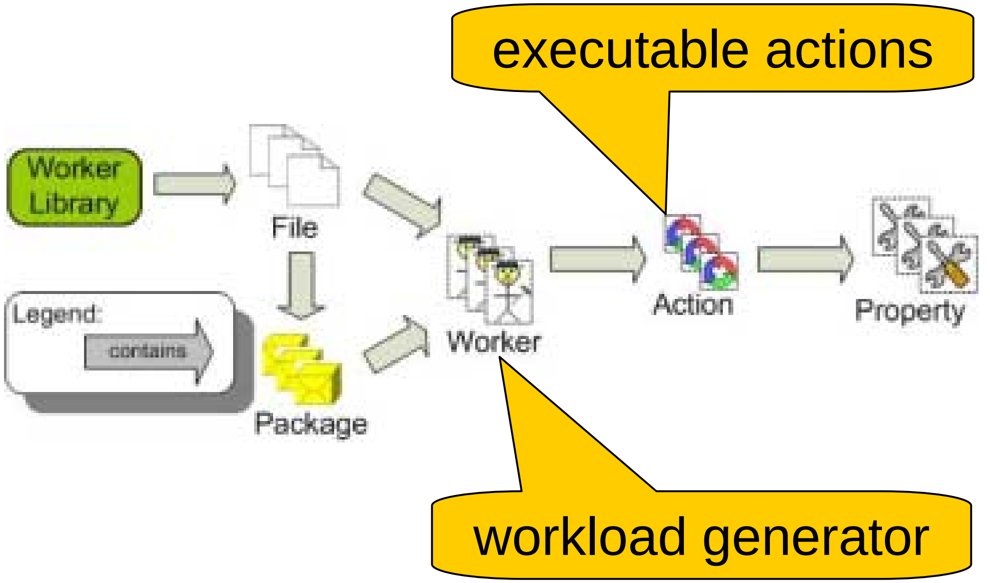
Context

- CBML as a standalone language is sufficient enough to capture behavior of a component
- For emulation purposes, it does not capture the reusable objects of a component & its workload (i.e., Challenge 2)

Research Contributions of WML

- Middleware, hardware, platform, & programming language independent DSML
- Used to define workload generators (workers) that contains actions to represent realistic operations
- Defined using a hierarchical structure that resembles common object-oriented programming packaging techniques that are consistent with conventional component technologies





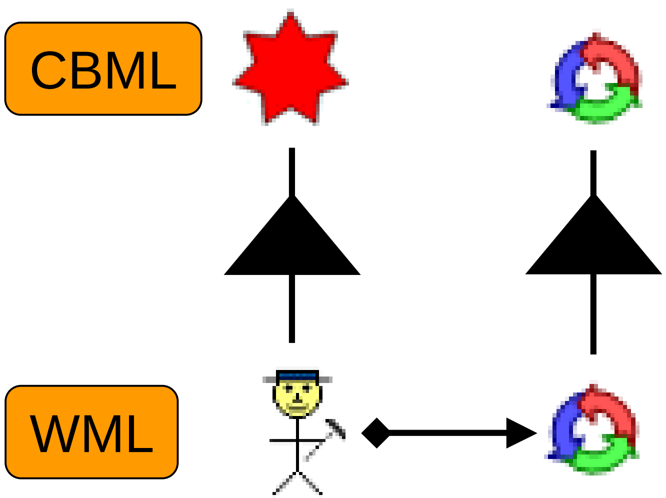
Integrating WML Models with CBML Models

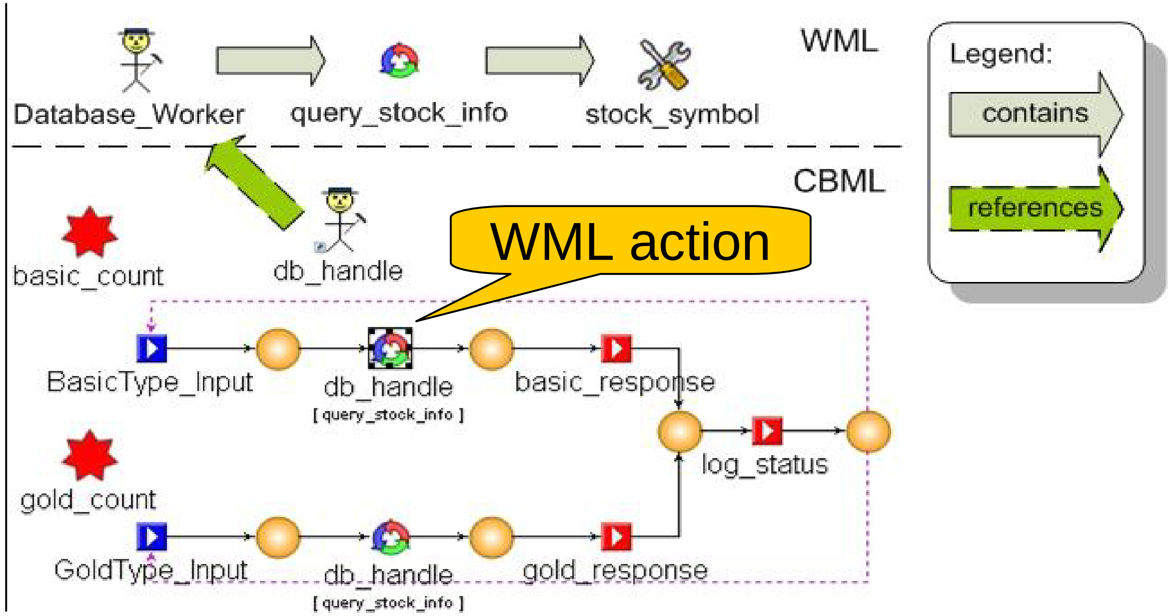
Context

- CBML & WML are standalone DSMLs with a distinct purpose
  - i.e., model behavior & workload, respectively
- WML is designed to complement CBML by providing CBML with reusable operations that can map to realistic operations

Integration Enablers

- WML *worker* elements have same model semantics as *variables*
- WML *actions* have same modeling semantics as CBML actions
- Allows WML elements to be used in CBML models

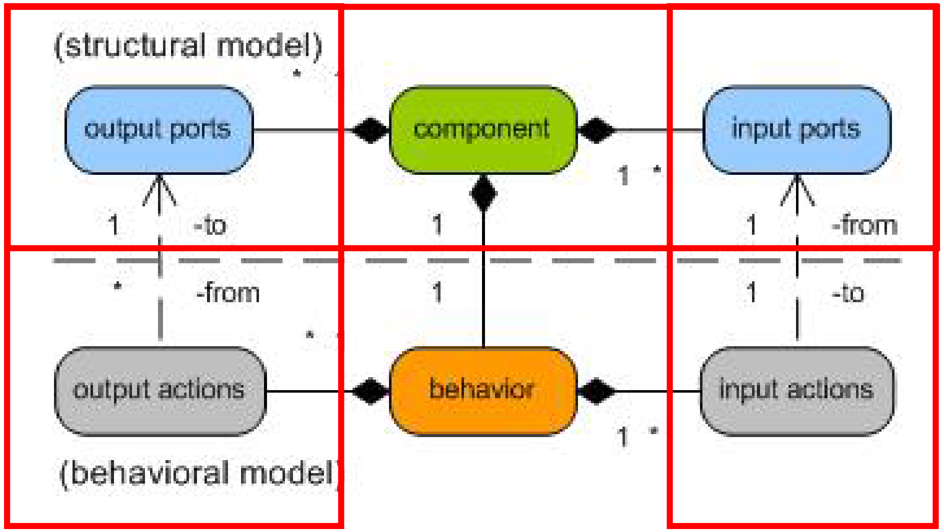




Integrating Behavioral and Structural DSMLs

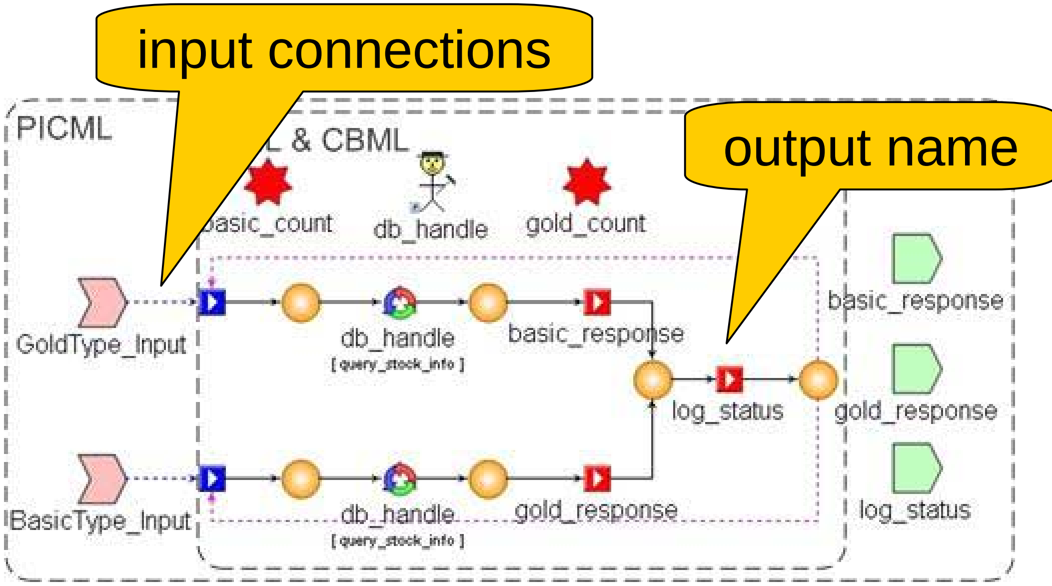
Context

- Structural DSML (e.g., the Platform Independent Component Modeling Language (PICML)) capture the makeup of component-based systems
- There is no correlation between a component's ports & its behavior

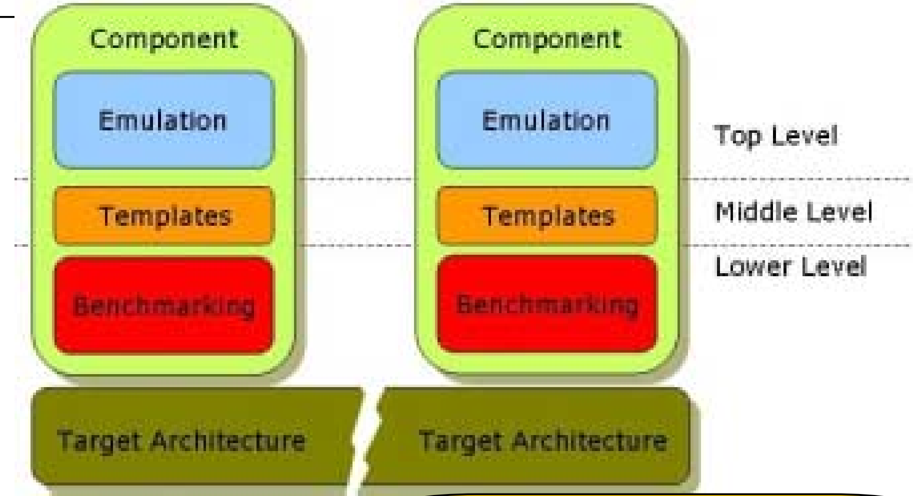


Integration Enablers

- Defined a set of *connector elements* that allow structural DSMLs to integrate with (or contain) CBML models
- Input ports directly connect to *Input Action* elements
- Output actions have the same name as their output port to reduce model clutter
  - i.e., prevent many to one connections



Code Generation for Emulation

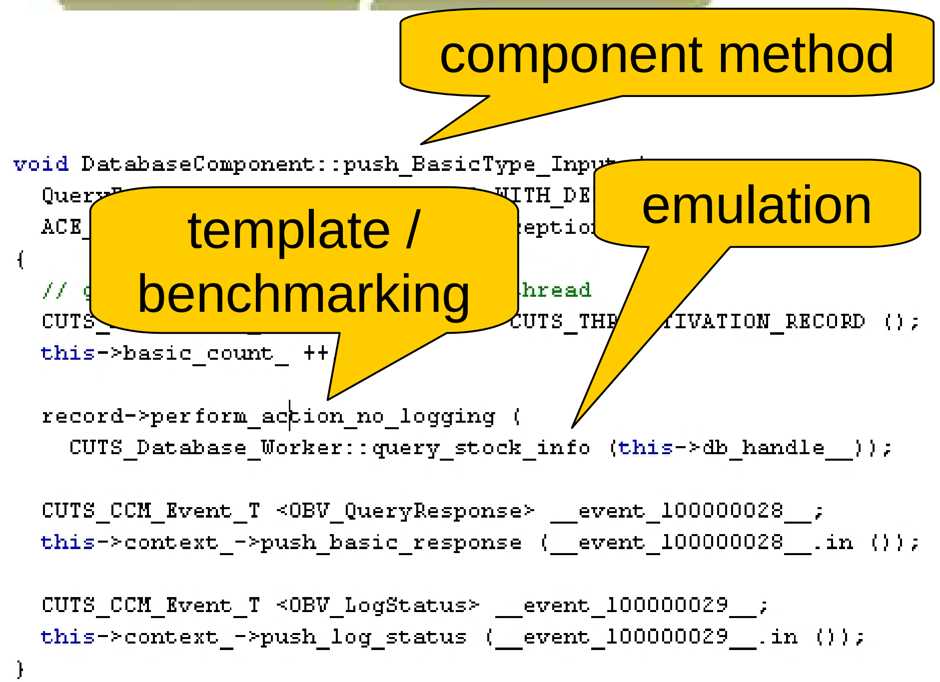


Context

- The generation technique should not be dependent on the underlying technology
- Although we are targeting CUTS, would should be able to generate emulation code for any benchmarking framework

Generation Enablers

- *Emulation layer* – represents the application layer's "business logic" where elements in WML used to parameterize CBML behavior are mapped to this layer
- *Template layer* – acts as a bridge between the upper emulation layer & lower benchmarking layer to allows each to evolve independently of each other
- *Benchmark layer* – the actual benchmarking framework (e.g., CUTS)



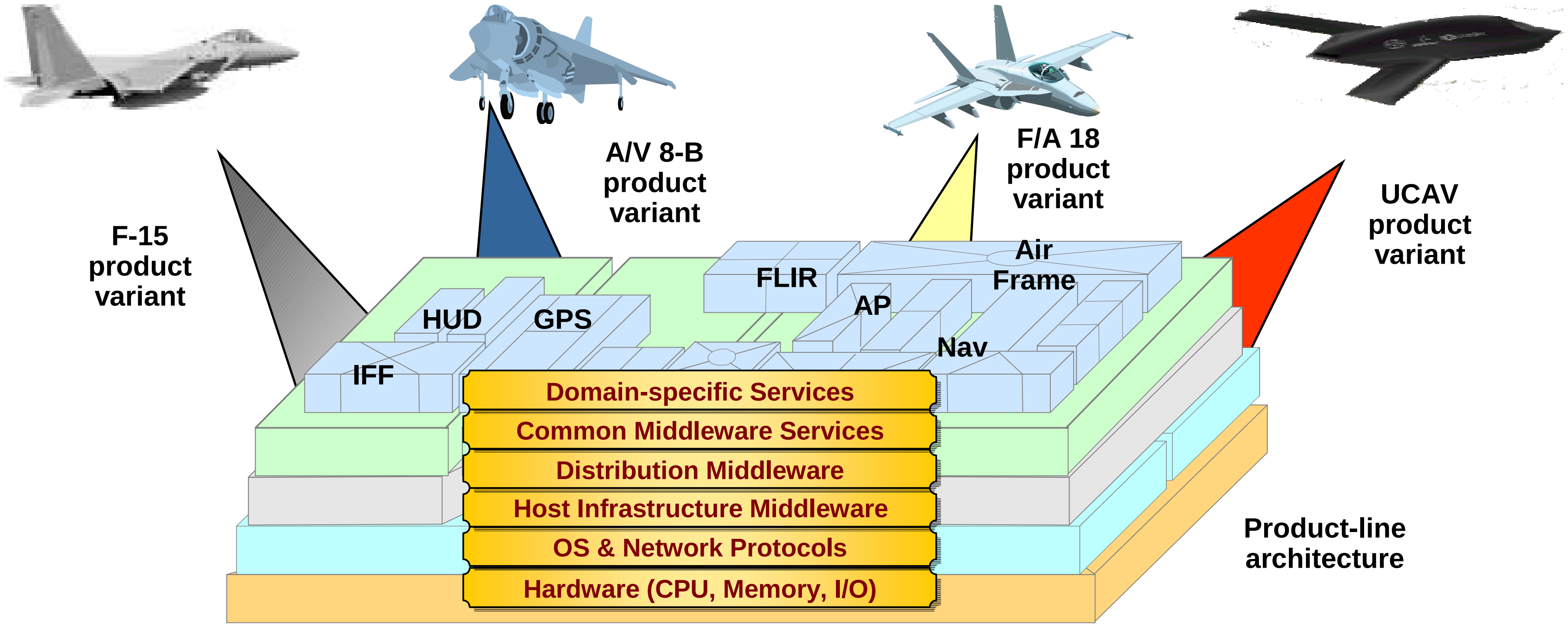
Part 8
Usecase-driven Middleware Optimizations

Feature Oriented CUStomizer (FOCUS)

&

Pattern Oriented Software Architecture Modeling Language (POSAML)

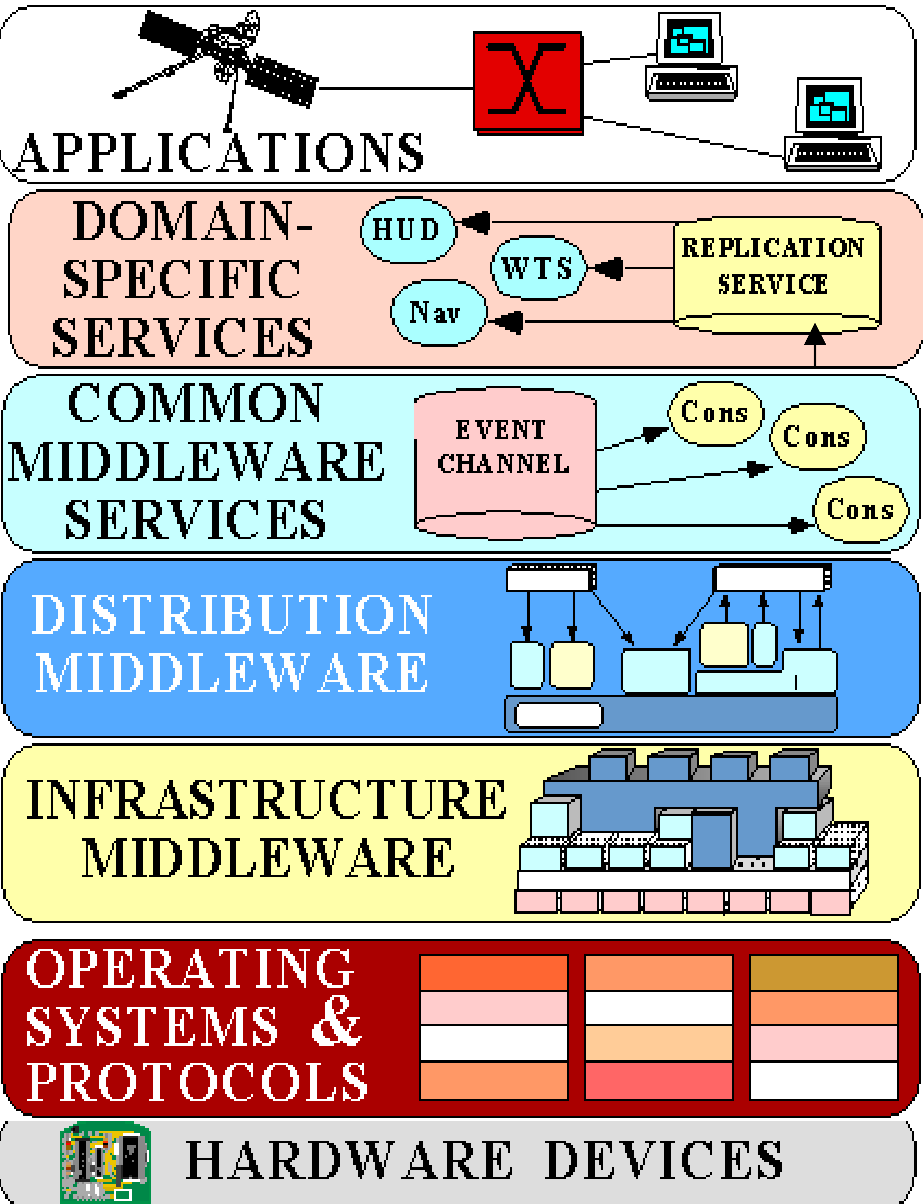
Optimizations for Product-line Architectures



- PLAs define a framework of components that adhere to a common architectural style with a clean separation of commonalities and appropriate provisions for incorporating variations
- ***Middleware*** factors out many reusable general-purpose & domain-specific services from traditional DRE application responsibility

Standards middleware is a key technology candidate for supporting and sustaining vision of software product-lines

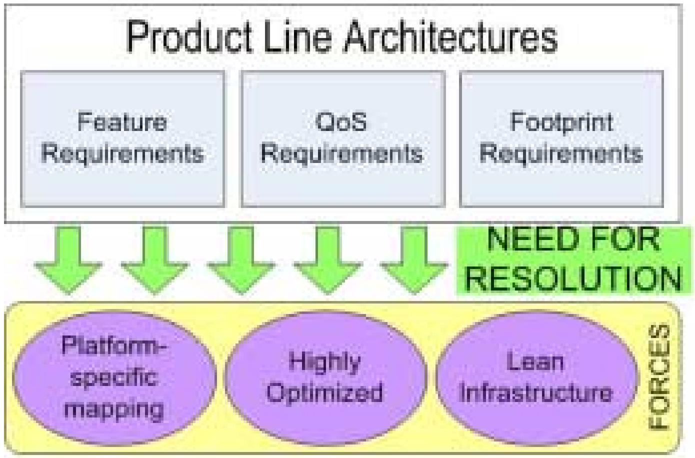
Middleware Structure & Functionality



- There are layers of middleware, just like there are layers of networking protocols
- Standards-based COTS middleware helps:
  - Control end-to-end resources & QoS
  - Leverage hardware & software technology advances
  - Evolve to new environments & requirements
  - Provide a wide array of reusable, off-the-shelf developer-oriented services
- **Problem**
  - Manually provisioning middleware is tedious, error-prone, & costly over system lifecycles

Need an intuitive, visual and declarative mechanism for middleware provisioning.

Technology Gaps in Middleware for PLAs



- PLAs have very ***"focused but crosscutting"*** requirements of underlying middleware infrastructure
  - Optimized for the platform
  - Lean footprint
  - Efficient configuration & deployment
  - Support run-time adaptations & reconfigurations
- Standards middleware development & optimizations philosophy catered to maintaining ***"generality, wide applicability, portability & reusability"***
  - *OS, compiler and hardware independent*
  - *e.g., CORBA, J2EE. .NET*
- These technology gaps are hindering PLA progress => adverse economic and societal consequences
  - e.g. shortcomings of pre-postulated

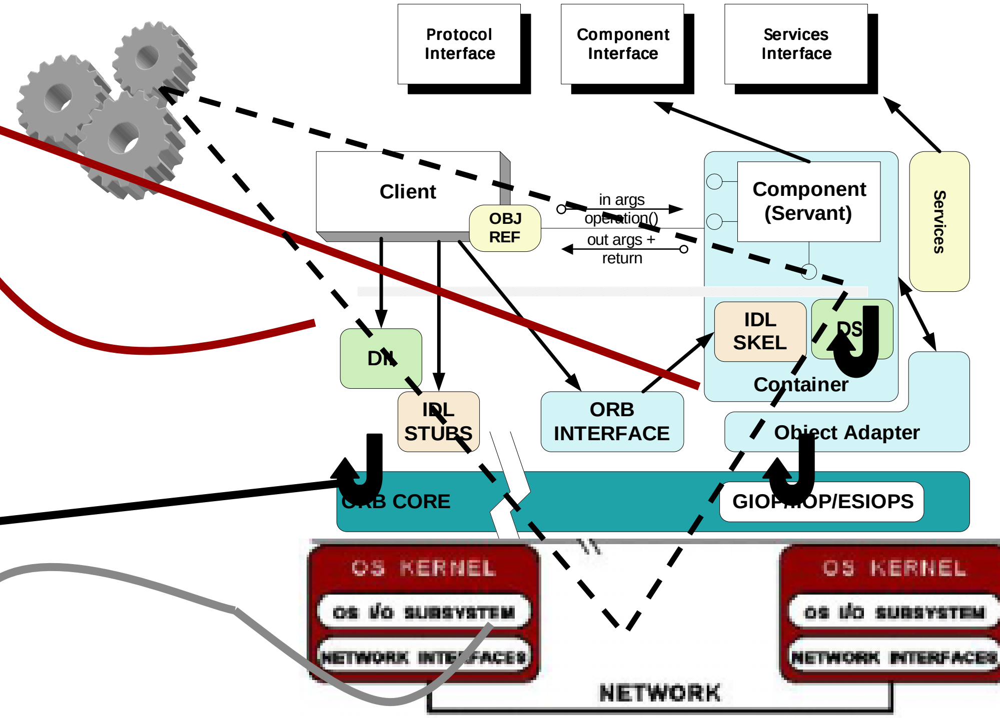
Need to tailor and optimize standards middleware for PLAs while continuing to provide standards compliance, portability and flexibility

Middleware Specialization Catalog

Specification-imposed specializations
- Layer-folding
- Constant propagation
- Memoization

Framework specializations
- Aspect Weaving techniques
  - Bridge → Reactor
  - Template method → Protocol
  - Strategy → Messaging, Wait, Demultiplexing

Deployment platform specializations
- Unroll copy loops
- Use emulated exceptions
- Leverage zero-copy data transfer buffers



- Development of reusable specialization patterns
- Identifying specialization points in middleware where patterns are applicable
- Domain-specific language (DSL) tools & process for automating the specializations

Feature Oriented CUStomizer (FOCUS)

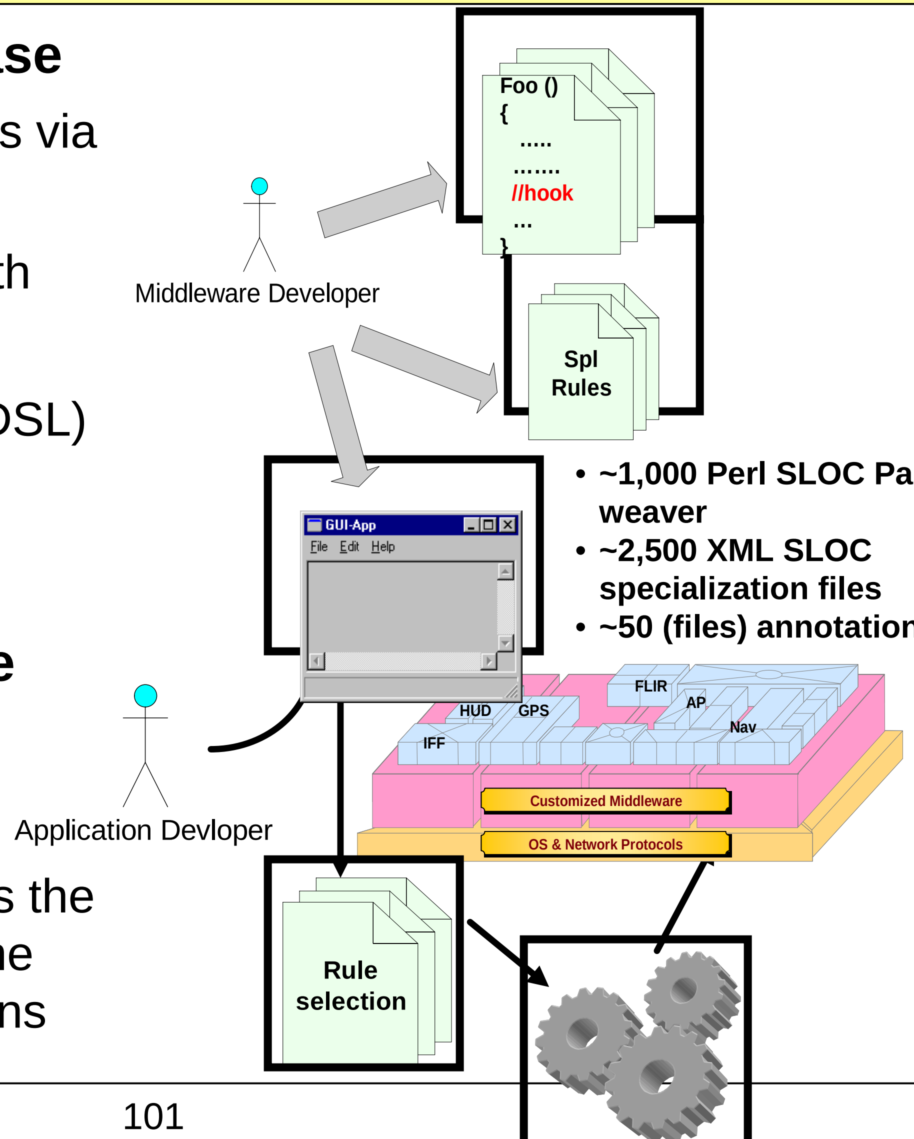
FOCUS addresses specialization challenges by building specialization language, tool, & process to capture & automate middleware specializations

Middleware Instrumentation Phase

- Capture specialization transformations via FOCUS specialization language
- Annotate middleware source code with specialization directives
- Create a domain-specific language (DSL) to capture middleware variability

Middleware Specialization Phase

- Analyzes & determines the type of specializations applicable
- *FOCUS transformation engine* selects the appropriate transformations & uses the annotations to automate specializations



- **~1,000 Perl SLOC Parser + weaver**
- **~2,500 XML SLOC specialization files**
- **~50 (files) annotations**

FOCUS Specialization Language (FSL)

FOCUS uses an XML DTD to create a DSL for capturing

Capability to specialize base implementations

- **Framework specialization requires code to be copied from derived to base classes**

FSL Approach

<copy-from-source>: Copy code from a specific point to a specific point between different files

<start-hook> → start copying

<end-hook> → hook to stop copying

<dest-hook> → destination

Capability to perform code substitutions

- **Devirtualize interfaces**
- **Replace base classes with derived class**

FSL Approach

- **<substitute>: Capability to do <search> … <replace> on code**

Capability to weave code at specified points

- **The layer folding specializations require code to be woven in along the request processing path**

FSL Approach

- **<add>: <hook> that specifies the annotated point; <data> where data is specified**

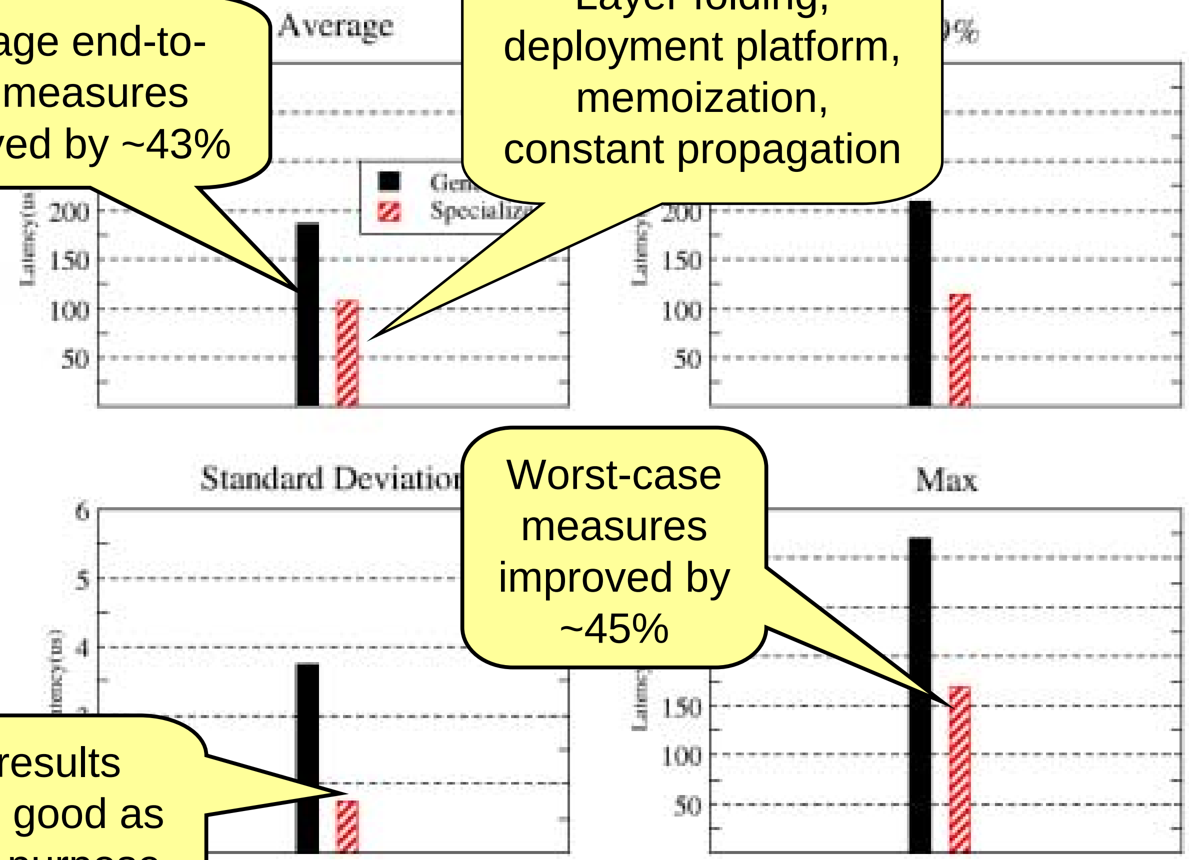
```
<substitute>
 <search>ACE_Reactor_Impl</search>
 <replace>ACE_Select_Reactor_Impl</replace>
</substitute>
```

```
<copy-from-source>
  <source>Select_Reactor.h</source>
  <copy-hook-start>HOOK-START </copy-hook-start>
  <copy-hook-end>HOOK-END </copy-hook-end>
  <dest-hook>HOOK-COPY</dest-hook>
</copy-from-source>
```

```
<add>
<hook>FORWARD_DECL</hook>
 <data>#include
“Select_Reactor.h”</data>
<add>
```

Cumulative Specialization Results

- **Applied to Bold Stroke BasicSP**
- **Specification related**
  - Layer folding
  - Memoization
  - Constant propagation (ignoring endianess)
- **Framework**
  - Aspect weaving (Reactor + protocol)
- **Deployment**
  - Loop unrolling + emulated exceptions



Average end-to-end measures improved by ~43%

Layer folding, deployment platform, memoization, constant propagation

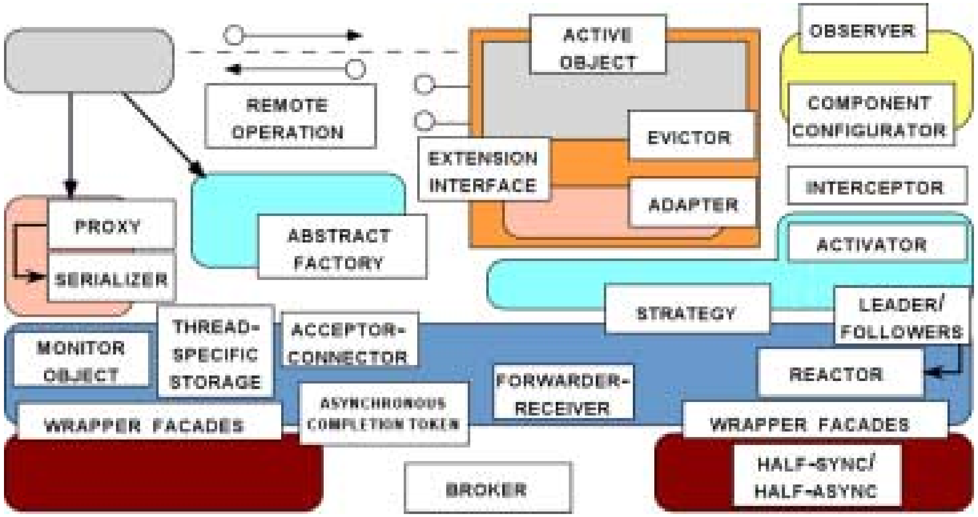
Worst-case measures improved by ~45%

Jitter results twice as good as general-purpose optimized TAO

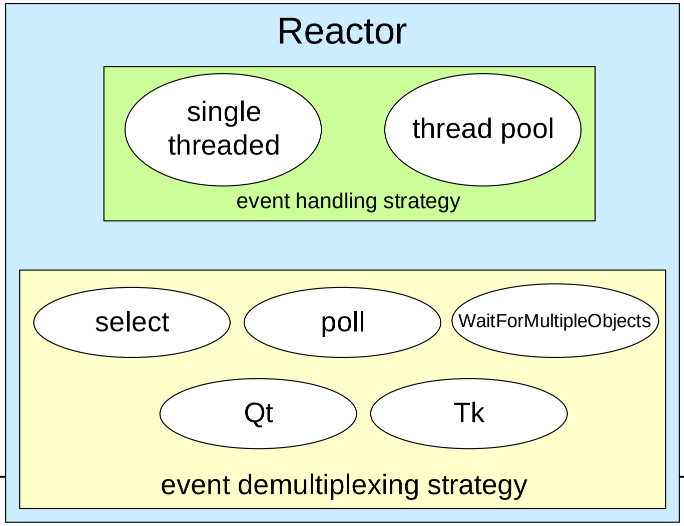
- **End-to-end client side throughput improved by ~65%.**
- **Results exceeded the hypothesis & evaluation criteria**

Automation Tool Requirements (1/2)

Criteria 1: Intuitive management of middleware variabilities that impact performance in significant ways

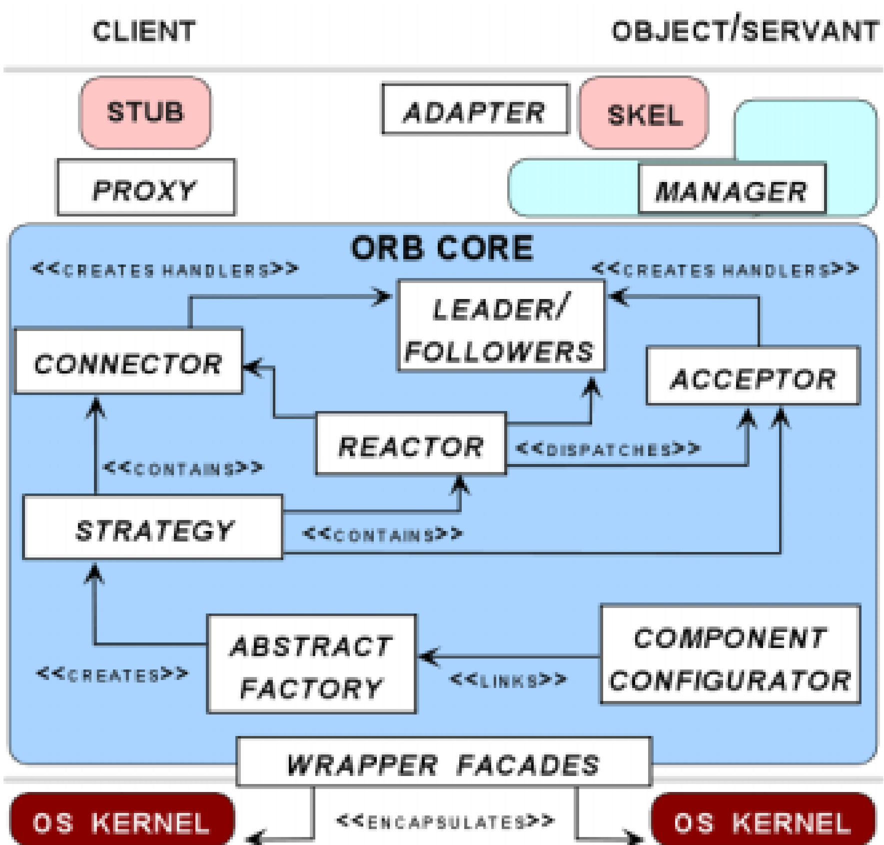


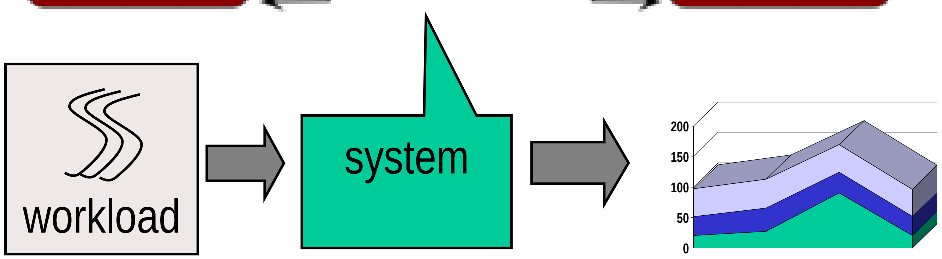
- **Tool must account for Compositional Variability**
  - Incurred due to variations in the compositions of mw building blocks
  - Need to address compatibility in the compositions and individual configurations
  - Dictated by needs of the domain
  - E.g., Leader-Follower makes no sense in a single threaded Reactor



- **Must account for Per-Block Configuration Variability**
  - Incurred due to variations in implementations & configurations for a patterns-based building block
  - E.g., single threaded versus thread-pool based reactor implementation dimension that crosscuts the event demultiplexing strategy (e.g., `select`, `poll`, `WaitForMultipleObjects`

Automation Tool Requirements (2/2)





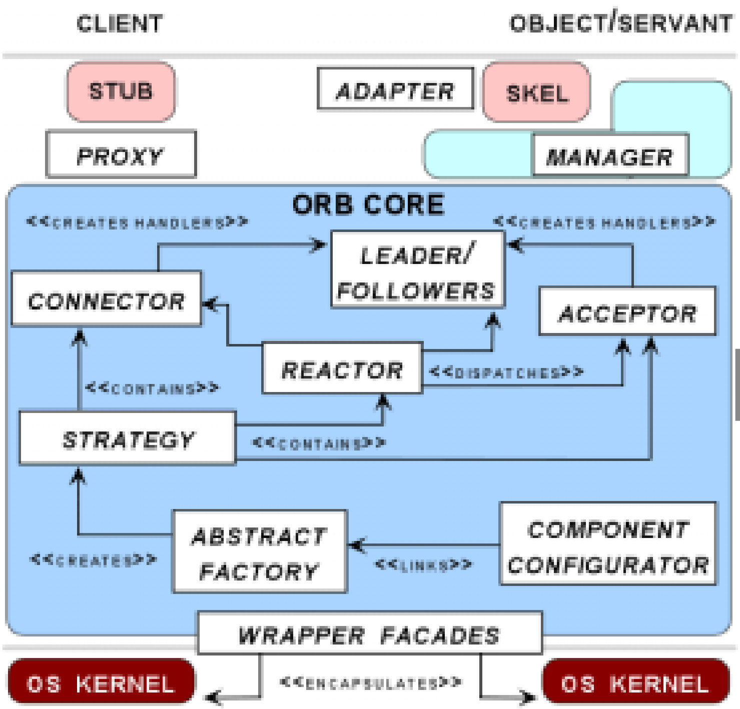
Criteria 2: Visual separation of concerns within the Unified Framework

- **Separation of concerns**
  - Unified framework must separate the provisioning and validating stages
  - Different actors should be able to use the visual aids in different stages of the application lifecycle

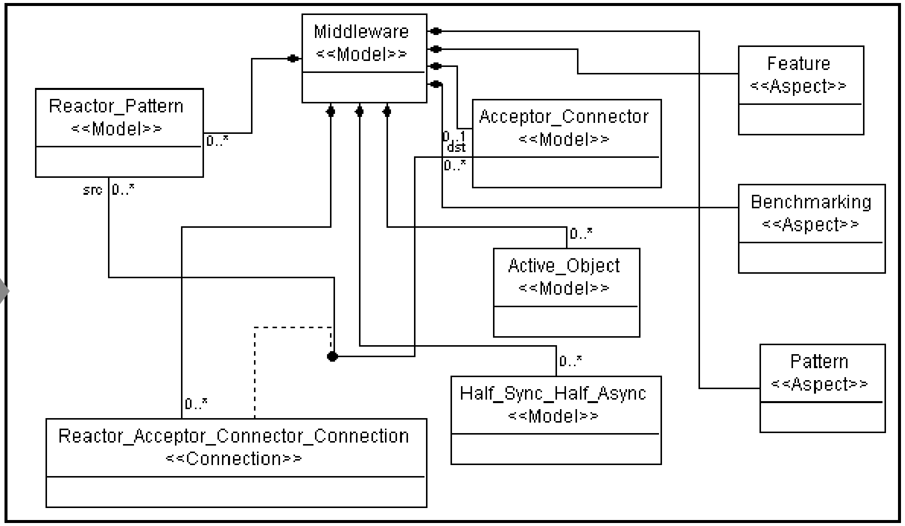
Criteria 3: Unified framework for middleware provisioning and QoS validation

- **Unified framework for provisioning and validating**
  - Provisioning decisions should be coupled with QoS validation
  - Decisions at one stage drive decisions at the next stage

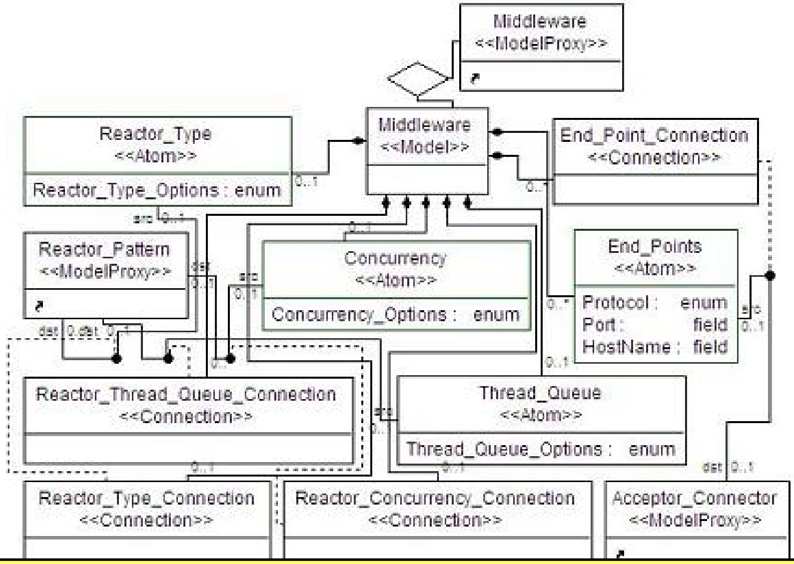
POSAML: A Visual Provisioning Tool



Metamodel for the POSA pattern language

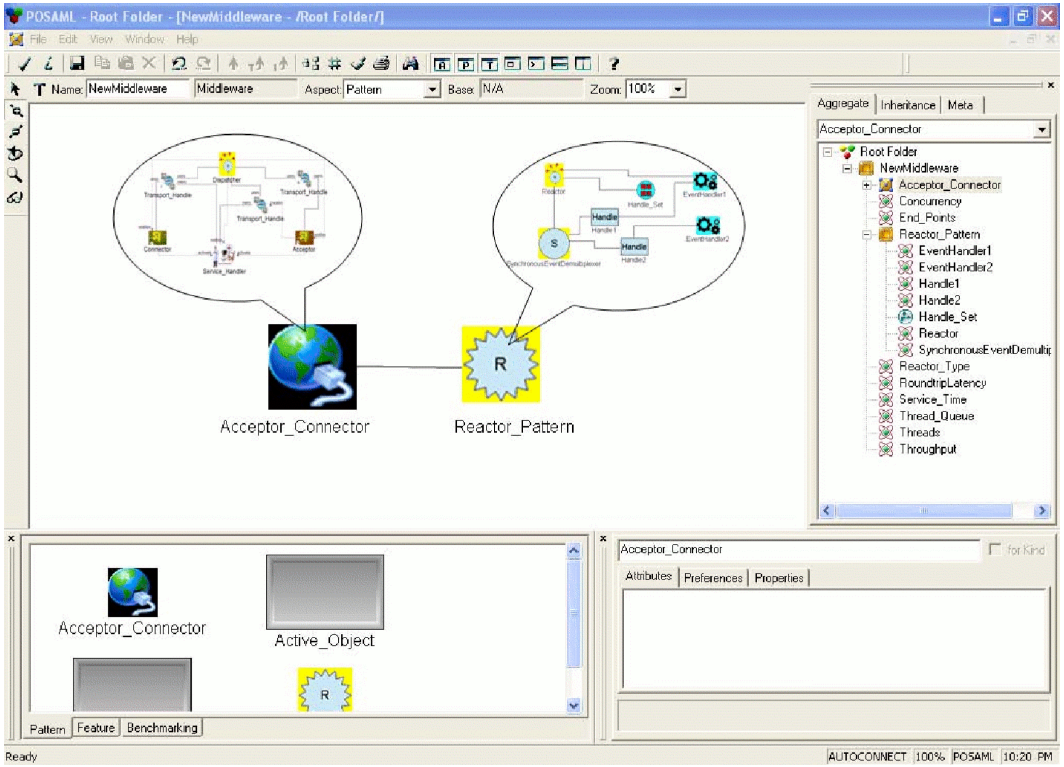


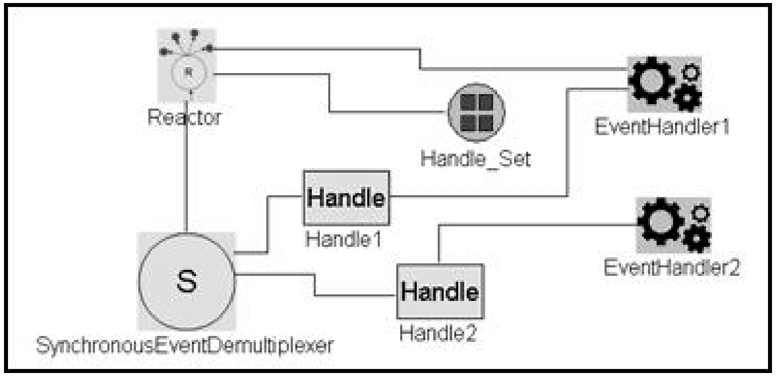
- POSAML – GME-based modeling language for middleware composition
- Provides a structural composition model
- Captures variability in blocks
- Generative programming capabilities to synthesize different artifacts e.g., benchmarking, configuration, performance modeling.

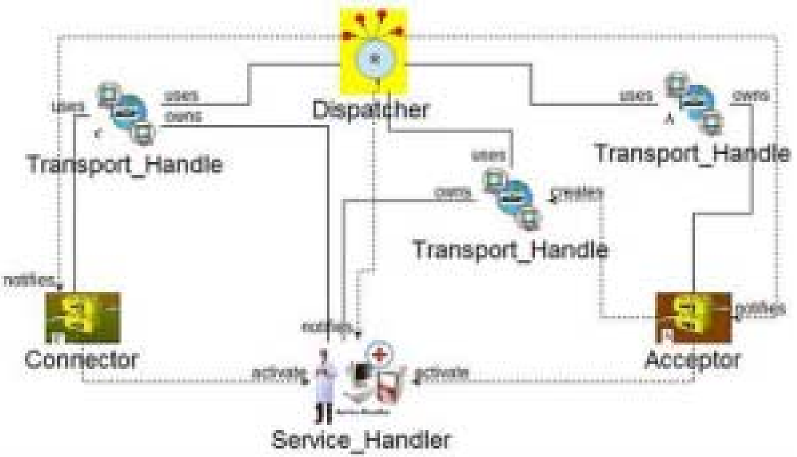


Feature modeling metamodel in POSAML

POSAML Unified Framework

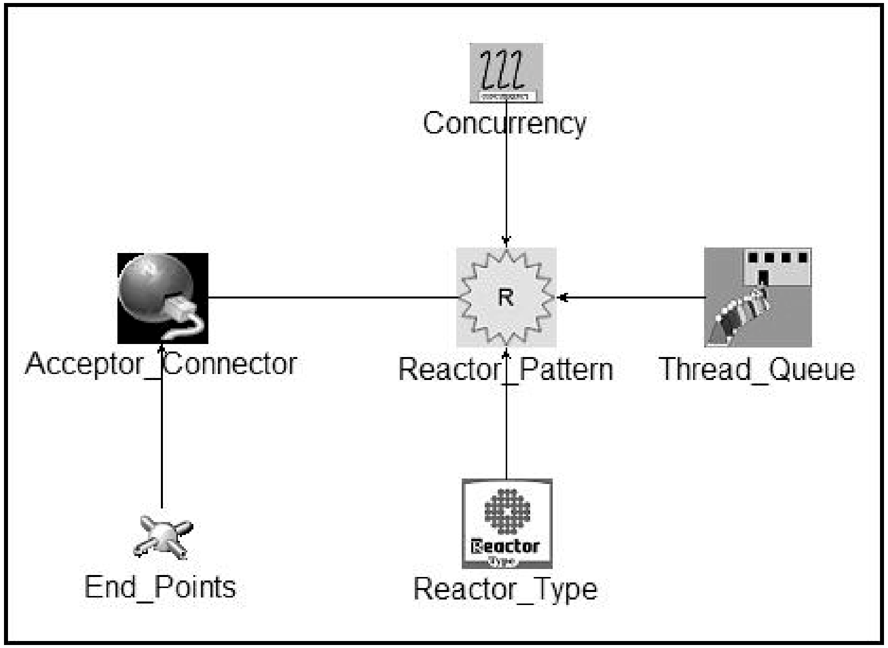


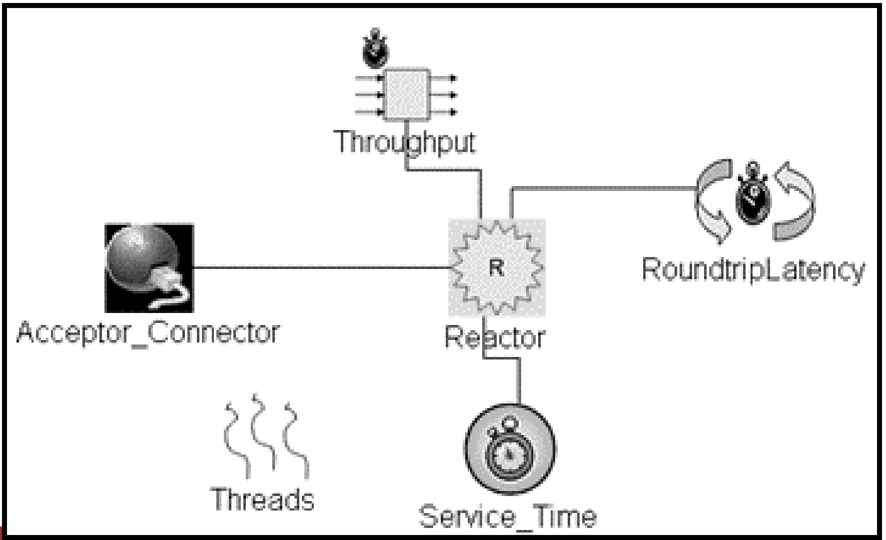




- Unified visual view enables modeling the middleware composition as a set of interacting patterns
- Individual patterns can be visually configured
  - E.g., reactor and acceptor-connector patterns
- POSAML languages conforms to the POSA pattern language enabling error-free composition of building blocks.

POSAML Separation of Concerns





- POSAML separates pattern feature modeling from pattern benchmarking
- Feature model allows selecting features of each pattern
  - E.g., reactor and acceptor-connector shown with concurrency models
- Benchmarking view separated from feature view
  - E.g., selecting parameters for elements of the pattern
- Views are unified *under the hood*

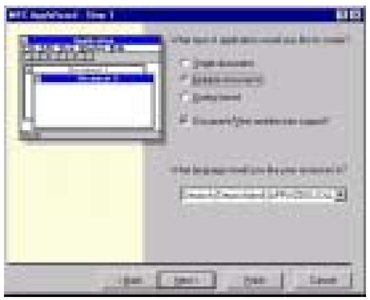
Structure Spectrum of DSLs

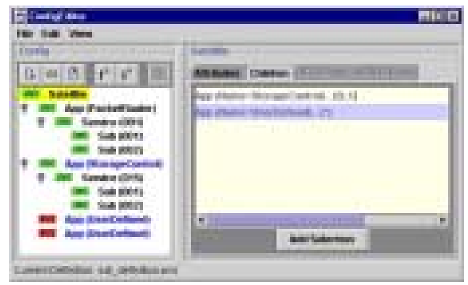
Routine configuration → Creative construction

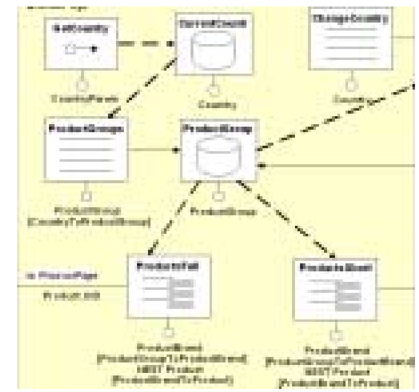
Wizards

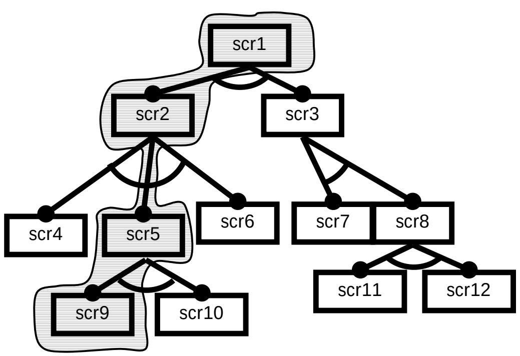
Feature-based configuration

Graph-like language (with user-defined elements)

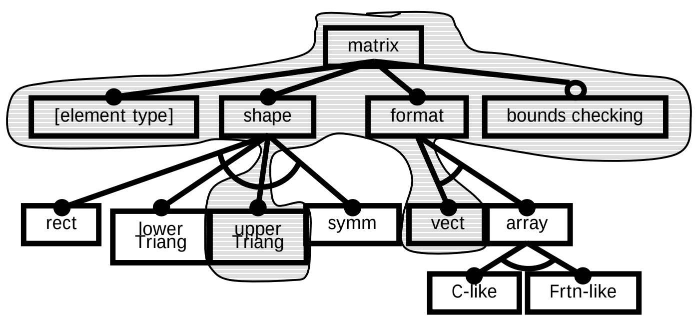




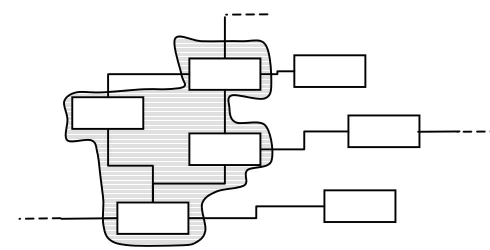




Path through a decision tree



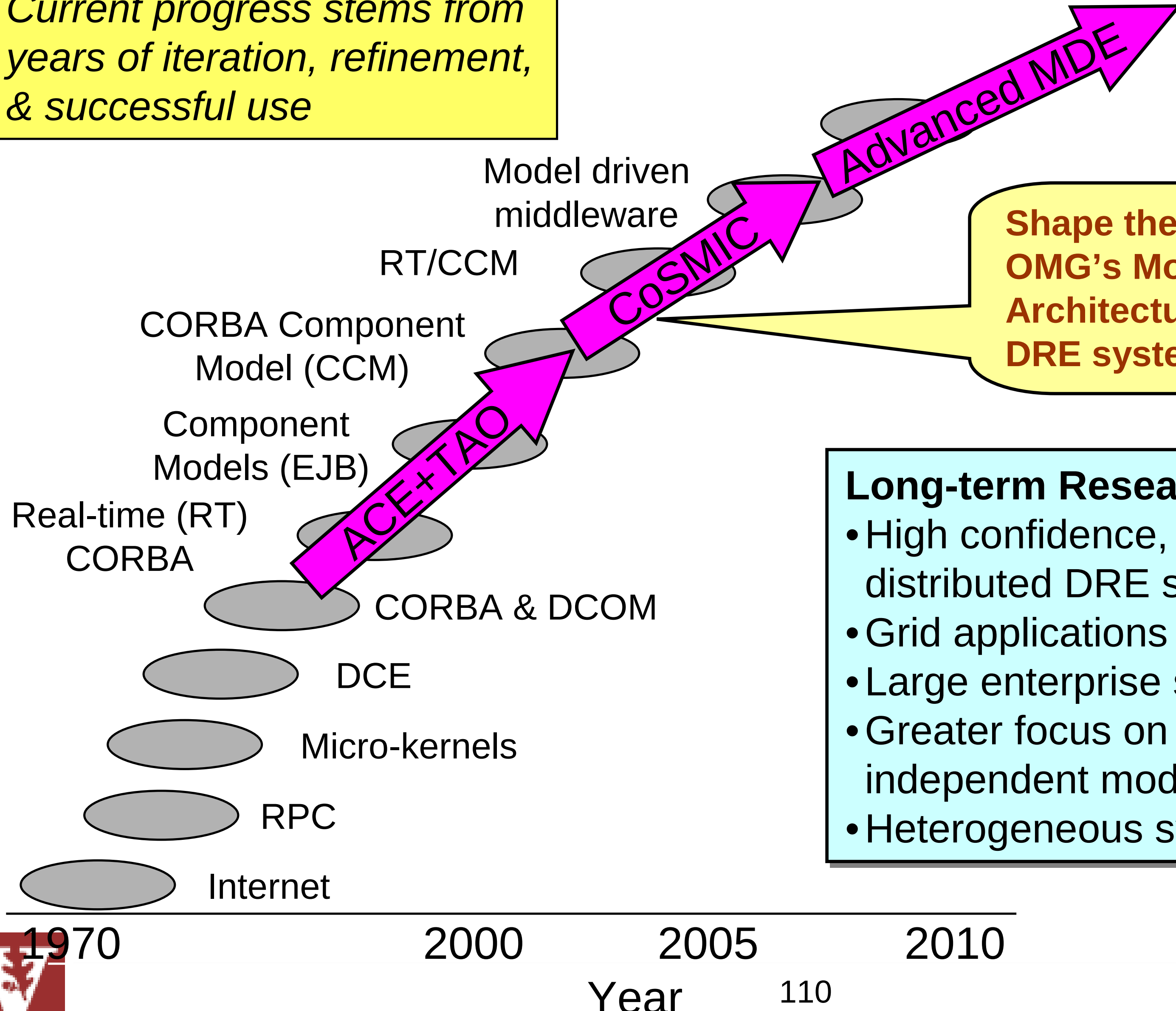
Subtree of a feature tree



Subgraph of an (infinite) graph

Research Impact & Future Work

Current progress stems from years of iteration, refinement, & successful use



Shape the standards e.g., OMG's Model Driven Architecture (MDA) for DRE systems

Long-term Research Directions

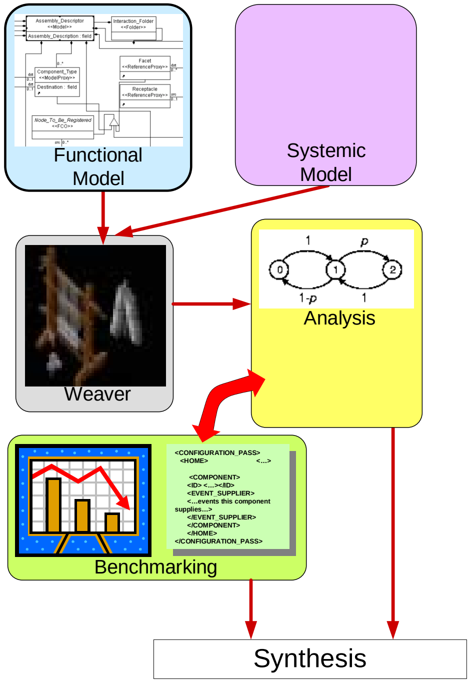
- High confidence, geographically distributed DRE systems
- Grid applications
- Large enterprise systems
- Greater focus on platform-independent models
- Heterogeneous systems

Acknowledgments

This tutorial is made possible due to the efforts of students, collaborators & mentors

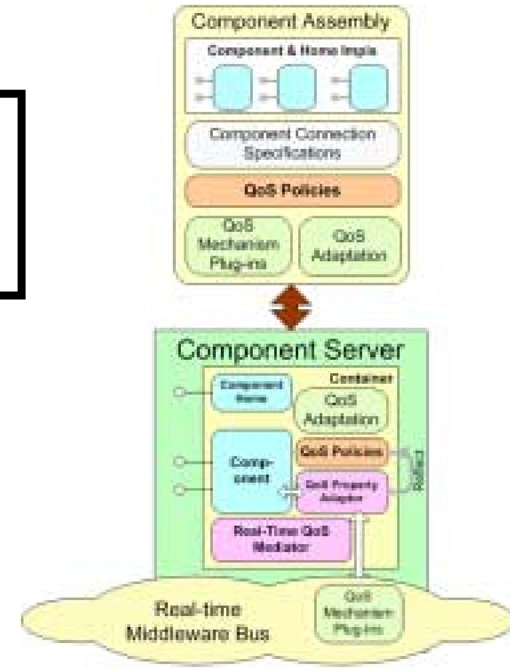
- **DOC Students & Staff**
  - Krishnakumar Balasubramanian
  - Arvind Krishna
  - Jaiganesh Balasubramanian
  - Emre Turkay
  - Jeff Parsons
  - Sumant Tambe
  - James Hill
  - Akshay Dabholkar
  - Amogh Kavimandan
  - Gan Deng
  - Joe Hoffert
  - George Edwards
  - Dimple Kaul
  - Arundhati Kogekar
  - Gabriele Trombetti
- **Collaborations**
  - Christopher Andrews
  - Theckla Louchios
  - Dr. Cemal Yilmaz
  - Dr. Sylvester Fernandez
  - Dr. Adam Porter
  - Thomas Damiano
  - Dr. Sherif Abdelwahed
  - Dr. Jeff Gray
  - Dr. Swapna Gokhale
- **Mentors**
  - Dr. Doug Schmidt
  - Dr. Janos Sztipanovits
  - Dr. Gabor Karsai
  - Dr. Joe Loyall
  - Dr. Joe Cross

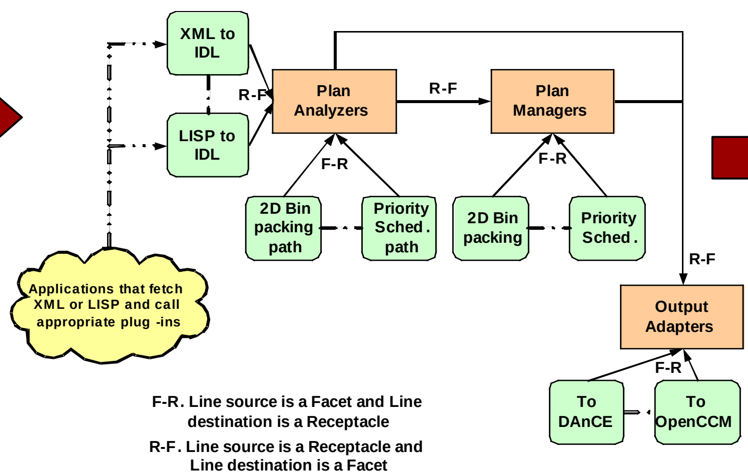
DOC Group Research on DRE Systems

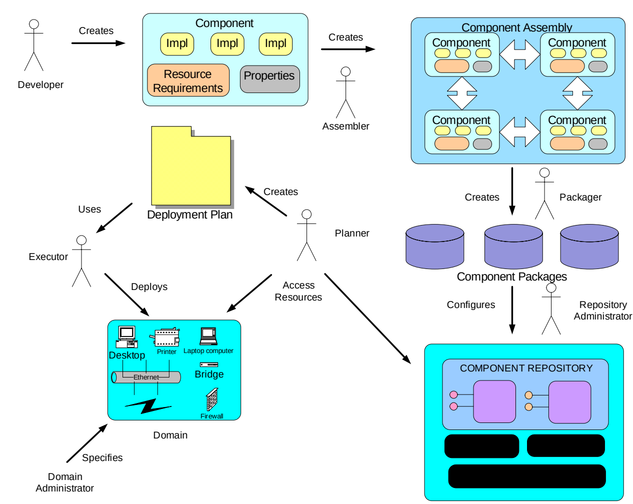


Multiple Levels of Abstraction

- **CoSMIC - Modeling Deployment & Configuration (D&C) crosscutting concerns**
- **RACE – Resource and Control Engine**
- **DAnCE – Deployment And Configuration Engine**
- **CIAO – QoS-enabled component middleware**



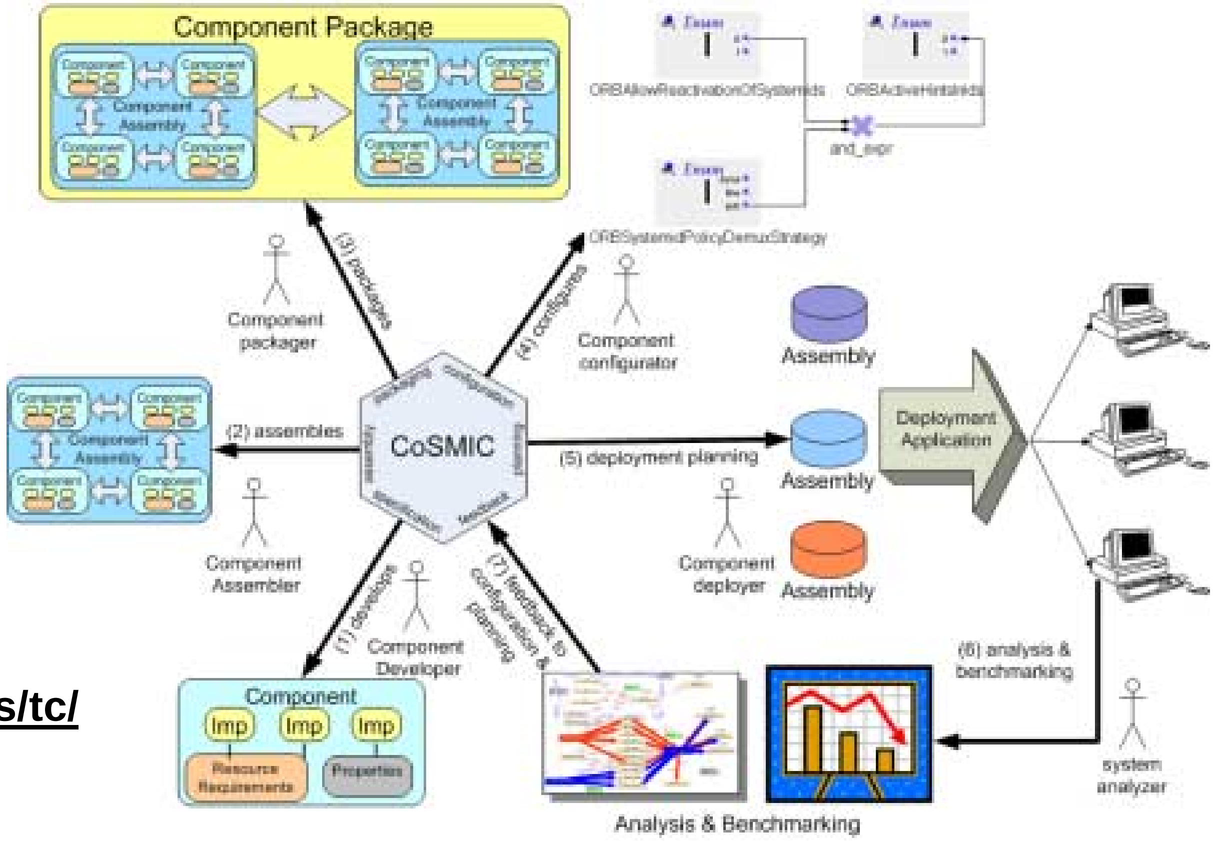




Concluding Remarks

- Model-Driven Engineering (MDE) is an important emerging generative technology paradigm that addresses key lifecycle challenges of DRE middleware & applications
- OMG PSIG on Model Integrated Computing

www.omg.org/news/meetings/tc/agendas/mic.htm



Many hard R&D problems with model-driven engineering remain unresolved!!

- CoSMIC MDE tools are based on the Generic Modeling Environment (GME)
  - CoSMIC is available from www.dre.vanderbilt.edu/cosmic
  - GME is available from www.isis.vanderbilt.edu/Projects/gme/default.htm