Applying Model-Driven Development to Distributed Real-time and Embedded Avionics Systems

Krishnakumar Balasubramanian, Arvind S. Krishna, Emre Turkay, Jaiganesh Balasubramanian,
Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt

***Abstract*— Model-driven development (MDD) is an emerging paradigm that uses domain-specific modeling languages (DSMLs) and generative technologies to provide "correct-by-construction" capabilities for many software development activities. This paper provides two contributions to the study of applying MDD to distributed real-time and embedded (DRE) systems that use standards-based quality of service (QoS)-enabled component middleware. First, it describes an MDD toolsuite called *Component Synthesis using Model-Integrated Computing* (CoSMIC), which is a collection of DSMLs and generative tools that support the development, configuration, deployment, and validation of component-based DRE systems. Second, it describes how we have applied CoSMIC to an avionics mission computing application to resolve key component-based DRE system development challenges. Our results show that the design-, deployment- and quality assurance (QA)-time capabilities provided by the DSMLs and generative capabilities in CoSMIC help to eliminate key complexities associated with specifying packaging, configuring, validating, and deploying QoS-enabled component middleware applications in the DRE systems domain.**

***Index Terms*— Model-driven Development, CORBA Component Model, CoSMIC**

I. Introduction

Emerging trends and challenges: Reusable software components and standards-based component models are increasingly being used to develop large-scale distributed real-time and embedded (DRE) systems [1]–[3]. This trend provides many advantages compared with earlier forms of DRE infrastructure software. For example, components provide higher-level abstractions than operating systems, third-generation programming languages, and earlier generations of middleware, such as distributed object computing (DOC) middleware. In particular, component middleware supports multiple views per component, transparent navigation, greater extensibility, and a higher-level execution environment based on containers, which alleviate many limitations of prior middleware technologies.

The additional capabilities of component-based platforms, however, also introduce new complexities associated with composing and deploying DRE systems using components, including the need to (1) design consistent component interface definitions, (2) validate interactions between components and generate valid component deployment descriptors, (3) configure application components and the underlying middleware and platform correctly, (4) ensure that requirements of components are met by target nodes where components are deployed, and (5) validate the selected configuration and deployment satisfies end-to-end QoS requirements. The lack of simplification and automation in resolving these challenges can significantly hinder the effective transition to – and adoption of – component middleware technology to develop DRE systems.

Solution approach → Model-driven development and validation of component-based DRE systems: To address the needs of DRE system developers outlined above, we have developed an open-source model-driven development (MDD) [4] toolsuite called *Component Synthesis using Model Integrated Computing* (CoSMIC) [5].[1] CoSMIC is an integrated collection of domain-specific modeling languages (DSMLs) and generative tools that support the development, configuration, deployment, and validation of component-based DRE systems.

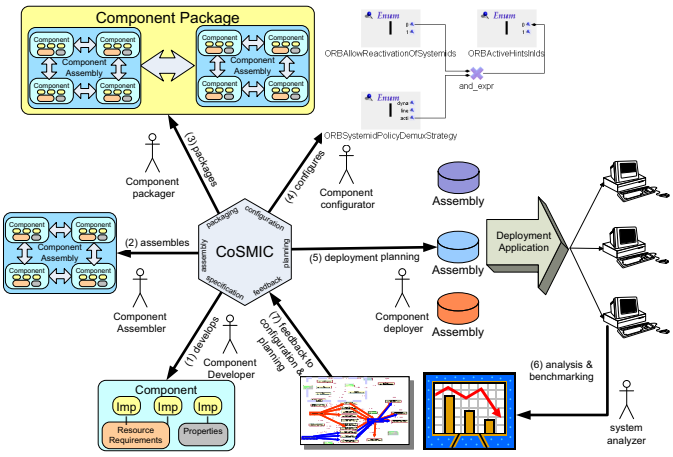

Fig. 1. The CoSMIC MDD Toolsuite

The MDD tools provided by CoSMIC address key activities in developing and validating component-based DRE systems shown in Figure 1 and described below:

- **Interface specification** – Defining and partitioning of application functionality as standalone components,
- **Interconnection specification and validation** – Composing the application from individual components by connecting them together and ensuring validity of these interactions,
- **Configuration** – Configuring the middleware with the

This work was sponsored in part by AFRL Contract#F33615-03-C-4112 for DARPA PCES Program, NSF ITR CCR-0312859, Raytheon, and a grant from Siemens CT.

The authors are with the Dept. of EECS, Vanderbilt University, Nashville, TN 37235. Contact: kitty@dre.vanderbilt.edu

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

appropriate parameters to satisfy the functional and systemic requirements of application,
- **Planning** – Making appropriate deployment decisions, including identifying the entities (such as CPUs) of the target environment where the packages will be deployed,
- **QoS assurance** – Runtime validation of desired QoS properties, as well as reconfiguration and resource management to maintain end-to-end QoS.

This paper describes how CoSMIC's integrated MDD toolsuite supports *correct-by-construction* techniques for the design, development, configuration, deployment, and validation of DRE systems by (1) providing expressive power equivalent to CORBA 3.x Interface Definition Language (IDL) in the specification of component interfaces, including import/export of IDL to/from PICML models, (2) implementing constraints on the modeling of assemblies, *e.g.*, ensuring certain ports of components are always connected, and allowing connections only between compatible ports, (3) allowing the configuration of the underlying middleware on which the artifacts will run, (4) modeling the deployment target at multiple levels of abstraction, including the hardware level (where nodes are modeled), the communication level (where the interconnects between the nodes are modeled), and the resource level (where resources used by the nodes are modeled), and (5) empirically evaluating the performance implications of configuring the underlying middleware in a particular configuration and deploying components on this middleware.

Paper organization: The remainder of this paper is organized as follows: Section II uses an avionics mission computing application as a representative example to describe common challenges associated with applying QoS-enabled component middleware to DRE systems; Section III shows how the CoSMIC DSMLs and generative tools help resolve these challenges and analyzes the results of experiments that evaluate the application of CoSMIC to our avionics application; Section IV compares our work with other MDD tools that support DRE systems; and Section V presents concluding remarks and summarizes lessons learned.

II. Challenges of Applying QoS-enabled Component Middleware to DRE Systems

Component middleware is systems software that enables reusable service components to be composed, configured, and installed to create applications rapidly and robustly. Our work in this paper focuses on the CORBA Component Model (CCM) [6], which is an OMG specification that standardizes the development, configuration, and deployment of component-based applications. CCM uses CORBA's distributed object computing (DOC) model [7] as its underlying architecture to avoid coupling application components to any particular language or platform.

In conjunction with colleagues at Washington University [8], we have developed the *Component-Integrated ACE ORB* (CIAO) [9] CCM implementation. CIAO extends our previous work on *The ACE ORB* (TAO) [10] by providing more powerful component-based abstractions using the specification, validation, packaging, configuration, and deployment techniques defined by the OMG CCM [6] and Deployment and Configuration (D&C) [11] specifications. Moreover, CIAO integrates the CCM capabilities outlined above with TAO's Real-time CORBA [10] features, such as thread-pools, lanes, and client-propagated and server-declared policies.

This section describes a CIAO-based avionics DRE application that implements the *Basic Single Processor* (*BasicSP*) scenario from the Boeing Bold Stroke component avionics mission computing project [1], which uses a *push event and pull data* publisher/subscriber communication paradigm [12] atop a QoS-enabled component middleware platform. We use the *BasicSP* example to showcase the problems encountered with developing DRE systems using component middleware and motivate the need for the CoSMIC MDD tools. This example is available in the CoSMIC and CIAO releases at `www.dre.vanderbilt.edu/cosmic` and `www.dre.vanderbilt.edu/CIAO`, respectively.

A. Applying QoS-enabled Component Middleware to a DRE Avionics System

The *BasicSP* application comprises four avionics mission computing components that periodically send GPS position updates to a pilot and navigator cockpit displays. As shown in Figure 2, a `Timer` component triggers a `GPS` navigation sensor component, which in turn publishes position information to an `Airframe` component. Upon receiving the data availability event, the `Airframe` component pulls data from the GPS, and informs a `Nav_Display` component. The `Nav_Display` component then updates the display by pulling position data from the `Airframe` component. The system requests new inputs from the GPS component at a rate of 20 Hz, and updates the display outputs with new aircraft position data at a rate of 20 Hz. The latency between the inputs to the system and the output display should be less than a single 20 Hz frame.

For the *BasicSP* scenario to satisfy the QoS requirement of ensuring display refresh rate of 20 Hz, it is necessary to examine the end-to-end critical path and configure the components appropriately. In particular, the latency between the `Airframe` and `Nav_Display` components needs to be minimized. To achieve this goal, it is necessary to determine the appropriate configurations for the individual components and then validate these configurations empirically to determine which one(s) satisfy the end-to-end QoS requirements.

Several characteristics of the *BasicSP* components are important in determining the configuration space. For example, the `Nav_Display` component receives updates only from the `Airframe` component and does not send messages back to the sender, *i.e.*, it just plays the role of a client. Likewise, the `Airframe` component communicates with both the `GPS` and the `Nav_Display` components, playing the role of a peer, though not concurrently processing requests since the events are handled sequentially.

In conjunction with colleagues at The Boeing Company [1] and Washington University [8], we have developed a prototype of the *BasicSP* application described above using the CCM and Real-time CORBA capabilities provided by CIAO [9].

The steps involved in developing this CIAO-based *BasicSP* application are described below:

1. Identify the components in the system and define their interfaces, which involves defining component ports and attributes, using the CORBA 3.x IDL features provided by CCM. In the *BasicSP* example, the control and data interfaces for the `Timer`, `GPS`, `Airframe`, `Nav_Display` components need to be specified. The control signals are published as events using a "push-based" model and the data is consumed using a "pull-based" model. This hybrid push/pull architecture is designed to minimize the execution dependencies in the Bold Stroke product-line architecture [13].

2. Define interactions between components and compose the BasicSP application, which involves keeping track of the types of each component's ports and ensuring that the interconnected components have matching ports defined. The *BasicSP* application is composed by selecting a set of component implementations from a library and generating XML deployment descriptors that contain connection information between component instances. Figure 2 shows the component interaction for the BasicSP example. This scenario begins

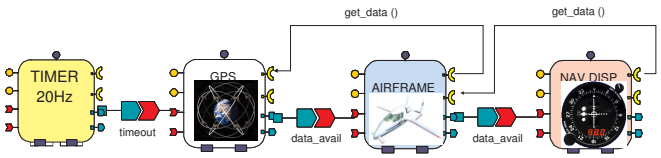

Fig. 2. The *BasicSP* Navigation Display Example

with the `GPS` being time-triggered by the TAO ORB's Real-time Event Service [14], shown as a `Timer` component. The pulse event from the `Timer` is received by the `GPS`, which is connected to the `Timer` component as an *event consumer*. The `GPS` generates its data and pushes a data available event to the `Airframe`, which is also connected as an *event consumer* to the `GPS`. TAO's Real-time Event Service then forwards the event to the `Airframe` component, which pulls the data using its *receptacle* connected to the `GPS` component, updates its state, and pushes a data available event. The Event Service forwards the event to the `Nav_Display` component, which in turn pulls the data through its *receptacle* from the `GPS`, updates its state, and displays it.

3. Configure the QoS-enabled component middleware where the BasicSP components will run, which involves configuring each of the `Timer`, `Nav_Display`, `GPS`, and `Airframe` components. To ensure the refresh rate of 20 Hz required by the applications, it is necessary to configure the middleware suitably. The problem, however, is that QoS-enabled middleware, such as CIAO, provides scores of configuration options, so choosing the right set from these options can be a daunting task. This problem is exacerbated by the fact that not all combinations of options form a semantically compatible set.

4. Deploy the BasicSP application onto its runtime platform, which involves ensuring that the implementation artifacts and the associated deployment descriptors are available on the actual target platform, and initiating the deployment process using the standard OMG D&C [11] framework and tools. In the *BasicSP* example, this activity involves using hand-written XML descriptors to deploy the application.

5. Empirically validate the configuration and deployment, which involves ensuring that the chosen component deployment and middleware configurations satisfy the system QoS. In the *BasicSP* scenario, this activity involves ensuring that all timer updates provided by the `GPS` component and delivered to the `Nav_Display` component in a lossless manner. In general, addressing the QoS validation challenges of DRE systems fielded in a particular environment requires a suite of benchmarking tests that are customized to the system's requirements and environment.

B. Challenges of Developing the BasicSP Application using QoS-enabled Component Middleware

The remainder of this section describes five common challenges that arises when applications, such as *BasicSP* and our earlier work on unmanned air vehicle (UAV) applications [15], are developed using QoS-enabled CCM in the absence of MDD tool support.

1) Alleviating Complexities in Component Interface Definition: The CORBA 3.x IDL used for CCM defines extensions to the syntax and semantics of CORBA 2.x IDL. Developers of CCM-based applications must therefore master the differences between CORBA 2.x IDL and CORBA 3.x IDL. For example, while CORBA 2.x interfaces can have multiple inheritance, CCM components can have only a single parent, so equivalent units of composition (*i.e.*, interfaces in CORBA 2.x and components in CCM) can have subtle semantic differences. Moreover, any component interface that needs to be accessed by component-unaware CORBA clients should be defined as a *supported* interface as opposed to a *provided* interface.

In any project that transitions from an object-oriented architecture to a component-based architecture, there is a likelihood of simultaneous existence of simple CORBA objects and more sophisticated CCM components. Design of component interfaces must therefore be done carefully. For example, the `Airframe` components receives events from the `GPS` component and sends events to the `Nav_Display` component. Since events are point-to-point, the `Airframe` component should use *emits* rather than *publishes*.

2) Validating Component Interactions and Deployment Descriptors: Even for DRE system developers well-versed in CORBA 3.x IDL, it is hard to keep track of components and their types using text-based IDL files, which provide no visual feedback to identify differences between components. Moreover, an IDL compiler will not be able to catch mismatches in the port types of two components that are connected together since component connection information is not defined in IDL. In addition to ensuring type compatibility between interconnected component types as part of interface definition, component developers must also ensure the same compatibility between instances of these component types in the XML descriptor files needed for deployment. Ensuring this compatibility is more problematic than ensuring type compatibility since the number of *component instances* typically

dwarfs the number of *component types* in large-scale DRE systems.

A CCM assembly file, which is an XML file illustrating component interconnections, is not well-suited to manual editing. In addition to learning IDL, DRE system developers must also learn XML to compose component-based DRE systems. In our *BasicSP* example, simply increasing the number of processors, or introducing sensors that operate at multiple rates increases the number of component instances and hence the component interconnections. The increase in component interconnections often grows faster than the number of component instances. Any errors in this step are therefore likely to go undetected until the deployment of the system at run-time.

3) Ensuring Suitable Middleware Configurations: QoS-enabled component middleware provides a range of configuration options that can be used to customize and tune the QoS properties of the middleware. For example, the CIAO QoS-enabled middleware provides ∼500 configuration options that can be used to tune its behavior [16]. DRE system developers need to configure and tune the performance of CIAO at multiple levels, including lower-level messaging and transport mechanisms, the object request broker (ORB) itself, up to higher-level middleware services (such as event notification, scheduling, and load balancing). This problem is exacerbated by the fact that not all combinations of options form a semantically compatible set.

In our *BasicSP* application scenario, components playing role of a client `Nav_Display` do not require locking and concurrency. Similarly, the `Airframe` component can disable locking for both its client and server side roles. Fine-tuning middleware performance is key to ensuring application level QoS is met. Errors in configuration can manifest in different forms, such as run-time crashes, undefined behavior, or in decreased QoS.

4) Associating Components with the Proper Deployment Target: In component-based systems there is often a disconnect between software implementation-related activities and the actual target system since (1) the software artifacts and the physical system are developed independently and (2) there is no way to associate these two entities using standard component middleware features. This disconnect can result in run-time failures due to the target environment's lack of capabilities to support the deployed component's requirements. These mismatches can also often be a source of missed optimization opportunities since knowledge of the target platform can help optimize component implementations and customize the middleware accordingly.

In our *BasicSP* application, the components can all reside on a single processor and hence can use collocation facilities provided by ORBs to eliminate unnecessary (de)marshaling. Without the ability to associate components with targets, errors due to incompatible component connections and incorrect XML descriptors are likely to show up only during system deployment at run-time.

5) Validating the QoS of the Selected Configuration and Deployment: QoS-enabled component middleware runs on a multitude of hardware, OS, and compiler platforms. DRE system developers often use trial-and-error methods to select the set of configuration options that maximizes the QoS attainable by the middleware. Unfortunately, the settings that maximize performance for a particular group of platforms and applications may not be suitable for different ones. QoS validation of the end system requires rigorous validation of the deployment plan and middleware configuration. This process historically involves tedious and error-prone handcrafting of low-level source, benchmarking, build, and script files [16].

An example benchmarking test for the *BasicSP* application could validate the configurations of the `Nav_Display` component. Similarly, measuring end-to-end latency via associating timers from when the `GPS` component sends out its position update to when `Nav_Display` component receives it, measures how configuration of all components affects end-to-end latency. Tuning the components and middleware to maximize QoS is hard without the ability to add timing information and measure different metrics, such as latency, throughput, or jitter.

C. Summary of Challenges

A common theme underlying the evaluation of component middleware challenges above is that errors and sources of performance overhead often go undetected until late in the development cycle. Even when these errors and overheads are eventually detected, fixing them often involves backtracking to multiple prior lifecycle steps, which impedes productivity and increases the level of effort. Without support from higher-level tools and techniques, therefore, the advantages of transitioning from DOC middleware to component middleware can be reduced significantly. These observations underscore the importance of enhancing design-time MDD tool support for DRE systems built using component middleware, which includes the importance of automating the configuration, validation, and deployment of DRE systems.

III. Resolving BasicSP Application Challenges with CoSMIC

This section examines how features of CoSMIC's DSMLs and generative tools can be applied to address the challenges discussed in Section II-B that arise when developing QoS-enabled component middleware-based DRE systems, such as our *BasicSP* application, without MDD tool support. Figure 3 describes the sequence of activities involved in using the CoSMIC MDD toolsuite to develop component-based DRE applications. At the heart of CoSMIC is the **Platform-Independent Component Modeling Language** (PICML). PICML is a DSML that provides capabilities to handle complex component engineering tasks, such as component interface specification, validation of component interactions, deployment planning, and composition of application using hierarchical component assemblies. PICML assists component developers with the *packaging* and *planning* phases shown in Figure 1.

Another important DSML in CoSMIC is the **Options Configuration Modeling Language** (OCML), which simplifies the specification and validation of complex DRE middleware and application configurations. OCML provides (1) generic modeling elements (such as numeric, boolean and string

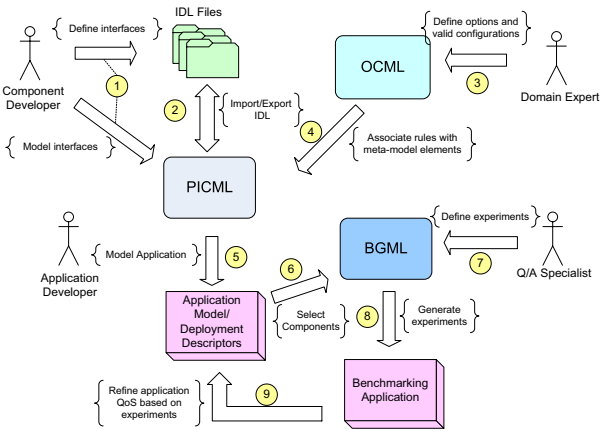

Fig. 3. Domain-Specific Modeling Languages in CoSMIC

options) to represent different middleware options and (2) generic constraint elements (such as rules) to capture option dependencies. OCML assists component developers in the *configuration* phase of developing component-based systems.

The **Benchmark Generation Modeling Language** (BGML) is a third DSML provided by CoSMIC to synthesize benchmarking testsuites that analyze the QoS performance of OCML-configured DRE systems. BGML can be used in the *configuration* phase to validate PICML assemblies and OCML configurations. It can also be used in the *planning* phase to validate 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.

A. Visual Component Interface Definition

A set of component, interface, and other datatype definitions may be created in PICML using either of the following two approaches shown in steps 1 and 2 of Figure 3:

- **Adding to existing definitions imported from IDL**. In this approach, existing CORBA software systems can be easily migrated to PICML using its *IDL Importer*, which takes any number of CORBA IDL files as input, maps their contents to the appropriate PICML model elements, and generates a single XML file that can be imported as a PICML model. This model can then be used as a starting point for modeling assemblies and generating deployment descriptors.
- **Creating IDL definitions from scratch**. In this approach, PICML's graphical modeling environment provides support for designing the interfaces using an intuitive "drag and drop" technique, making this process largely self-explanatory and independent of platform-specific technical knowledge. Most of the grammatical details are implicit in the visual language, *e.g.*, when the model editor screen is showing the "scope" of a definition, only icons representing legal members of that scope will be available for dragging and dropping. CORBA IDL can be generated from PICML enabling generation of software artifacts in languages having a CORBA IDL mapping.

Application to the BasicSP example scenario. By modeling the *BasicSP* components using PICML, the problems associated with multiple inheritance and semantics of IDL are flagged at design time. By providing a visual environment for defining the interfaces, PICML therefore resolves many problems described in Section II-B.1 associated with definition of component interfaces. In particular, by modeling the interface definitions, PICML alleviates the need to model a subset of interfaces for analysis purposes, which has the added advantage of preventing skew between the models of interfaces used by analysis tools and the interface used in implementations.

B. Valid component interaction definition and descriptor generation

PICML defines the *static semantics* of a system using a constraint language and enforces these semantics early in the development cycle, *i.e.*, at design-time. Static semantics refer to the "well-formedness" rules of the language. By elevating the level of abstraction via MDD techniques, however, the corresponding well-formedness rules of DSMLs like PICML actually capture semantic information, such as constraints on composition of models, and constraints on allowed interactions.

There is a significant difference in the early detection of errors in the MDD paradigm compared with traditional object-oriented or procedural development using a conventional programming language compiler. In PICML, OCL constraints are used to define the static semantics of the modeling language, thereby disallowing invalid systems to be built using PICML, *i.e.*, PICML enforces the correct-by-construction paradigm.

Application to the BasicSP example scenario. In the context of our *BasicSP* application all the components can be modeled as a CCM assembly as shown in step 5 of Figure 3. PICML enables the visual inspection of types of ports of components and the connection between compatible ports, including flagging error when attempting connection between incompatible ports. PICML also differentiates different types of connections using visual cues, such as dotted lines and color, to quickly compare the structure of an assembly. By providing a visual environment coupled with rules defining valid constructs, PICML therefore resolves many problems described in Section II-B.2 with ensuring consistent component interactions. By enforcing the constraints during creation of component models and interconnections – and by disallowing connections to be made between incompatible ports – PICML completely eliminates the manual effort required to perform these kinds of checks.

In addition to ensuring design-time integrity of systems built using OCL constraints, PICML also generates the complete set of deployment descriptors that are needed as input to the component deployment mechanisms. Since the rules determining valid assemblies are encoded into PICML via its metamodel and enforced using constraints, PICML ensures that the generated XML describes a valid system. Generation

of XML is done in a programmatic fashion by writing a `Visitor` class that uses the Visitor pattern [17] to traverse the elements of the model and generate XML. The generated XML descriptors also ensure that the names associated with instances are unique so that individual component instances can be identified unambiguously at run-time.

C. Configuration of QoS-enabled Component Middleware

The first step in ensuring end-to-end QoS is to configure the component and the underlying middleware and platform appropriately. Component configuration is done via associating `Property` and `Requirement` elements with the components. To ensure both syntactic and semantically correct configuration of underlying middleware, PICML leverages the OCML tool. OCML defines *a type system* that middleware developers use to specify their platform-specific (in our case middleware) option space.

As shown in step 3 of Figure 3, *standard configurations* can be modeled via OCML and application developers can use these configurations specify the set of options that suit their components. It also enforces *dependency rules* that prevent application developers from choosing invalid combinations of configuration sets that could result in incorrect/undefined behavior. A *Configuration File Generator* (CFG) application is developed for application developers to specify the appropriate set of option configurations and validation of this set.

OCML has been applied to represent the options and dependencies of the CIAO QoS-enabled component middleware.[2] As shown in step 4 of Figure 3, PICML leverages OCML in the following manner:

- The CFG for CIAO has been associated with `ImplementationArtifact` elements in the PICML model (these artifacts represent entities that are deployed onto nodes, such as DLLs and shared objects). The CFG populates the configuration attribute of each artifact based on the options chosen by application developers.
- The OCML configuration exporter reads the information in the configuration attribute and generates a `svc.conf` file, which captures middleware configuration parameters. The deployment infrastructure parses this file to configure the middleware appropriately.

Application to the BasicSP example scenario. In the context of the *BasicSP* scenario, OCML can be used to configure the underlying middleware on which the components are run via the two step process described earlier. Figure 4 illustrates the CFG wizard that application developers interact with to configure the `Airframe` component in the *BasicSP* scenario. In determining the appropriate configuration of the middleware hosting this component, we require that this component interacts with both the `GPS` and `Nav_Display` components. This precludes the need for concurrency and locking within this component. Using this information, application developers select the middleware configuration. The CFG also displays the documentation for each option.

[2]The OCML options as applied to CIAO are available from $COSMIC_ROOT/PSM/CCM/OCML directory in the CoSMIC distribution.

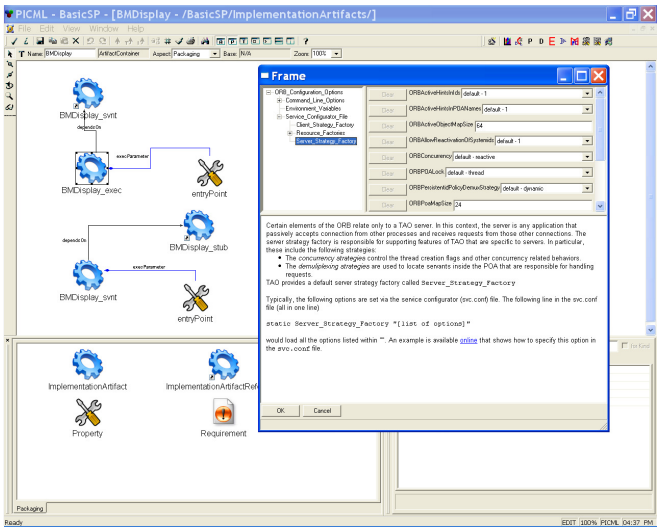

Fig. 4. Middleware Configuration via OCML

OCML tool alleviates complexities associated with configuring component middleware by ensuring both (1) *syntactic correctness*, *i.e.*, options reflecting actual middleware setting and (2) *semantic validity*, *i.e.*, eliminating inconsistent options. OCML eliminates incorrect and incompatible configurations thus preventing undefined and incorrect behavior at run-time. OCML however, requires application developers to be familiar with the ORB configurations though the documentation is provided to guide the configuration process.

D. Deployment Planning

To satisfy multiple QoS requirements, DRE systems are often deployed in heterogeneous execution environments. To support such environments, component middleware strives to be largely independent of the specific target environment in which application components will be deployed.

PICML can be used to specify the target environment where the DRE system will be deployed, which includes defining: (1) *nodes*, where the individual components and component packages are loaded and used to instantiate those components, (2) *interconnects* among nodes to which inter-component software connections are mapped to allow the instantiated components to communicate, and (3) *bridges* among interconnects, where interconnects provide a direct connection between nodes and bridges to provide routing capability between interconnects. Nodes, interconnects, and bridges collectively represent the target environment.

Once the target environment is specified via PICML, allocation of component instances onto nodes of the target target environment can be performed. This activity is referred to as *component placement*, where systemic requirements of the components are matched with capabilities of the target environment and suitable allocation decisions are made. Allocation can either be: (1) *static*, where the domain experts know the functional and QoS requirement of each of the components, as well as knowledge about the nodes of the target environment. In such a case, the job of the allocation is to create a deployment plan comprising the components→node

mapping specified by the domain expert, or (2) *dynamic*, where the domain expert specifies the constraints on allocation of resources at each node of the target environment, and the job of the allocation is to choose a suitable component→node mapping that meets both the functional and QoS requirement of each of the components, as well as the constraints on the allocation of resources.

PICML currently provides facilities for specifying static allocation of components. Domain experts can visually map the components with the respective target nodes, as well as provide additional hints, such as whether the components need to be process-collocated or host-collocated, provided two components are deployed in the same target node. PICML generates a deployment plan from this information, which is used by the CIAO run-time deployment engine to perform the actual deployment of components to nodes.

Application to the BasicSP example scenario. In the context of the *BasicSP* example, PICML can be used to specify the mapping between the component and the target environment. By modeling the target environment in the *BasicSP* example using PICML, therefore, the problem with a disconnect between components and the deployment target described in Section II-B.4 can be resolved. In case there are multiple possible component→node mappings, PICML can be used to experiment with different combinations since it generates descriptors automatically. For example, `GPS` components might be mapped onto different types of navigation sensors on the aircraft, each with different QoS properties until the desired QoS is achieved. PICML thus eliminates the low-level, tedious, and error-prone manual effort involved in writing a deployment plan.

E. Validating System QoS

Addressing the QoS validation challenges of DRE systems fielded in a particular environment requires a suite of benchmarking tests that are customized to the system's environment. Moreover, these tests can also help validate the deployment plan, *i.e.*, provide a baseline or an estimate of system performance such as end-to-end latency/throughput. This will help the planner to change the component node mappings in order to maximize QoS.

To facilitate QoS validation, PICML leverages the BGML tool (as shown in step 6 of Figure 3), which captures key QoS validation concerns of QoS-enabled middleware, such as (1) modeling how distributed system components interact with each other and (2) representing metrics that can be applied to specific configuration options and platforms. BGML allows QA specialists to create benchmarking experiments by synthesizing: (1) the header files and source code that implement the functionality, (2) the configuration and script files that tune the underlying middleware and automate the task of running tests and output generation, and (3) project build files (*e.g.*, makefiles) required to generate the executable code.

In particular, BGML has been integrated with PICML as a separate paradigm accessible from within PICML. For the purpose of QoS validation, 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-set can also be used to model and simulate background load present during the experimentation process. These workload elements are then mapped to individual platform specific code in the interpretation process as shown in step 8 of Figure 3.

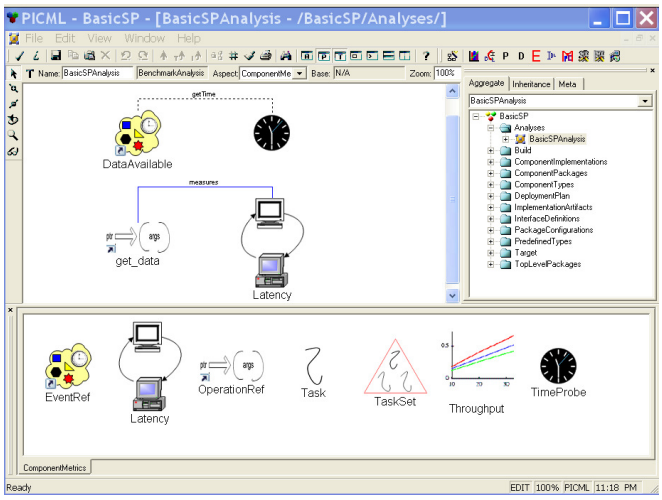

Fig. 5. QoS Validation via BGML

Application to the BasicSP example scenario. In the context of the *BasicSP* example, BGML can be used to measure QoS (end-to-end latency) between when the `GPS` component publishes a position update and when the `Nav_Display` component displays the image to the pilot. Below we describe how BGML in concert with OCML and PICML can be used to validate QoS for a chosen deployment plan and middleware configuration.

We first used the PICML MDD tool to model the BasicSP DRE system scenario. This step first involved importing the BasicSP IDL files, defining component types, and modeling their interactions and finally modeling the deployment artifacts. To understand the impact of configuration on QoS, we wanted to observe how the possible configurations for the `Nav_Display` component affects the end-to-end QoS. The next step in achieving our goal was to use the OCML tool to generate the possible configurations for the component under test.[3] Using the documentation generated by OCML CFG, we narrowed down the configuration space for the `Nav_Display` component: Further examination of this reduced configuration space reveals that some of the configurations settings can be set *a priori*, *i.e.*, without experimentation. For example, the component interacts with only one source and do not need synchronization. These option settings (options A-F) can be directly determined (shown in bold) in Table I. For the remaining configurations, where both options are suitable, the possible configuration combinations were generated using OCML.

[3]For the remaining components, we used the default CIAO middleware configuration.

Option Label	Option Name	Option Settings
A	ORBReactorMaskSignals	{**0**, 1}
B	ORBInputCDRAllocator	{ **null**, thread}
C	ORBReactorType	{ **select_st**, mt}
D	ORBProfileLock	{thread, **null**}
E	ORBObjectLock	{thread, **null**}
F	ORBConnectionCacheLock	{**null**, thread}
G	ORBClientConnectionHandler	{RW, ST}
H	ORBTransportMuxStrategy	{EXCLUSIVE, MUXED}
I	ORBFlushingStrategy	{LF, reactive}
J	ORBConnectStrategy	{LF, reactive}

TABLE I
CONFIGURATION SPACE FOR THE `Nav_Display` COMPONENT

Using the target modeling capabilities of PICML, we created a deployment scenario as described in Table II.

Hosts	DOC	ACE	TANGO
CPU Type	AMD Athlon	AMD Athlon	Intel Xeon
CPU Speed (GHz)	2	2	1.9
Memory (GB)	1	1	2
Cache (KB)	512	512	2048
Compiler (gcc)	3.2.2	3.3	3.3.2
OS (Linux)	Red Hat 9	Red Hat 8	Debian
Kernel	2.4.20	2.4.20	2.4.23
`Avionics`	`NavDisplay`	`Airframe`	`GPS`

TABLE II
TESTBED AND DEPLOYMENT SUMMARY

Finally, using the BGML tool, as shown in Figure 5, we associate Timer values to capture latency between the `GPS` and `Nav_Display` components to measure end-to-end QoS. Additionally, the build files required to compile the benchmarking code with the BasicSP source libraries were synthesized using BGML model interpreters. Table III

Setting	Latency (μsecs)
(G1, H1, I2, J2)	504
(G1, H1, I2, J1)	528
(G1, H1, I1, J2)	529
(G1, H1, I1, J1)	532
(G1, H2, I1, J1)	536
(G1, H2, I2, J1)	548
(G1, H2, I1, J2)	552
(G1, H2, I2, J1)	562
(G2, H1, I2, J2)	568
(.....)	

TABLE III
LATENCY QOS DISTRIBUTION FOR THE `Nav_Display` COMPONENTS

tabulates the latency distributions for the client-side display based components. We use the notation *A1*, *B2*, etc. to identify the individual options within each category. For example, the `-ORBConnectStrategy` value of `LF` is denoted as *J1*.

The top 8 configurations (out of a possible 16) are shown in Table III sorted by increasing order of latency values. A closer look at the values reveals a clear pattern of configuration options and its effect on QoS (end-to-end) latency. For example, the option G1 has the greatest effect on performance, *i.e.*, changing its value to G2 increases latency by $\sim$4μsecs for the *robot assembly* scenario and by $\sim$50μsecs in the *Basic_SP* scenario. After G, the option H influences latency the most, *i.e.*, changing its value from H1 to H2 worsens latency by $\sim$2μsecs in the first case and by $\sim$30μsecs in the second case. Using our PICML, OCML and BGML tools, therefore, we determined the appropriate configuration for the `Nav_Display` component to be:

```
static Advanced_Resource_Factory
  "-ORBReactorMaskSignals 0 -ORBInputCDRAllocator null
   -ORBReactorType select_st -ORBConnectionCacheLock null"

static Client_Strategy_Factory
  "-ORBTransportMuxStrategy EXCLUSIVE -ORBProfileLock null
   -ORBClientConnectionHandler RW
   -ORBFlushingStrategy reactive
   -ORBConnectStrategy reactive"
```

These configuration values (1) disable locking (`ORB-ProfileLock`, `ORBInputCDRAllocator`), (2) set the ORB's concurrency mechanism to single-threaded reactive (`ORBReactorType`, `ORBConnectStrategy`, and `ORB-FlushingStrategy`), and (3) use an exclusive connection for request demultiplexing (`ORBTransportMux-Strategy`).

F. Summary of Results

The results presented in this section show how CoSMIC provides many benefits that overcome key limitations with conventional QoS-enabled component middleware described in Section II by generating software artifacts that are *correct by construction*. In particular, it generates IDL files that are guaranteed to compile, thereby eliminating errors in the component definitions. CoSMIC also disallows creation of semantically inconsistent models, and generates deployment descriptors that are guaranteed to be syntactically valid. In addition, CoSMIC ensures that component interactions are guaranteed to have type compatibility. The generated configuration files are both syntactically and semantically correct, thereby ensuring the underlying middleware is configured appropriately. The generated deployment plans eliminate impossible component→node allocations. CoSMIC also supports the hierarchical composition of components without any extra overhead associated with each layer in the hierarchy. Finally, it provides mechanisms for QA specialists to evaluate how the configuration, platform and the deployment plans influence application QoS thereby allowing refining and regeneration of plans that can meet end-to-end QoS requirements.

IV. 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.

a) **Cadena:** Cadena [18] 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 [19] with KSU to integrate PICML with Cadena's model checking tools, so we can accelerate the development and verification of DRE systems.

b) **VEST and AIRES:** The *Virginia Embedded Systems Toolkit* (VEST) [20] and the *Automatic Integration of Reusable Embedded Systems* (AIRES) [21] 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.

c) **ESML:** The *Embedded Systems Modeling Language* (ESML) [22] was developed by ISIS at Vanderbilt University to provide a visual metamodeling language 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. Using these analyses, design decisions (such as component allocations to the target execution platform) can be performed. Unlike PICML, ESML is platform-specific since it is customized for the Boeing Bold Stroke PRiSm QoS-enabled component model [1], [23]. ESML also does not support nested assemblies and the allocation of components are tied to processor boards, which is a proprietary feature of the Bold Stroke 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.

V. 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 approach to resolving these complexities is model-driven development (MDD) [4]. 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, and ICE.

This paper describes the capabilities of the CoSMIC MDD toolsuite developed at Vanderbilt University. To showcase how CoSMIC helps resolve key complexities of QoS-enabled component middleware, we applied several of its domain-specific modeling languages (DSMLs) to the *Basic Single Processor* (BasicSP) scenario from the Boeing Bold Stroke component avionics mission computing product suite [1]. Using the *BasicSP* 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, and
- **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 [18], VEST [20], and AIRES [21]) and the actual deployed and validated component implementations [19].

The following are a summary of lessons learned based on our experience developing and evaluating CoSMIC:

1. Component and platform modeling improves DRE systems reasoning: The results of applying CoSMIC to the component-based *BasicSP* application example show how it enables the comprehension of the system at a higher level of abstraction relative to conventional DOC middleware approaches. In particular, CoSMIC's DSML-based approach reduces the effort involved in component interface definition by $\sim$50%, eliminates common errors in defining component interactions, and generation of deployment descriptors including deployment plan completely, and eliminates the duplication of components with similar functionality by allowing reuse through hierarchical composition.

2. Early detection of errors improves productivity significantly: Most of the errors that CoSMIC eliminates at design- and deployment-time are discovered only at run-time with conventional component development techniques, due to a combination of complexities in development of components, coupled with *out-of-band* specification (using XML) of component interconnections. This finding underscores the importance of CoSMIC's MDD approach, which helps increase the effectiveness of applying QoS-enabled component middleware technologies to the DRE systems domain.

3. Addressing ad hoc approaches of configuration: CoSMIC's MDD tools and process can (1) alleviate key complexities involved in understanding the impact of middleware configurations on application QoS and (2) bring rigor to otherwise *ad hoc* processes used by developers to configure and deploy middleware for DRE systems.

4. End-to-end toolchains for DRE systems need to bridge analysis and empirical results: Though CoSMIC provides developers of DRE system with many capabilities for specifying component-based systems and their interactions, it is not

yet a complete end-to-end solution for model-based design and analysis of component-based DRE systems. Our future work will therefore focus on dynamic component allocation, performance analysis of component systems by empirically evaluating component interactions with respect to various performance metrics, and performance modeling of components with regard to meeting real-time deadlines. We are developing model-based solutions for these problems and integrating the resulting tools into the CoSMIC toolsuite.

5. Need for process automation: A limitation of our current MDD approach is that the experiments we ran required considerable human intervention, *e.g.*, changing the node component association in the models and re-running the model interpreters to generate the XML metadata. This level of effort is not an inherent limitation of our process, however, but rather a limitation of our modeling tool infrastructure, which does not yet support scripting of models and model-interpreters. Our future work will therefore address this limitation by collaborating with the tool developers to make it more scriptable.

REFERENCES

[1] D. C. Sharp and W. C. Roll, "Model-Based Integration of Reusable Component-Based Avionics System," in *Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

[2] P. Sharma, J. Loyall, G. Heineman, R. Schantz, R. Shapiro, and G. Duzan, "Component-Based Dynamic QoS Adaptations in Distributed Real-Time and Embedded Systems," in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, Agia Napa, Cyprus, Oct. 2004, pp. 1208–1224.

[3] R. Noseworthy, "IKE 2 – Implementing the Stateful Distributed Object Paradigm ," in *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*. Washington, DC: IEEE, Apr. 2002.

[4] J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. New York: John Wiley & Sons, 2004.

[5] Institute for Software Integrated Systems, "Component Synthesis using Model Integrated Computing (CoSMIC)," www.dre.vanderbilt.edu/cosmic, Vanderbilt University.

[6] *CORBA Components*, OMG Document formal/2002-06-65 ed., Object Management Group, June 2002.

[7] *The Common Object Request Broker: Architecture and Specification*, 3.0.2 ed., Object Management Group, Dec. 2002.

[8] N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian, "Configuring Real-time Aspects in Component Middleware," in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, Agia Napa, Cyprus, Oct. 2004, pp. 1520–1537.

[9] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, and C. D. Gill, "QoS-enabled Middleware," in *Middleware for Communications*, Q. Mahmoud, Ed. New York: Wiley and Sons, 2003, pp. 131–162.

[10] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

[11] *Deployment and Configuration Adopted Submission*, OMG Document ptc/03-07-08 ed., Object Management Group, July 2003.

[12] D. C. Sharp, "Avionics Product Line Software Architecture Flow Policies," in *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Oct. 1999.

[13] B. S. Doerr and D. C. Sharp, "Freeing Product Line Architectures from Execution Dependencies," in *Proceedings of the 11th Annual Software Technology Conference*, Apr. 1999.

[14] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*. Atlanta, GA: ACM, Oct. 1997, pp. 184–199.

[15] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems," in *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Francisco, CA, Mar. 2005, pp. 190–199.

[16] C. Yilmaz, A. Krishna, A. Memon, A. Porter, D. C. Schmidt, A. Gokhale, and B. Natarajan, "Main Effects Screening: A Distributed Continuous Quality Assurance Process for Monitoring Performance Degradation in Evolving Software Systems," in *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, St. Louis, MO, May 2005.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[18] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems," in *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, May 2003.

[19] G. Trombetti, A. Gokhale, D. C. Schmidt, J. Hatcliff, G. Singh, and J. Greenwald, "A Integrated Model-driven Development Environment for Composing and Validating Distributed Real-time and Embedded Systems," in *Model Driven Software Development- Volume II of Research and Practice in Software Engineering*, S. Beydeda, M. Book, and V. Gruhn, Eds. New York: Springer-Verlag, 2005.

[20] J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis, "VEST: An Aspect-based Composition Tool for Real-time Systems," in *Proceedings of the IEEE Real-time Applications Symposium*. Washington, DC: IEEE, May 2003.

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

[22] G. Karsai, S. Neema, B. Abbott, and D. Sharp, "A Modeling Language and Its Supporting Tools for Avionics Systems," in *Proceedings of 21st Digital Avionics Systems Conf.*, Aug. 2002.

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