Model Driven Middleware: A New Paradigm for Developing Distributed Real-time and Embedded Systems ⋆

Aniruddha Gokhale [a,*], Krishnakumar Balasubramanian [a], Arvind S. Krishna [a], Jaiganesh Balasubramanian [a], George Edwards [a], Gan Deng [a], Emre Turkay [a], Jeffrey Parsons [a], Douglas C. Schmidt [a]

[a] *Institute for Software Integrated Systems, Vanderbilt University, Campus Box 1829 Station B, Nashville, TN 37235, USA*

Abstract

Distributed real-time and embedded (DRE) systems have become critical in domains such as avionics (*e.g.*, flight mission computers), telecommunications (*e.g.*, wireless phone services), tele-medicine (*e.g.*, robotic surgery), and defense applications (*e.g.*, total ship computing environments). These types of systems are increasingly interconnected via wireless and wireline networks to form systems of systems. A challenging requirement for these DRE systems involves supporting a diverse set of quality of service (QoS) properties, such as predictable latency/jitter, throughput guarantees, scalability, 24x7 availability, dependability, and security that must be satisfied simultaneously in real-time. Although increasing portions of DRE systems are based on QoS-enabled commercial-off-the-shelf (COTS) hardware and software components, the complexity of managing long lifecycles (often ∼15-30 years) remains a key challenge for DRE developers and system integrators. For example, substantial time and effort is spent retrofitting DRE applications when the underlying COTS technology infrastructure changes.

This paper provides two contributions that help improve the development, validation, and integration of DRE systems throughout their lifecycles. First, we illustrate the challenges in creating and deploying QoS-enabled component middleware-based DRE applications and describe our approach to resolving these challenges based on a new software paradigm called Model Driven Middleware (MDM), which combines model-based software development techniques with QoS-enabled component middleware to address key challenges faced by developers of DRE systems - particularly composition, integration, and assured QoS for end-to-end operations. Second, we describe the structure and functionality of CoSMIC (Component Synthesis using Model Integrated Computing), which is an MDM toolsuite that addresses key DRE application and middleware lifecycle challenges, including developing component functionality, partitioning the components to use distributed resources effectively,

validating the software configurations, assuring multiple simultaneous QoS properties in real-time, and safeguarding against rapidly changing technology.

Key words: MDA: Model Driven Architecture, MDM: Model Driven Middleware, MIC: Model Integrated Computing, CCM: CORBA Component Model, D&C: Deployment and Configuration

---

1 Introduction

1.1 Emerging Trends

Computing and communication resources are increasingly used to control mission-critical, large-scale distributed real-time and embedded (DRE) systems. Figure 1 illustrates a representative sampling of DRE systems in the medical imaging, commercial air traffic control, military combat operational capability, electrical power grid system, and industrial process control domains. These types of DRE systems share the following characteristics:

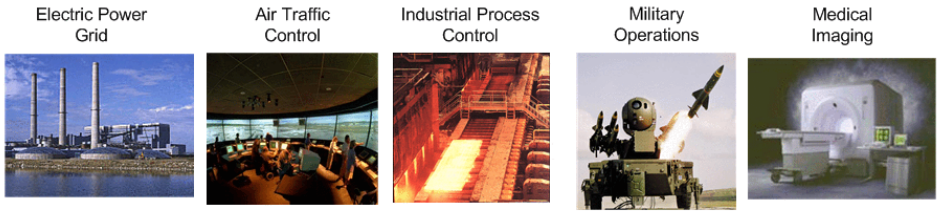

Fig. 1. **Example Large-scale Distributed Real-time and Embedded Systems**

1. Heterogeneity. Large-scale DRE systems often run on a variety of computing platforms that are interconnected by different types of networking technologies with varying QoS properties. The efficiency and predictability of DRE systems built using different infrastructure components varies according to the type of computing platform and interconnection technology.

2. Deeply embedded properties. DRE systems are frequently composed of multiple embedded subsystems. For example, an anti-lock braking software control system forms a resource-constrained subsystem that is part of a larger DRE application controlling the overall operation of an automobile.

3. Simultaneous support for multiple quality of service (QoS) properties. DRE software controllers [1] are increasingly replacing mechanical and human control of critical systems. These controllers must simultaneously

⋆ Work supported by AFRL Contract#F33615-03-C-4112 for DARPA PCES Program
* Corresponding Author Email: a.gokhale@vanderbilt.edu

support many challenging QoS constraints, including (1) *real-time requirements*, such as low latency and bounded jitter, (2) *availability requirements*, such as fault propagation/recovery across distribution boundaries, (3) *security requirements*, such as appropriate authentication and authorization, and (4) *physical requirements*, such as limited weight, power consumption, and memory footprint. For example, a distributed patient monitoring system requires predictable, reliable, and secure monitoring of patient health data that can be distributed in a timely manner to healthcare providers.

4. Large-scale, network-centric operation. The scale and complexity of DRE systems makes it infeasible to deploy them in disconnected, standalone configurations. The functionality of DRE systems is therefore partitioned and distributed over a range of networks. For example, an urban bio-terrorist evacuation capability requires highly distributed functionality involving networks connecting command and control centers with bio-sensors that collect data from police, hospitals, and urban traffic management systems.

5. Dynamic operating conditions. Operating conditions for large-scale DRE systems can change dynamically, resulting in the need for appropriate adaptation and resource management strategies for continued successful system operation. In civilian contexts, for instance, power outages underscore the need to detect failures in a timely manner and adapt in real-time to maintain mission-critical power grid operations. In military contexts, likewise, a mission mode change or loss of functionality due to an attack in combat operations requires adaptation and resource reallocation to continue with mission-critical capabilities.

1.2 Technology Challenges and Solution Approaches

Although the importance of the DRE systems described above has grown significantly, software for these types of systems remains considerably harder to develop, maintain, and evolve [2,3] than mainstream desktop and enterprise software. A significant part of the difficulty stems from the historical reliance of DRE systems on proprietary hardware and software technologies and development techniques. Unfortunately, proprietary solutions often fail to address the needs of large-scale DRE systems over their extended lifecycles. For instance, as DRE systems grow in size and complexity the use of proprietary technologies can make it hard to adapt DRE software to meet new functional or QoS requirements, hardware/software technology innovations, or emerging market opportunities.

During the past decade, a substantial amount of R&D effort has focused on developing standards-based *middleware*, such as Real-time CORBA [4] and QoS-enabled CORBA Component Model (CCM) middleware [5], to address the challenges outlined in the previous paragraph. As shown in Figure 2, middleware is systems software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware and provides the following capabilities:

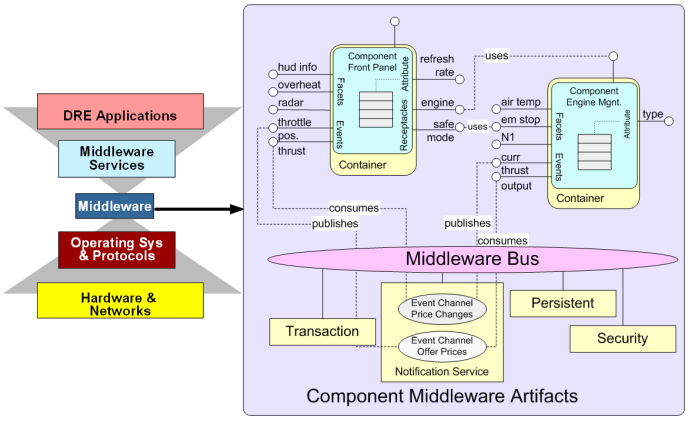

Fig. 2. **Component Middleware Layers and Architecture**

1. Control over key end-to-end QoS properties. A hallmark of DRE systems is their need to control the end-to-end scheduling and execution of CPU, network, and memory resources. QoS-enabled component middleware is based on the expectation that QoS properties will be developed, configured, monitored, managed, and controlled by a different set of specialists (such as middleware developers, systems engineers, and administrators) than those responsible for programming the application functionality in traditional DRE systems.

2. Isolation of DRE applications from heterogeneous operating systems and networks. Standards-based QoS-enabled component middleware defines *communication mechanisms* that can be implemented over many networks and OS platforms. Component middleware also supports *containers* that (a) provide a common operating environment to execute a set of related components and (b) shield the components from the underlying networks, operating systems, and even the underlying middleware implementations. By reusing the middleware's communication mechanisms and containers, developers of DRE systems can concentrate on the application-specific aspects of their systems and leave the communication and QoS-related details to middleware developers.

3. Reduction of total ownership costs. QoS-enabled component middleware defines crisp boundaries between components, which can help to reduce dependencies and maintenance costs associated with replacement, integration, and revalidation of components. Likewise, common components (such as event notifiers, resource managers, naming services, and replication managers) can be reused, thereby helping to further reduce development, maintenance, and validation costs.

1.3 Unresolved Technology Gaps for DRE Applications

Despite significant advances in standards-based QoS-enabled component middleware, however, there remain significant technology gaps that make it hard to support large-scale DRE systems in domains that require simultaneous

support for multiple QoS properties, including shipboard combat control systems [6] and supervisory control and data acquisition (SCADA) systems that manage regional power grids. Key technology gaps include the following:

1. Lack of effective isolation of DRE applications from heterogeneous middleware platforms. Advances in middleware technology and various standardization efforts, as well as market and economical forces, have resulted in a multitude of middleware stacks, such as CORBA, J2EE, SOAP, and .NET. This heterogeneity makes it hard to identify the right middleware for a given application domain. DRE systems are therefore built with too much reliance on a particular underlying middleware technology, resulting in maintenance and migration problems over system lifecycles.

2. Lack of tools for effectively composing DRE applications from components. DRE component middleware enables application developers to develop individual QoS-enabled components that can be composed together into *assemblies* that form complete DRE systems. Although this approach supports the use of "plug and play" components in DRE systems, system integrators now face the daunting task of composing the right set of compatible components that will deliver the desired semantics and QoS to applications that execute in large-scale DRE systems.

3. Lack of tools for configuring component middleware. In QoS-enabled component middleware frameworks, application components and the underlying component middleware services can have a large number of attributes and parameters that can be configured at various stages of the development lifecycle, such as:

- *During component development*, where default values for these attributes could be specified.
- *During application integration*, where component defaults could be overridden with domain specific defaults.
- *During application deployment*, where domain specific defaults are overridden based on the actual capabilities of the target system.

It is tedious and error-prone, however, to manually ensure that all these parameters are semantically consistent throughout a large-scale DRE system. Moreover, such *ad hoc* specification approaches have no formal basis for validating and verifying that the configured middleware will indeed deliver the end-to-end QoS requirements of applications throughout a DRE system.

4. Lack of tools for automated deployment of DRE applications on heterogeneous target platforms. The component assemblies described in bullet 2 above must be deployed in the distributed target environment before applications can start to run. DRE system integrators must therefore perform the complex task of mapping the individual components/assemblies onto specific nodes of the target environment. This mapping involves ensuring semantic compatibility between the requirements of the individual components, and the capabilities of the nodes of the target environment.

5. Lack of principled methodologies to support dynamic adaptation capabilities. To maintain end-to-end QoS in dynamically changing environments, middleware needs to be *adaptive*, which requires some means to (a) instrument the applications, middleware, operating systems, and networks to collect resource usage data and (b) adapt system behavior based on the collected data. DRE system developers have historically defined instrumentation and program adaptation mechanisms in an *ad hoc* way and used the collected data to maintain the desired QoS properties. This approach creates a tight coupling between applications and the underlying middleware, while also scattering the code that is responsible for reflection and adaptation throughout many parts of DRE middleware and applications, which makes it hard to configure, validate, modify, and evolve complex DRE systems consistently.

This paper describes how we are addressing the technology gaps described above using *Model Driven Middleware* (MDM). MDM is an emerging paradigm that integrates *model-based software techniques* (including Model-Integrated Computing [7,8] and the OMG's Model Driven Architecture [9]) with *QoS-enabled component middleware* (including Real-time CORBA [4] and QoS-enabled CCM [5]) to help resolve key software development and validation challenges encountered by developers of large-scale DRE middleware and applications. In particular, MDM tools can be used to specify requirements, compose DRE applications and their supporting infrastructure from the appropriate set of middleware components, synthesize the meta-data, collect data from application runs, and analyze the collected data to re-synthesize the required meta-data. These activities can be performed in a cyclic fashion until the QoS constraints are satisfied end-to-end.

1.4 Paper Organization

The remainder of paper is organized as follows: Section 2 describes key R&D challenges associated with large-scale DRE systems and outlines how the MDM paradigm can be used to resolve these challenges; Section 3 describes our work on MDM in detail, focusing on our CoSMIC toolsuite that integrates OMG MDA technology with QoS-enabled component middleware; Section 4 compares our work on CoSMIC with related research activities; and Section 5 presents concluding remarks.

2 Key DRE Application R&D Challenges and Resolutions

This section describes in detail the following R&D challenges associated with building large-scale DRE systems using component middleware that were outlined in Section 1:

(1) Safeguarding DRE applications against technology obsolescence
(2) Ensuring composition of valid DRE applications from sub-components
(3) Choosing semantically compatible configuration options
(4) Making effective deployment decisions based on target environment

(5) Assuring QoS at run-time

For each challenge listed above we describe the context in which it arises, the specific technology problem that needs to be solved, and outline how Model Driven Middleware (MDM) tools can be applied to help resolve the problem. Section 3 then describes how we are implementing these MDM solutions via CoSMIC, which is a toolsuite that combines MDA technology (such as the Generic Modeling Environment (GME) [10]) with QoS-enabled component middleware (such as the Component Integrated ACE ORB (CIAO) [5] that adds advanced QoS capabilities to the OMG CORBA Component Model). MDM expresses software functionality and QoS requirements at higher levels of abstraction than is possible using conventional programming languages (such as C, C++, and Java) or scripting languages (such as Perl and Python).

2.1 Challenge 1 - Safeguarding DRE Applications Against Technology Obsolescence

• **Context.** Component middleware refactors what was often historically *ad hoc* application functionality into individually reusable, composable, and configurable units. Component developers must select their component middleware platform and implementation language(s). Component developers may also choose to provide different implementations of the same functionality that use different algorithms and data structures to tailor their components for different use cases and target environments. This intellectual property must be preserved over extended periods of time, i.e., ∼15-30 years.

• **Problem – Accidental complexities in identifying the right technology and safeguarding against technology obsolescence.** Recent improvements in middleware technology and various standardization efforts, as well as market and economical forces, have resulted in a multiplicity of middleware stacks, such as those shown in Figure 3. The heterogeneity shown

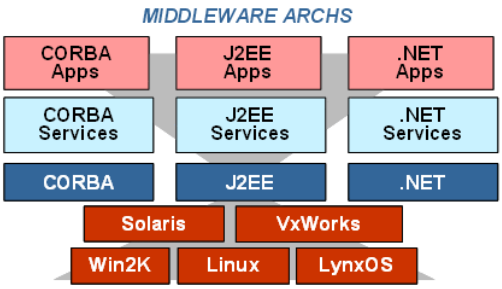

Fig. 3. **Popular Middleware Stacks**

in this figure makes it hard to identify the right middleware for a given application domain. Moreover, there are limitations on how much application code can be refactored into reusable patterns and components in various layers of each middleware stack. These refactoring limits in turn affect the optimization possibilities that can be implemented in different layers of the middle-

ware. Binding applications to one middleware technology - and expressing the application's QoS requirements in terms of that underlying technology - introduces unnecessary coupling between the application and the underlying middleware. Such early binding makes these applications obsolete when the underlying middleware is incapable of meeting application requirements that change over its lifetime.

• **Solution approach.** Our approach to Challenge 1 is to apply the MDM paradigm to model the functional and systemic (*i.e.*, QoS) requirements of components separately at higher levels of abstraction than that provided by conventional programming languages or scripting tools. MDM analysis and synthesis tools can then map these middleware independent models onto the appropriate middleware technology, which itself might change over the application's lifetime. Section 3 describes the architecture of CoSMIC, which is an integrated suite of MDM tools we are developing to address the challenge of identifying the right middleware technology and safeguarding against technology obsolescence.

2.2 Challenge 2 – Composing Valid DRE Applications from Component Libraries

•**Context.** Component-based applications are composed from a set of reusable components. Composition is an important step in developing component-based applications and composition techniques affect the reusability and semantics of the composite. Composition is typically performed by *packaging* (*i.e.*, bundling component implementations with associated systemic metadata), where a component can either be a standalone unit or an *assembly* (*i.e.*, group of inter-dependent, inter-connected components).

• **Problem – Inherent complexities in composing applications from a set of components.** As illustrated in Figure 4, composing an DRE ap-

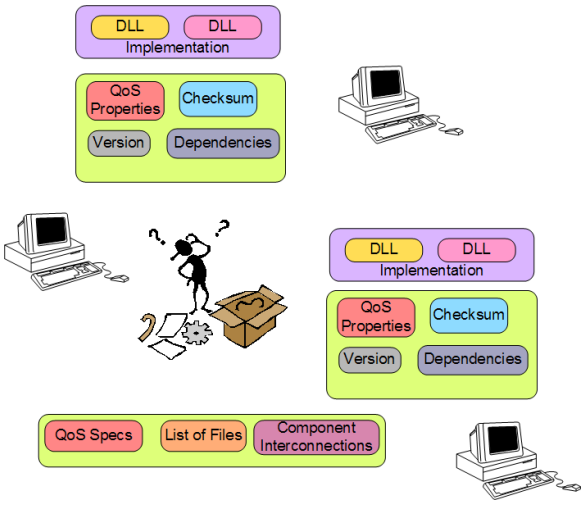

Fig. 4. **Application Composition**

plication by packaging components presents many problems to component

packagers. First, component connections should be checked for type incompatibility before they can be connected together. Second, collaborating components must be checked to ensure they have compatible semantics, which is hard to capture via interface signatures alone. For example, if a component developer has provided different implementations of the same functionality, it is necessary to assemble components that are semantically- and binary-compatible with each other. For DRE systems it is also essential that the assembled packages maintain the desired systemic QoS properties.

Challenge 2 therefore involves ensuring syntactic, semantic, systemic, and binary compatibility of assembled packages. *Ad hoc* techniques (such as manually selecting the components) are tedious, error-prone, and lack a solid analytical foundation to support verification and validation, and ensuring that the end-to-end QoS properties are satisfied with the given assembly. Likewise, *ad hoc* techniques for determining, composing, assembling, and deploying the right mix of semantically compatible, QoS-enabled COTS middleware components do not scale well as the DRE system size and requirements increase.

• **Solution approach.** Our approach to Challenge 2 involves developing MDM tools to represent component assemblies using the modeling techniques described in Section 3.2. These component assemblies are amenable to model checking [11], which in turn can ensure semantic and binary compatibility.

2.3 Challenge 3 – Choosing Semantically-compatible Configuration Options

• **Context.** Assuming a suitable component packaging capability exists, the next challenge involves configuring packages to achieve the desired functionality and systemic behaviour. Configuration involves selecting the right set of tunable knobs and their values at different layers of the middleware. For example, in QoS-enabled component middleware [5], both the components and the underlying component middleware framework may have a large number of configurable and tunable parameters, such as end-to-end priorities, size of thread pools, internal buffer sizes, locking mechanisms, timeout values, and request dispatching strategies.

• **Problem – Inherent complexities in middleware configuration.** In a large-scale DRE application, hundreds or thousands of components must be inter-connected. As shown in Figure 5, the number of configuration options and the set of compatible options can be overwhelming. This problem is exacerbated as the number of components increases. It is therefore tedious and error-prone to manually verify that the set of chosen options and their values are semantically consistent throughout a large-scale DRE system. Moreover, such *ad hoc* approaches have no formal basis for validating and verifying that the configured middleware will indeed deliver the end-to-end application QoS requirements.

• **Solution approach.** Our approach to Challenge 3 involves developing MDM configuration tools that support the (1) modeling and synthesis of configuration parameters for the middleware, (2) containers that provide the ex-

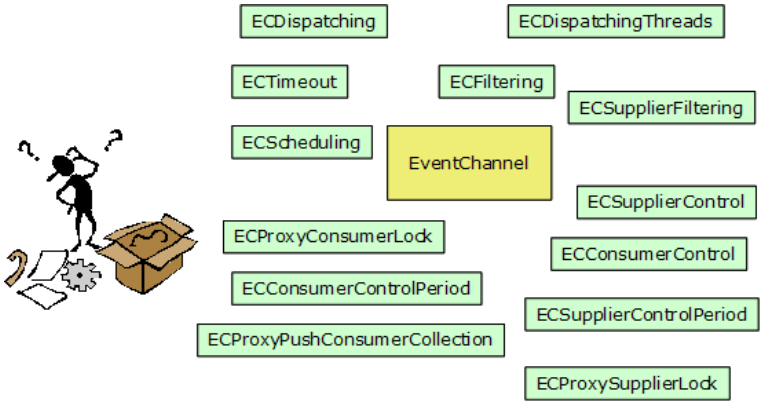

Fig. 5. **Middleware Configuration Challenges**

ecution context for application components, and (3) configuration of common middleware services, such as event notification, security, and replication. Section 3.3 describes how our MDM tools help ensure configuration parameters at different layers of a middleware stack are tuned to work correctly and efficiently with each other.

2.4 Challenge 4 – Making Effective Deployment Decisions based on Target Environment

• **Context.** Applications that run in DRE systems often possess multiple QoS requirements, such as acceptable deadlines for various time-critical functionality, support for specific synchronization mechanisms, and resource limits that the middleware must enforce on the target platform. This enforcement process involves planning and preparing the deployment of components. The goal is to satisfy the functional and systemic requirements of DRE applications by making appropriate deployment decisions, which take into account the properties of the target environment, and to retain flexibility by not committing prematurely to physical resources.

• **Problem – Satisfying multiple QoS requirements simultaneously.** As illustrated in Figure 6, planning includes specifying the target environment and making appropriate component deployment decisions. Deployment involves coming up with a mapping between the components of the application and the nodes of the target environment where these components will run. This mapping is very hard to do manually, *i.e.*, it is equivalent to assignment problems in operations research. As the number of components and their target nodes increases, satisfying multiple QoS requirements simultaneously cannot be solved without automated methods.

• **Solution approach.** Due to the layering and partitioning of large-scale DRE systems, it is necessary to have a sequence of steps that will ensure that functional dependencies are met and the systemic requirements are satisfied after deployment. Our approach to challenge 4 involves developing the MDM tools described in Section 3.4 that (1) model the target environment and (2) determine how deployment can be made based on an analysis of required

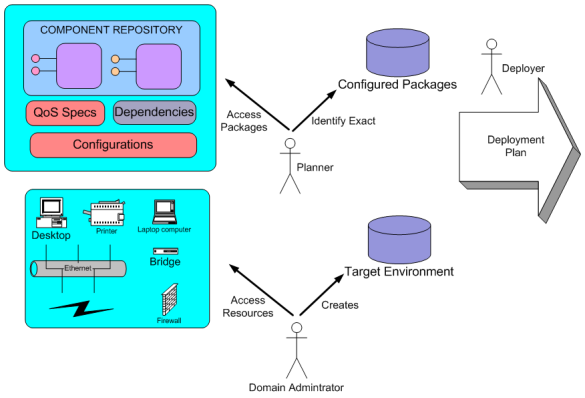

Fig. 6. **Planning for Deployment**

end-to-end QoS of components, and capabilities of the nodes in the given target environment. For example, target environment modeling includes the network topology, the network technology and the available bandwidth, the CPUs, and the OS they run and its available memory that are used to make suitable deployment decisions. Moreover, target environment models can be combined with component package models to synthesize custom test suites that can benchmark different aspects of DRE application and middleware performance. In turn, this empirical benchmark data can be used in end-to-end QoS prediction analysis tools to guide the deployment of components throughout a DRE system.

2.5 Challenge 5 – Assuring QoS at Run-time

• **Context.** After a large-scale DRE system has been deployed, the operational conditions of the target environment may change. Various factors are responsible for these changes, including excessive loads on CPUs, network congestion, or different types of faults. Even in such dynamically changing conditions, it is important that the mission-critical capabilities of DRE systems continue to operate at their desired QoS levels.

• **Problem – Dealing with the vagaries of a distributed environment.** Once the application is deployed, it is necessary to maintain end-to-end QoS in dynamically changing environments. This activity motivates the need for an adaptive and reflective DRE middleware capable of distributed dynamic resource management. Adaptation requires instrumenting software to reflect upon the run-time application, middleware, operating systems, and network resource usage data and adapting the behavior based on the collected data. DRE system developers have historically defined middleware instrumentation and program adaptation mechanisms in an *ad hoc* way and used the collected data to maintain the desired QoS properties. This approach creates a tight coupling between applications and the underlying middleware, while also scattering the code that is responsible for reflection and adaptation throughout many parts of DRE software, which makes it hard to configure, validate, modify, and evolve complex DRE systems consistently.

• **Solution approach.** Our approach to Challenge 5 involves applying MDM tools to model abnormal and/or failure conditions in the run-time environment. As described in Section **??**, these models can be combined with the functional and systemic models of DRE systems to synthesize appropriate controllers that encode dynamic resource management strategies and adaptation logic for applications and middleware.

3 Resolving DRE Application Lifecycle Challenges with Model Driven Middleware

To address the challenges described in Section 2, principled methods are needed to specify, develop, compose, integrate, and validate the application and middleware software used by DRE systems. These methods must enforce the physical constraints of DRE systems, as well as satisfy the system's stringent functional and systemic QoS requirements. Achieving these goals requires a set of integrated Model Driven Middleware (MDM) tools that allow developers to specify application and middleware requirements at higher levels of abstraction than that provided by low-level mechanisms, such as conventional general-purpose programming languages, operating systems, and middleware platforms.

In the context of DRE middleware and applications, MDM tools can be applied to:
• **Model** different functional and systemic properties of DRE systems via separate middleware- and platform-independent models [12]. Domain-specific aspect model weavers [13] can integrate these different modeling aspects into composite models that can be further refined by incorporating middleware and platform-specific properties.
• **Analyze** different—but interdependent—characteristics and requirements of DRE system behavior (such as scalability, predictability, safety, schedulability, and security) specified via models. Model *interpreters* [10] translate the information specified by models into the input format expected by model checking [11] and analysis tools [14]. These tools can check whether the requested behavior and properties are feasible given the specified application and resource constraints. Tool-specific model analyzers [15,16] can also analyze the models and predict [17] expected end-to-end QoS of the constrained models.
• **Synthesize** platform-specific code and metadata that is customized for a particular QoS-enabled component middleware and DRE application properties, such as end-to-end timing deadlines, recovery strategies to handle various run-time failures in real-time, and authentication and authorization strategies modeled at a higher level of abstraction [18,19].
• **Provision** middleware and applications by assembling and deploying the selected components end-to-end using the configuration metadata synthesized by MDM tools. In the case of legacy components that were developed without

consideration of QoS, the provisioning process may involve invasive changes to existing components to provide the hooks that will adapt to the metadata. The changes can be implemented in a relatively unobtrusive manner using program transformation systems, such as DMS [20].

• **Assure** run-time QoS properties are delivered to applications in DRE systems, *e.g.*, via modeling dynamic adaptation and resource management strategies that use hybrid control-theoretic [21] techniques.

OMG MDA technologies initially focused largely on enterprise applications. More recently, MDA technologies have emerged to customize QoS-enabled component middleware for DRE systems, including aerospace [22], telecommunications [23], and industrial process control [24]. This section describes our R&D efforts that focus on integrating the MDA paradigm with QoS-enabled component middleware to create an MDM toolsuite called CoSMIC (Component Synthesis using Model Integrated Computing). As shown in Figure 7, CoSMIC consists of an integrated collection of modeling, analysis, and synthesis tools that address key lifecycle challenges of DRE middleware and applications.

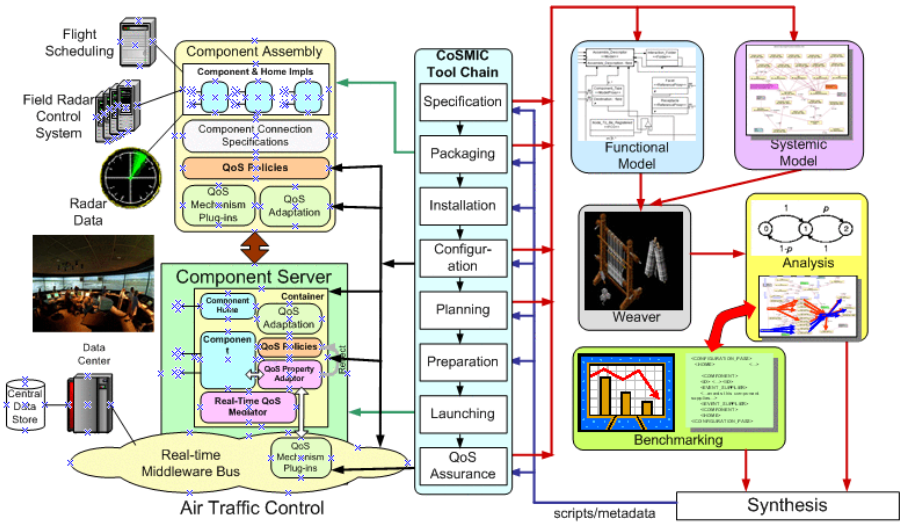

Fig. 7. **CoSMIC Model Driven Middleware Toolsuite**

The CoSMIC tools are based on the Generic Modeling Environment (GME) [10], which is a meta-modeling environment that defines the modeling paradigms for each stage of the CoSMIC tool chain. The CoSMIC tools use GME to enforce their "correct by construction" techniques, as opposed to the "construct by correction" techniques commonly used by post-construction tools, such as compilers and script validators. CoSMIC ensures that the rules of construction – and the models constructed according to these rules – can evolve together over time. Each CoSMIC tool synthesizes metadata in XML and passes this XML to the next stage of the tool chain, where it is further processed and refined.

The CoSMIC toolsuite currently uses a *platform-specific model* (PSM) approach that integrates the modeling technology with our CIAO QoS-enabled component middleware [5]. We chose CIAO as our initial focus since it is targeted to meet the QoS requirements of DRE applications. As other component middleware platforms (such as J2EE and .Net) mature and become suitable for DRE applications, we will enhance the CoSMIC toolsuite so it supports *platform-independent models* (PIMs) and then include the necessary patterns and policies to map the PIMs to individual PSMs for the various component middleware platforms.

The remainder of this section describes the tools in the CoSMIC toolsuite, focusing on the modeling paradigms we developed for each tool and how the tool helps resolve the R&D challenges described in Section 2. To make the tool discussions concrete, however, we first describe a representative scenario of a DRE avionics system developed using QoS-enabled component middleware.

3.1 Demonstrating CoSMIC via Boeing Avionics Scenarios

The QoS requirements of different DRE systems vary, *e.g.*, DRE systems involving audio streams require predictable end-to-end latencies and jitter, whereas video streams often require significant bandwidth. QoS-enabled middleware should therefore be configured appropriately to deliver the desired QoS to the target DRE systems. To demonstrate the deployment challenges of DRE applications, we use a navigation display simulation based on the Boeing Bold Stroke avionics mission computing environment that receives the global positions from a GPS device and displays them at a GUI display in a timely manner. The desired data request and the display frequencies are fixed at 40 Hz. The BoldStroke architecture uses a *push event/pull data* publisher/-subscriber communication paradigm [25] atop the PRISM QoS-enabled component middleware platform [26].

The component interaction for the navigation display example is depicted in Figure 8. The scenario shown in Figure 8 begins with the GPS being invoked

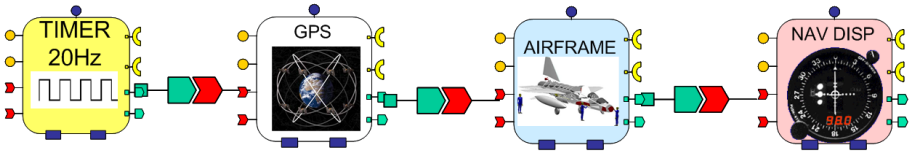

Fig. 8. **Navigation Display Collaboration Example**

by the TAO Real-time Event Service [27], shown as a `Timer` component. On receiving a pulse event from the `Timer`, the `GPS` generates its data and issues a data available event. TAO's Real-time Event Service then forwards the event

on to the `Airframe` component, which retrieves the data from the `GPS` component, updates its state, and issues a data available event. The Event Service forwards the event to the `Nav_Display` component, which in turn retrieves the data from the `GPS`, updates its state, and displays it.

3.2 Model-driven Component Packaging: Resolving Component Packaging Challenges

CoSMIC provides the Composable Adaptive Software Systems (COMPASS) tool to resolve the problem of packaging component functionality described in Challenge 2 of Section 2.2. COMPASS defines a modeling paradigm that allows DRE application integrators to model the component assembly and packaging aspect. In addtion, COMPASS provides built-in constraint checkers that validate syntactic, semantic, and binary compatibility of the assembled components. The COMPASS model interpreter also synthesizes metadata that describes certain properties of component packages, such as ...

Figure 9 illustrates how COMPASS fits into the overall CoSMIC tool chain. Below we describe how the COMPASS tool is used by DRE application inte-

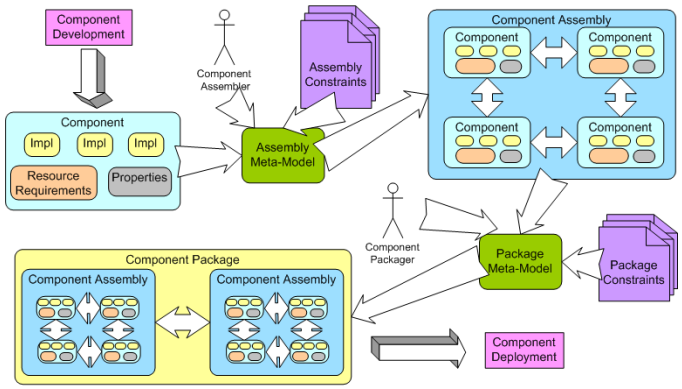

Fig. 9. **COMPASS**

grators.

3.2.1 Modeling Paradigm

The modeling paradigm of COMPASS comprises different packaging and configuration artifacts, as well as the legal domain-specific associations between the various artifacts. The modeling paradigm enables application integrators to visualize the packages at different levels of abstractions *i.e.*, at the level of package, assembly, and individual components. Visualization of abstractions is achieved by using the hierarchy inherent in composition-based approaches of software development *i.e.*, it utilizes the hierarchy of individual packages, the

set of assemblies contained within a package, and the individual components contained as part of each assembly.

Since components can be composed from assemblies of sub-components, individual components must be associated with information about their properties and requirements so that informed decisions can be made at composition time by application integrators and tools. By making both properties and requirements as *first-class* entities of the modeling paradigm, COMPASS ensures that the properties of the set of available components can be matched against the set of requirements. This matching is done via metrics defined by the OMG *Deployment and Configuration of Component-based Distributed Applications* (D&C) specification [28], including (1) *quantity*, which is a restriction on number (em e.g., number of available processors), (2) *capacity*, which is a restriction on consumption (*e.g.*, available bandwidth), (3) *minimum*, which is a restriction on the allowed minimum (*e.g.*, minimum latency), (4) *maximum*, which is a restriction on the allowed maximum (*e.g.* maximum throughput), (5) *equality*, which is a restriction on the allowed value *e.g.*, the required operating system), and (6) *selection*, which is a restriction on a range of allowed values (*e.g.*, allowed versions of a library satisfying a dependency).

3.2.2 Constraint Specification

COMPASS provides a constraint checker to ensure that packages it creates are valid. This checker plays a crucial role in enforcing CoSMIC's "correct by construction" techniques. Constraints are defined on elements in the COMPASS meta-model using the Object Constraint Language (OCL) [29], which is a strongly typed, declarative, query and constraint language that has formal semantics that domain experts can use to describe their domain constraints. For example, COMPASS defines constraints to capture the restrictions that exist in the context of component packaging and configuration, including (1) creation of component packages, (2) interconnection of component packages, (3) composition of packages, (4) creation of component assemblies, (5) interconnection of component assemblies, (6) composition of assemblies, (7) creation of components, and (8) interconnection of components.

Adding constraints to the COMPASS meta-model ensures that illegal connections are not made among the various modeling elements. These constraints help catch errors early in the component development cycle. Since COMPASS performs static model checking, it has the added advantage that sophisticated constraint checking can be done prior to application instantiation, without incurring the cost of run-time constraint checking.

3.2.3 Model Interpretation

The COMPASS model interpreter translates the various packaging and configuration information captured in the models constructed using its meta-model into a set of *descriptors*, which are files containing meta-data that describe the systemic information of component-based DRE applications. The output of the COMPASS model interpreter serves as input to other downstream tools, such as the deployment planner described in Section 3.4 that uses information in the descriptors to deploy the components. The descriptors generated by COMPASS model interpreter are XML documents that conform to a XML Schema [30,31]. To ensure interoperability with other CoSMIC modeling tools, COMPASS synthesizes descriptors conforming to the XML schema defined by the OMG D&C specification, which defines the following four different types of descriptors:

- *Component package descriptor*, which describes the elements in a package.
- *Component implementation descriptor*, which describes elements of a specific implementation of an interface, which might be a single implementation or an assembly of interconnected sub-component implementations.
- *Implementation artifact descriptor*, which describes elements of a component implementation.
- *Component interface descriptor*, which describes the interface of a single component along with other elements such as component ports.

The output of COMPASS can be validated by running the descriptors through any XML schema validation tool, such as Xerces. The generated descriptors are input to the CoSMIC run-time infrastructure, which uses this information to instantiate the different components of the application and interconnect the different components.

3.3 Model-driven Middleware Configuration: Resolving Configuration Challenges

CoSMIC provides a group of tools to address the problem of multi-layer middleware configuration discussed in Challenge 3 of Section 2. The group comprises the Option Configuration Modeling Language (OCML) tool that handles ORB-level configurations, the Event QoS Aspect Language (EQAL) tool that addresses container-level configurations, and the Federated Event Service Modeling Language (FESML) tool that addresses application-level configurations. Each tool consists of two parts: (1) a *modeling paradigm* in which models can be built and (2) a *model interpreter* that synthesizes configuration metadata in service configuration files.

3.3.1 Modeling paradigms.

The meta-model for each tool outlined above defines a modeling paradigm and contains the various types of configuration models, individual configuration parameters, and constraints that enforce model dependencies. Below we describe the modeling paradigm for the three tools.

• **OCML.** The OCML modeling paradigm addresses middleware-level configuration options. OCML contains artifacts to define and categorize the middleware options and to configure the middleware with these options. OCML also generates the documentation for the middleware options. OCML is based on the Graphical Modeling Environment (GME). As shown in Figure 10 the OCML tool is intended to be used by both middleware developers and application developers. Middleware developers design the model of the middle-

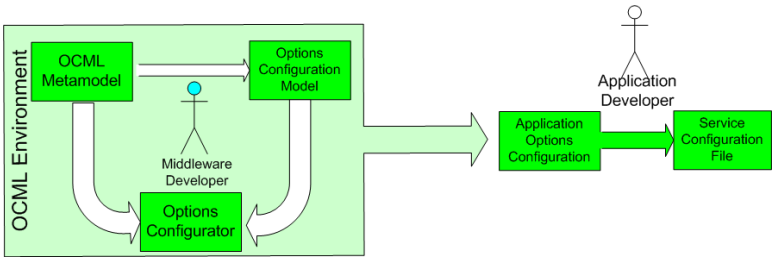

Fig. 10. **OCML Process**

ware configuration properties (Options Configuration Model) with the OCML metamodel. Application developers then configure the middleware to satisfy a specific application's QoS requirements.

The options configuration model contains two artifacts: (a) the *structure* artifact, which contains all the available options categorized hierarchically in different folders (*e.g.*, OCML has been used to model all the configuration options provided by the TAO [32] ORB) and (b) the *rules* artifact, which contains the dependency relations among these options.

The Options Configuration Model is interpreted through a model interpreter and an Options Configurator application is generated according to the artifacts defined by the model. The generated application is used by application developers to model a permutation of ORB configuration options to be used for configuring the ORB. After validating the compatibility of the selected options, a ORB-specific service configuration file is generated.

• **EQAL.** EQAL focuses on the configuration of real-time event services in QoS-enabled component middleware. EQAL currently provides support for event service configuration in the Component-Integrated ACE ORB (CIAO). CIAO employs two types of CORBA event services: a real-time event service and a notification service. These services allow components to asynchronously and anonymously send and receive customized data structures called events.

EQAL allows the specification of an event mechanism (*i.e.*, real-time event service or notification service) and the configuration of that mechanism for each component event connection. Each mechanism has unique capabilities and requires a distinct set of modeling constructs. Policies and strategies that can be modeled include, but are not limited to, filtering, correlation, timeouts, locking, disconnect control, and priority. Various policies have differing scope, from a single port to an entire event channel. EQAL allows the modeler to create reusable and sharable configurations at each level of granularity. The modeler assigns configurations to individual event connections and then constructs filters for each connection.

• **FESML.** One strategy by which a real-time event service can be configured to minimize network traffic is building federations of event services that share filtering information to minimize or eliminate the transmission of unwanted events to a remote entity. Moreover, federation of event services allows events that are being communicated in one channel to be made available on other channels. The channels could communicate with each other through CORBA IIOP Gateways, UDP, or IP Multicast [33]. Connecting event channels from different systems together will allow event information to be interchanged, providing a level of integration among the systems.

To ensure support for synthesizing the configuration of federation of event services, CoSMIC provides the Federated Event Service Modeling Language (FESML) tool. As part of the CoSMIC tool chain, FESML uses MDA technology and provides a visual interface for modeling the interactions among different event artifacts in the distributed system. These artifacts include event consumers, event suppliers, event channels, CIAO Gateways, UDP Senders, UDP Receivers and Multicast ports.

3.3.2 Constraint Specification

Dependencies among middleware QoS policies, strategies, and configurations are complex. Ensuring coherency among policies and configurations has been a major source of complexity in component middleware. One of CoSMIC's primary benefits is the prevention of inconsistent QoS parameters during modeling time through constraints. Constraints ensure that only valid models can be constructed and interpreted.

• **OCML.** The rules artifact of OCML is used to define the constraints which the ORB service configuration is required to satisfy. These constraints are enforced to be satisfied by the application developer in the Service Configuration Modeling Environment.

• **EQAL.** EQAL automatically verifies the validity of event service configurations and notifies the user during modeling time of incompatible QoS properties. Consequently, EQAL dramatically reduces the time and effort involved in configuring components with stringent real-time requirements.

• **FESML.** To ensure that the federation of Event Service work properly, event channel settings are validated. FESML provides a built-in constraint model checker that checks for syntactic and semantic compatibility of the federation of event channels. This model checker provides us the opportunity to detect consistent event channel settings in an early design phase rather than the assembly and deployment phase.

3.3.3 Model Interpretation

The CoSMIC middleware configuration toolset provides model interpreters that synthesize middleware configuration files and component descriptor files.
• **OCML.** The middleware-specific options configuration language is validated against the OCML meta-model and when interpreted generates the following:

- Source code for the service configuration design environment. Service configuration design environment is used by the application developer to generate ORB service configuration files.
- Source code for a handcrafted service configuration file validation tool.
- An HTML file documenting all the options and the dependencies.

This procedure is illustrated in Figure 10.
• **EQAL.** EQAL encompasses two model interpreters. The first interpreter generates XML descriptor files that conform to the Boeing Bold Stroke XML schema for Real-time Event Service component configuration. These descriptor files identify the real-time requirements of individual connections. The second interpreter generates the service configuration files that specify event channel policies and strategies. The component deployment framework parses these files, creates event channels, and configures each connection, while shielding the actual component implementations from the lower-level middleware services. Currently, these files must be written by hand a tedious process that is repeated for each component deployment. Accordingly, the automation of this process, and the guarantee of model validity, improves the reusability of components across diverse deployment scenarios.
• **FESML.** FESML includes a model interpreter that generates XML files to specify the configuration of the federation of event channels. The information captured in the descriptor files include the relationship between each artifacts, the physical location of each supplier, consumer, event channel, CIAO Gateway, etc. These files can then be fed to the CIAO assembly and deployment tool to deploy the system.

3.4 Model-driven Configuration and Deployment of Components: Resolving Deployment Planning Challenges

CoSMIC provides the Model Integrated Deployment and Configuration Environment for Composable Software Systems (MIDCESS) and the CCM Per-

formance (CCMPerf) tools to resolve the problem of deployment planning described in challenge 4 of Section 2.4. MIDCESS can be used to specify the *target environment* for deploying packages. A target environment is a model of the computing resource environment (such as processor speed and type of operating system) in which a component-based application will execute. The various entities of the target model include:

(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 intercommunicate.
(3) **Bridges** among interconnects. Interconnects provide a direct connection between nodes, while bridges provide a routing capability between interconnects.

Nodes, interconnects, and bridges are collected into a *domain*, which collectively represents the target environment.

Using the target environment information available from MIDCESS, CCMPerf [34] can then be used to create experiments that measure *black-box* (*e.g.*, latency, jitter, and throughput) and white-box (*e.g.*, context-switch overhead) metrics that can be used to evaluate the consequences of mixing and matching component assemblies in a particular target environment. In the context of the BasicSP scenario, CCMPerf can be used to identify the set of nodes that minimize latency between any two components. The experiments in CCMPerf can be divided into the following three experimentation categories:

(1) *Distribution middleware* tests that quantify the performance of CCM-based applications using black-box and white-box metrics, for example, measuring latency for navigation updates to propagate to the `Nav_Display` component for a given domain,
(2) *Common middleware services* tests that quantify the suitability of using different implementations of CORBA services, such as using Real-time Event [35] service against the Notification Services [36] for delivering periodic trigger updates to the `Nav_Display` component, and
(3) *Domain-specific middleware* tests that quantify the suitability of CCM implementations to meet the QoS requirements of a particular DRE application domain, such as jitter metrics for associating real-time policies with component servers and containers that host `Timer` and `Nav_Display` components in the BasicSP scenario.

A model-driven approach to deployment planning allows modelers to get information about the target environment, get the middleware configuration information, and generate tests at the push of button. Without modeling techniques, these tedious and error-prone code would have to be written by hand. In a hand-crafted approach, changing the configuration would entail re-writing

the benchmarking code. In a model based solution, however, the only change will be in the model and the necessary experimentation code will be automatically generated. A model based solution also provides the right abstraction to visualize and analyze the overall planning phase rather than looking at the source code. In the ensuing paragraphs we describe the design of MIDCESS and CCMPerf.

Figure 11 illustrates how MIDCESS and CCMPerf are designed to be a link in the CoSMIC tool chain that enables developers to model the planning phase of the component development process. Below we describe how the MIDCESS

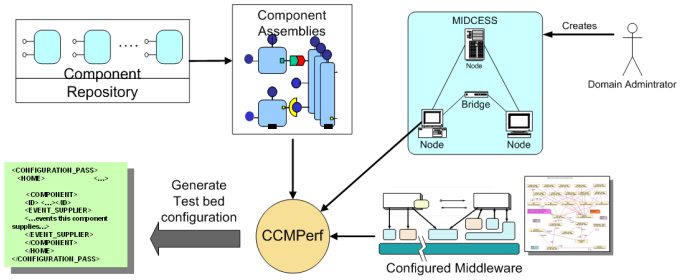

Fig. 11. **PLANNING**

and the CCMPerf tools are used by the domain administrators and planners.

3.4.1 Modeling Paradigm

The meta-model for each tool defines a modeling paradigm and contains the various types of configuration models, individual configuration parameters, and constraints that enforce model dependencies.

• **MIDCESS.** MIDCESS is a graphical tool that provides a visual interface for specifying the target environment for deploying DRE applications. The modeling paradigm contains entities to model the various artifacts of the target environment for deploying composable software systems and also the interconnections between those artifacts. The modeling paradigm also allows the domain administrators to visualize the target environment at various levels of abstractions *i.e.* at the level of domains and sub-domains. MIDCESS also provides built-in constraint checkers that check for the semantic compatibility of the specified target environment. For example, the constraint checker could check for connections involving bridges and make sure that no two nodes are directly connected using a bridge.

The MIDCESS tool enables the modeling of the following features of a target environment:

(1) Specification of node elements and the interconnections between the node elements.
(2) Specification of the attributes of each of the nodes.

(3) Hierarchical modeling of the individual nodes that share certain basic attributes (such as their type), but vary in the processing power, supported OS etc.
(4) Hierarchical modeling of the interconnects to specify the different varieties of connections possible in the target environment.
(5) Hierarchical modeling of the domain to have sub-domains.

• **CCMPerf.** The modeling paradigm of CCMPerf is defined in a manner that will allow its integration with other paradigms, for example, COMPASS. To achieve the aforementioned goal, CCMPerf defines *Aspects*, *i.e.*, visualizations of existing meta model that allows the modeler to depict component interconnection and associate metrics the above interaction. The following are the three aspects defined in CCMPerf

(1) *Configuration Aspect*, that defines the interface that are provided and required by the individual component, for example modeling the events propagated by the `Airframe` Component,
(2) *Metric Aspect*, that defines the metric captured in the benchmark, for example associating latency information for GPS position updates generated by the `GPS` components and received by the `Nav_Display` component, and
(3) *Inter-connection Aspect*, that defines how the components will interact in the particular benchmarking experiment, for example connecting the provides and required ports of the `Airframe` component with the corresponding ports of the `Nav_Display` component.

3.4.2 Constraints Specification

The meta-model for each tool defines a modeling paradigm and contains the various types of configuration models, individual configuration parameters, and constraints that enforce model dependencies.

• **MIDCESS.** MIDCESS contains a constraint checker to ensure that the target environments specified by the tool are semantically compatible. Constraints are defined using the Object Constraint Language (OCL) [29], which is a strongly typed, declarative, query and constraint language. MIDCESS defines constraints to enforce restrictions in the (1) specification of node elements, (2) specification of interconnect elements, (3) specification of bridge elements, (4) specification of resource elements, and (5) interconnection of various elements of the domain.

• **CCMPerf.** Additionally, a constraint checker validates the experiment precluding the possibility of invalid configuration, such as: (1) conflicting metrics (*e.g.*, using both back box and white box metrics in a given experiment), (2) invalid connections (*e.g.*, not connecting a required interface with the corresponding provides interface). In the BasicSP scenario this constraint violation corresponds to connecting `Nav_Display` ports directly to the `GPS` component instead of the `Ariframe` component and (3) incompatible exchange format

(*e.g.*, connecting a point-to-point entity with a point to multi point entity). Connecting `Timer` refreshes (events) to position updates generated for the `Nav_Display` ("pull" operations) correspond to such a violation. Constraints are defined in the CCMPerf meta model are defined using OCL [29]. The use of constraints ensure that the experiment is correct a priori minimizing errors at run-time.

3.4.3 Model Interpretation

The meta-model for the MIDCESS and CCMPerf tools defines a modeling paradigm and contains the various types of configuration models, individual configuration parameters, and constraints that enforce model dependencies.

• **MIDCESS.** MIDCESS generates a *domain descriptor* that describes the domain aspect of the target model environment of composable software systems. This descriptor is an XML document that conforms to a XML Schema defined by the *Deployment and Configuration of Component-based Distributed Applications Specification* [28]. The output of MIDCESS can therefore be validated by running the descriptor through a tool that supports XML schema validation. The generated descriptor is also used by the CoSMIC run-time infrastructure, which uses information in the descriptor to make deployment planning decisions.

• **CCMPerf.** From the CCMPerf meta-model, an interpreter generates the necessary descriptor files that provide meta-data to configure the experiment. In addition to the descriptor files, the CCMPerf interpreter also generates benchmarking code that monitors and records the values for the variables under observation. To allow the experiments to be carried out in varied hardware platforms, script files can be generated to run experiments.

3.4.4 Resolving BasicSP Scenario Configuration Challenges using MIDCESS and CCMPerf

We now demonstrate how the BasicSP configuration challenges can be resolved using the above tools. For example, providing the application developer with the QoS metrics such as latency, throughput or jitter for the scenario on a target platform at design time would help in making intelligent decisions on mapping components on to appropriate nodes in the domain. For this achieving this goal, the engineer, would use the CCMPerf modeling environment, to compose a representative test application and associate certain, QoS requirement, such as minimizing latency with the scenario. Figures 13, depicts how the component interaction and QoS association can be modeled using CCMPerf.

The CCMPerf interpreters generate all the scaffolding code required to set up,

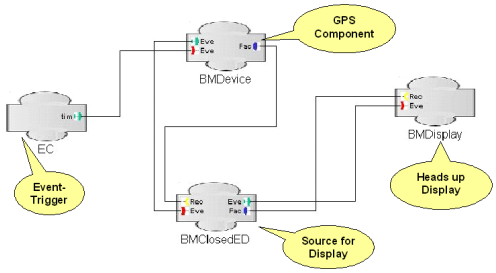

Fig. 12. **Modeling Component Interaction using CCMPerf**

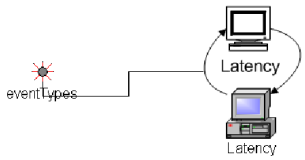

Fig. 13. **Associating QoS Attributes with BasicSP Scenario**

run and tear down the experiment. The following steps help resolve configuration and deployment challenges for DRE applications:

(1) Determine the QoS expected from the middleware.
(2) Select a set of middleware configurations options using the OCML tool (Section 3.3) that are expected to provide these QoS guarantees. It is assumed that the middleware developers will have the appropriate insights to select the right options.
(3) Use the CCMPerf tool to generate a testsuite for evaluating QoS delivered by the middleware.
(4) Use the MIDCESS tool to model the target configuration and synthesize the necessary descriptors for component deployment. the set.
(5) For each configuration option discussed in step 4, run the generated benchmarking tests to evaluate the QoS [1] .
(6) Repeat steps 4-5 for DRE systems in different nodes in the domain by mapping components to individual nodes. If a particular combination of configuration option along with the target mapping set delivers similar QoS properties, then it is good candidate solution.

4 Related Work

This section reviews related work on model-based software development and describes how modeling, analysis, and generative programming techniques are being used to model and provision QoS capabilities for DRE component middleware and applications.

Model-based software development. Our work on Model Driven Middleware extends earlier work on Model-Integrated Computing (MIC) [7,39,40,8] that focused on modeling and synthesizing embedded software. MIC provides a unified software architecture and framework for creating Model-Integrated Program Synthesis (MIPS) environments [10]. Examples of MIC technology used today include the Generic Modeling Environment (GME) [10] and Ptolemy [41] (used primarily in the real-time and embedded domain) and MDA [9] based

[1] The challenges arising from the explosion in the configuration space can be alleviated using pruning techniques discussed in research [37,38].

on UML [42] and XML [43] (which have been used primarily in the business domain). Our work on CoSMIC combines the GME tool and UML modeling language to model and synthesize QoS-enabled component middleware for use in provisioning DRE applications. In particular, CoSMIC is enhancing GME to produce domain-specific modeling languages and generative tools for DRE applications, as well as developing and validating new UML profiles (such as the UML profile for CORBA [44], the UML profile for quality of service [45], and UML profile for schedulability, performance and time [46]) to support DRE applications.

As part of an ongoing collaboration [47] between ISIS, University of Utah, and BBN Technologies, work is being done to apply GME techniques to model an effective resource management strategy for CPU resources on the Timesys Linux real-time OS [48]. Timesys Linux allows applications to specify CPU reservations for an executing thread, which guarantee that the thread will have a certain amount of CPU time, regardless of the priorities of other threads in the system. Applying GME modeling to develop the QoS management strategy simplifies the simulation and validation necessary to assure end-to-end QoS requirements for CPU processing.

The Virginia Embedded System Toolkit (VEST) [49] is an embedded system composition tool that enables the composition of reliable and configurable systems from COTS component libraries. VEST compositions are driven by a modeling environment that uses the GME tool [10]. VEST also checks whether certain real-time, memory, power, and cost constraints of DRE applications are satisfied.

The Cadena [11] project provides an MDA toolsuite with the goal of assessing the effectiveness of applying static analysis, model-checking, and other light-weight formal methods to CCM-based DRE applications. The Cadena tools are implemented as plug-ins to IBM's Eclipse integrated development environment (IDE) [50]. This architecture provides an IDE for CCM-based DRE systems that ranges from editing of component definitions and connections information to editing and debugging of auto-generated code templates.

Commercial successes in model-based software development include the Rational Rose [51] suite of tools used primarily in enterprise applications. Rose is a model driven development toolsuite that is designed to increase the productivity and quality of software developers. Its modeling paradigm is based on the Unified Modeling Language (UML). Rose tools can be used in different application domains including business and enterprise/IT applications, software products and systems, and embedded systems and devices. In the context of DRE applications, Rose has been applied successfully in the avionics mission computing domain [2].

Other commercial successes include the Matlab Simulink and Stateflow tools that are used primarily in engineering applications. Simulink is an interactive tool for modeling, simulating, and analyzing dynamic, multidomain systems. It provides a modeling paradigm that covers a wide range of domain areas, including control systems, digital signal processors (DSPs), and telecommunication systems. Simulink is capable of simulating the modeled system's behavior, evaluating its performance, and refining the design. Stateflow is an interactive design tool for modeling and simulating event-driven systems. Stateflow is integrated tightly with Simulink and Matlab to support designing embedded systems that contain supervisory logic. Simulink uses graphical modeling and animated simulation to bridge the traditional gap between system specification and design.

Program transformation technologies. Program Transformation [20] is the act of changing one program to another. It provides an environment for specifying and performing semantic-preserving mappings from a source program to a new target program. Program transformation is used in many areas of software engineering, including compiler construction, software visualization, documentation generation, and automatic software renovation.

Program transformations are typically specified as rules that involve pattern matching on an abstract syntax tree (AST). The application of numerous transformation rules evolves an AST to the target representation. A transformation system is much broader in scope than a traditional generator for a domain-specific language. In fact, a generator can be thought of as an instance of a program transformation system with specific hard-coded transformations. There are advantages and disadvantages to implementing a generator from within a program transformation system. A major advantage is evident in the pre-existence of parsers for numerous languages [20]. The internal machinery of the transformation system may also provide better optimizations on the target code than could be done with a stand-alone generator.

Generative Programming (GP) [52] is a type of program transformation concerned with designing and implementing software modules that can be combined to generate specialized and highly optimized systems fulfilling specific application requirements. The goals are to (1) decrease the conceptual gap between program code and domain concepts (known as achieving high intentionality), (2) achieve high reusability and adaptability, (3) simplify managing many variants of a component, and (4) increase efficiency (both in space and execution time).

GenVoca [19] is a generative programming tool that permits hierarchical construction of software through the assembly of interchangeable/reusable components. The GenVoca model is based upon stacked layers of abstraction that can be composed. The components can viewed as a catalog of problem solutions that are represented as pluggable components, which then can be used

to build applications in the catalog domain.

Yet another type of program transformation is aspect-oriented software development (AOSD). AOSD is a new technology designed to more explicitly separate concerns in software development. The AOSD techniques make it possible to modularize crosscutting aspects of complex DRE systems. An aspect is a piece of code or any higher level construct, such as implementation artifacts captured in a MDA PSM, that describes a recurring property of a program that crosscuts the software application *i.e.*, aspects capture crosscutting concerns). Examples of programming language support for AOSD constructs include AspectJ [53] and AspectC++ [54].

5 Concluding Remarks

Large-scale DRE applications are increasingly being developed using QoS-enabled component middleware [5]. QoS-enabled component middleware provides policies and mechanisms for provisioning and enforcing large-scale DRE application QoS requirements. The middleware itself, however, does not resolve the challenges of choosing, configuring, and assembling the appropriate set of syntactically and semantically compatible QoS-enabled DRE middleware components tailored to the application's QoS requirements. Moreover, any given middleware API does not resolve all the challenges posed by obsolescence of infrastructure technologies and its impact on long-term DRE system lifecycle costs.

It is in this context that the OMG's Model Driven Architecture (MDA) is an effective paradigm to address the challenges described above. The MDA is a software development paradigm that applies domain-specific modeling languages systematically to engineer computing systems. This paper provides an overview of the emerging paradigm of *Model Driven Middleware* (MDM), which integrates *model-based software techniques* (including Model-Integrated Computing [7,8] and the OMG's Model Driven Architecture [9]) with *QoS-enabled component middleware* (including Real-time CORBA [4] and QoS-enabled CCM [5]) to help resolve key software development and validation challenges encountered by developers of large-scale DRE middleware and applications. The MDM analysis-guided composition and deployment of DRE applications helps to provide a verifiable and certifiable basis for ensuring the consistency and fidelity of DRE applications, such as those deployed in safety-critical domains like avionics control, medical devices, and automotive systems.

To illustrate recent progress on MDA technologies, this paper describes CoSMIC, which is an MDM toolsuite that combines the power of domain-specific

modeling, aspect-oriented domain modeling, mathematical analysis, generative programming, QoS-enabled component middleware, and run-time dynamic adaptation and resource management to resolve key challenges that occur throughout the DRE application lifecycle. CoSMIC currently provides platform-specific metamodels that address the packaging, middleware configuration, deployment planning and runtime QoS assurance challenges. The middleware platform we use to demonstrate our MDM R&D efforts is the Component-Integrated ACE ORB (CIAO) [5], which is QoS-enabled implementation of the CORBA Component Model (CCM). As other component middleware technologies mature to the point where they can support DRE applications, the CoSMIC tool-chain will be enhanced to support platform-independent models and their mappings to various platform-specific models.

All material presented in this paper is based on the CoSMIC MDM toolsuite available for download at `www.dre.vanderbilt.edu/cosmic`. The associated QoS-enabled component middleware platform CIAO can be downloaded from `www.dre.vanderbilt.edu/CIAO`.

References

[1] K. Ogata, Modern Control Engineering, Prentice Hall, Englewood Cliffs, NJ, 1997.

[2] D. C. Sharp, Reducing Avionics Software Cost Through Component Based Product Line Development, in: Proceedings of the 10th Annual Software Technology Conference, 1998.

[3] Joseph K. Cross and Patrick Lardieri, Proactive and Reactive Resource Reallocation in DoD DRE Systems, in: Proceedings of the OOPSLA 2001 Workshop "Towards Patterns and Pattern Languages for OO Distributed Real-time and Embedded Systems", 2001.

[4] A. S. Krishna, D. C. Schmidt, R. Klefstad, A. Corsaro, Real-time CORBA Middleware, in: Q. Mahmoud (Ed.), Middleware for Communications, Wiley and Sons, New York, 2003.

[5] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, C. D. Gill, QoS-enabled Middleware, in: Q. Mahmoud (Ed.), Middleware for Communications, Wiley and Sons, New York, 2003.

[6] D. C. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, L. DiPalma, Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems, CrossTalk.

[7] J. Sztipanovits, G. Karsai, Model-Integrated Computing, IEEE Computer 30 (4) (1997) 110–112.

[8] J. Gray, T. Bapty, S. Neema, Handling Crosscutting Constraints in Domain-Specific Modeling, Commun. ACM (2001) 87–93.

[9] Object Management Group, Model Driven Architecture (MDA), OMG Document ormsc/2001-07-01 Edition (Jul. 2001).

[10] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, G. Karsai, Composing Domain-Specific Design Environments, IEEE Computer.

[11] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, 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, 2003.

[12] G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty, Model-Integrated Development of Embedded Software, Proceedings of the IEEE 91 (1) (2003) 145–164.

[13] J. Gray, J. Sztipanovits, T. Bapty, S. Neema, A. Gokhale, D. C. Schmidt, Two-level Aspect Weaving to Support Evolution of Model-Based Software, in: R. Filman, T. Elrad, M. Aksit, S. Clarke (Eds.), Aspect-Oriented Software Development, Addison-Wesley, Reading, Massachusetts, 2003.

[14] A. Bondavalli, I. Mura, I. Majzik, Automated dependability analysis of UML designs, in: Proc. of Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 1998.

[15] S. Gokhale, J. R. Horgan, K. S. Trivedi, Integration of specification, simulation and dependability analysis, in: Workshop on Architecting Dependable Systems, Orlando, FL, 2002.

[16] S. Gokhale, "Cost–constrained reliability maximization of software systems", in: Proc. of Annual Reliability and Maintainability Symposium (RAMS 04) (To Appear), Los Angeles, CA, 2004.

[17] S. Gokhale, K. S. Trivedi, "Reliability prediction and sensitivity analysis based on software architecture", in: Proc. of Intl. Symposium on Software Reliability Engineering (ISSRE 02), Annapolis, MD, 2002.

[18] S. Neema, T. Bapty, J. Gray, A. Gokhale, Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems, in: Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, PA, 2002.

[19] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, M. Sirkin, The GenVoca Model of Software-System Generators, IEEE Software 11 (5) (1994) 89–94.

[20] I. Baxter, DMS: A Tool for Automating Software Quality Enhancement, Semantic Designs (www.semdesigns.com), 2001.

[21] T. A. Henzinger, S. Sastry (Eds.), Hybrid Systems: Computation and Control - Lecture Notes in Computer Science, Springer Verlag, New York, NY, 1998.

[22] L. M. Aeronautics, Lockheed Martin (MDA Success Story), www.omg.org/mda/mda_files/LockheedMartin.pdf (Jan. 2003).

[23] L. G. Networks, Optical Fiber Metropolitan Network, www.omg.org/mda/mda_files/LookingGlassN.pdf (Jan. 2003).

[24] A. Railways, Success Story OBB, www.omg.org/mda/mda_files/SuccessStory_OeBB.pdf/ (Jan. 2003).

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

[26] D. C. Sharp, 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, 2003.

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

[28] Object Management Group, Deployment and Configuration Adopted Submission, OMG Document ptc/03-07-08 Edition (Jul. 2003).

[29] Object Management Group, Unified Modeling Language: OCL version 2.0 Final Adopted Specification, OMG Document ptc/03-10-14 Edition (Oct. 2003).

[30] H. S. Thompson, D. Beech, M. Maloney, N. M. et al., XML Schema Part 1: Structures, W3C Recommendation (2001).
URL www.w3.org/TR/xmlschema-1/

[31] P. V. Biron, A. M. et al., XML Schema Part 2: Datatypes, W3C Recommendation (2001).
URL www.w3.org/TR/xmlschema-2/

[32] D. C. Schmidt, et al., TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems, IEEE Distributed Systems Online 3 (2).

[33] C. O'Ryan, D. C. Schmidt, J. R. Noseworthy, Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations, International Journal of Computer Systems Science and Engineering 17 (2).

[34] A. S. Krishna, N. Wang, B. Natarajan, A. Gokhale, D. C. Schmidt, G. Thaker, CCMPerf: A Benchmarking Tool for CORBA Component Model Implementations, in: Proceedings of the 10th Real-time Technology and Application Symposium (RTAS '04), IEEE, Toronto, CA, 2004.

[35] Object Management Group, Event Service Specification Version 1.1, OMG Document formal/01-03-01 Edition (Mar. 2001).

[36] Object Management Group, Notification Service Specification, Object Management Group, OMG Document telecom/99-07-01 Edition (Jul. 1999).

[37] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, B. Natarajan, Skoll: Distributed Continuous Quality Assurance, in: Proceedings of the 26th IEEE/ACM International Conference on Software Engineering, IEEE/ACM, Edinburgh, Scotland, 2004.

[38] A. S. Krishna, D. C. Schmidt, A. Porter, A. Memon, D. Sevilla-Ruiz, Improving the Quality of Performance-intensive Software via Model-integrated Distributed Continuous Quality Assurance, in: Proceedings of the 8th International Conference on Software Reuse, ACM/IEEE, Madrid, Spain, 2004.

[39] D. Harel, E. Gery, Executable Object Modeling with Statecharts, in: Proceedings of the 18th International Conference on Software Engineering, IEEE Computer Society Press, 1996, pp. 246–257.
URL `citeseer.nj.nec.com/article/harel97executable.html`

[40] M. Lin, Synthesis of Control Software in a Layered Architecture from Hybrid Automata, in: HSCC, 1999, pp. 152–164.
URL `citeseer.nj.nec.com/92172.html`

[41] J. T. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems, International Journal of Computer Simulation, Special Issue on Simulation Software Development Component Development Strategies 4.

[42] Object Management Group, Unified Modeling Language (UML) v1.4, OMG Document formal/2001-09-67 Edition (Sep. 2001).

[43] Extensible Markup Language (XML) 1.0 (Second Edition), `www.w3c.org/XML` (Oct. 2000).

[44] Object Management Group, UML Profile for CORBA, OMG Document formal/02-04-01 Edition (Apr. 2002).

[45] Object Management Group, UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Joint Revised Submission, OMG Document realtime/03-05-02 Edition (May 2003).

[46] Object Management Group, UML Profile for Schedulability, Final Draft OMG Document ptc/03-03-02 Edition (Mar. 2003).

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

[48] TimeSys, TimeSys Linux/RT 3.0, `www.timesys.com` (2001).

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

[50] Object Technology International, Inc., Eclipse Platform Technical Overview: White Paper, Object Technology International, Inc., Updated for 2.1, Original publication July 2001 Edition (Feb. 2003).

[51] Matthew Drahzal, Rose RealTime – A New Standard for RealTime Modeling: White Paper, Rational (IBM)., Rose Architect Summer Issue 1999 Edition (Jun. 1999).

[52] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, Boston, 2000.

[53] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, Lecture Notes in Computer Science 2072 (2001) 327–355. URL `citeseer.nj.nec.com/kiczales01overview.html`

[54] Olaf Spinczyk and Andreas Gal and Wolfgang Schröder-Preikschat, AspectC++: An Aspect-Oriented Extension to C++, in: Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), 2002.