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

Aniruddha Gokhale [a,*], Krishnakumar Balasubramanian [a,1], Arvind S. Krishna [a,1], Jaiganesh Balasubramanian [a], George Edwards [a,1], Gan Deng [a,1], Emre Turkay [a,1], 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 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: MDM: Model Driven Middleware, CCM: CORBA Component Model, D&C: Deployment and Configuration

⋆ Work supported by AFRL Contract#F33615-03-C-4112 for DARPA PCES Program, Raytheon and a grant from Siemens CT

* Corresponding Author Email: a.gokhale@vanderbilt.edu

[1] Author has since graduated from Vanderbilt University

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.

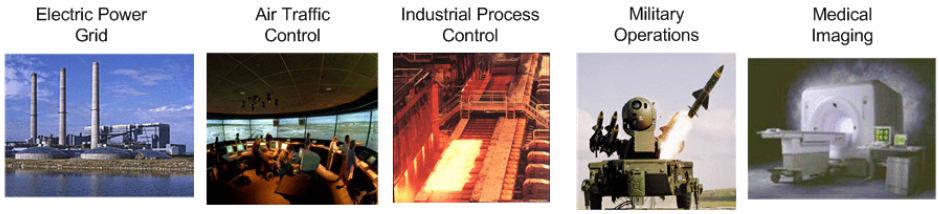

Fig. 1. **Example Large-scale Distributed Real-time and Embedded Systems**

These types of DRE systems share the following characteristics:
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 quality of service (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 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.

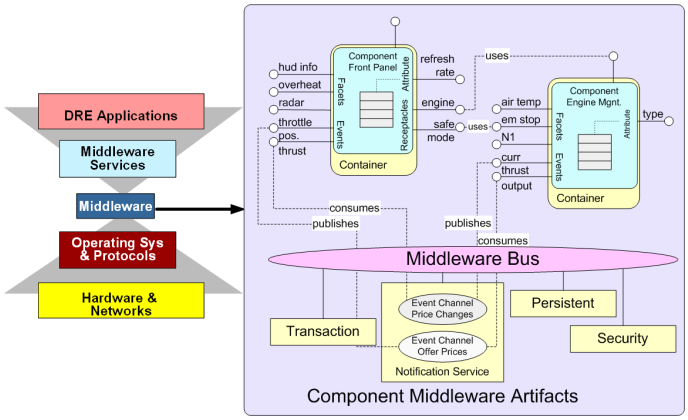

Fig. 2. **Component Middleware Layers and Architecture**

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:

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.

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 development 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 metadata, collect data from application runs, and analyze the collected data to re-synthesize the required metadata. 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:

a. Safeguarding DRE applications against technology obsolescence
b. Ensuring composition of valid DRE applications from sub-components
c. Choosing semantically compatible configuration options

d. Making effective deployment decisions based on target environment

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 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 middleware. 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.

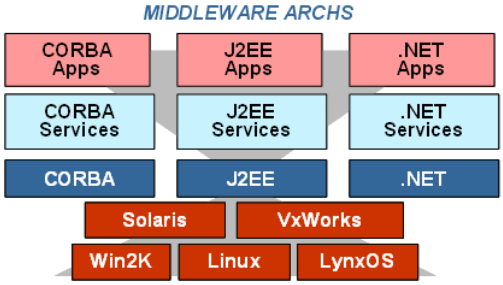

Fig. 3. **Popular Middleware Stacks**

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 a DRE application 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.

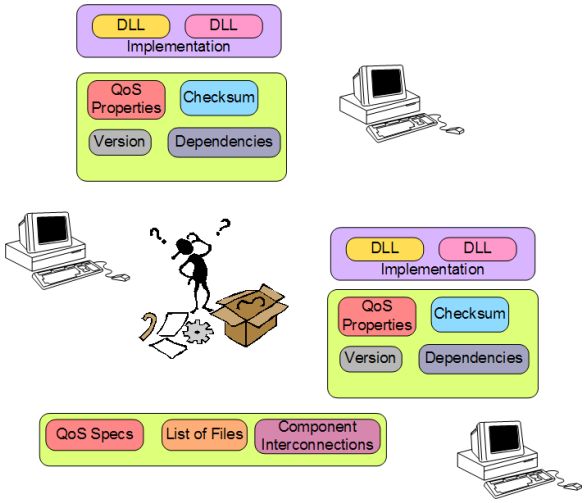

Fig. 4. **Application Composition Challenges**

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. In particular, our MDM approach provides the following capabilities:

- Creating models of the various components as black boxes that are part of an application
- Modeling the interconnections between the components
- Specifying systemic QoS properties of the components
- Building component assemblies *i.e.*, a set of components connected according to a well-defined specification that can be viewed and used as a single sub-component of a larger component and
- Bundling multiple component implementations into packages, which serve as the basic unit of composition in the MDM approach.

Moreover, 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 behavior. 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.

Moreover, *QoS-enabled* component middleware platforms are intended to leverage the benefits of component-based software development while simultaneously preserving the optimization patterns and principles of DOC middleware, such as its support for publisher/subscriber services. Before developers of event-based DRE systems can derive benefits from QoS-enabled component middleware, however, they must first reduce the complexity of configuring and deploying publisher/subscriber services. In particular, DRE system developers are faced with the following challenges when trying to use publisher/subscriber mechanisms provided by conventional component middleware:

a. **Configuring publisher/subscriber QoS**, where there are no standard means of configuring the component middleware mechanisms that can deliver appropriate QoS to DRE systems, and
b. **Deploying federated publisher/subscriber services**, where there are no standard policies and mechanisms to deploy a federation of publisher/subscriber services for DRE systems.

Problem – Inherent complexities in middleware configuration. In a large-scale DRE application, hundreds or thousands of components must be interconnected. 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.

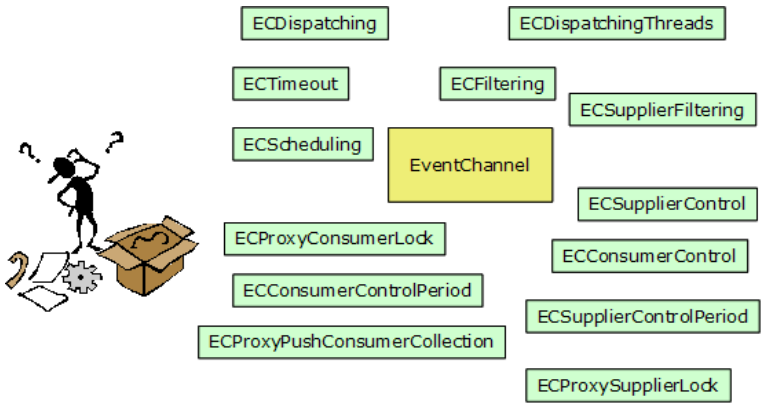

Fig. 5. **Middleware Configuration Challenges**

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 execution 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

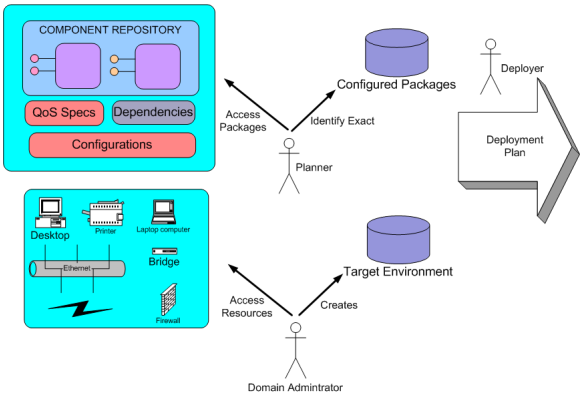

Fig. 6. **Planning for Deployment**

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 hard to do manually, *i.e.*, it is equivalent to resource allocation problems common in operations research, where given a finite set of resources, a finite set of tasks, and a set of constraints on allocation of resources to tasks, a solution that allocates resources to tasks that satisfies all the constraints is required. As the number of components and their target nodes increases, keeping track of the constraints manually is very time consuming, and in some cases 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 end-to-end QoS of components, and capabilities of the nodes in the given target environment [12,13]. 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.

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.

Figure 7 illustrates how in the context of DRE middleware and applications, MDM tools can be applied to:

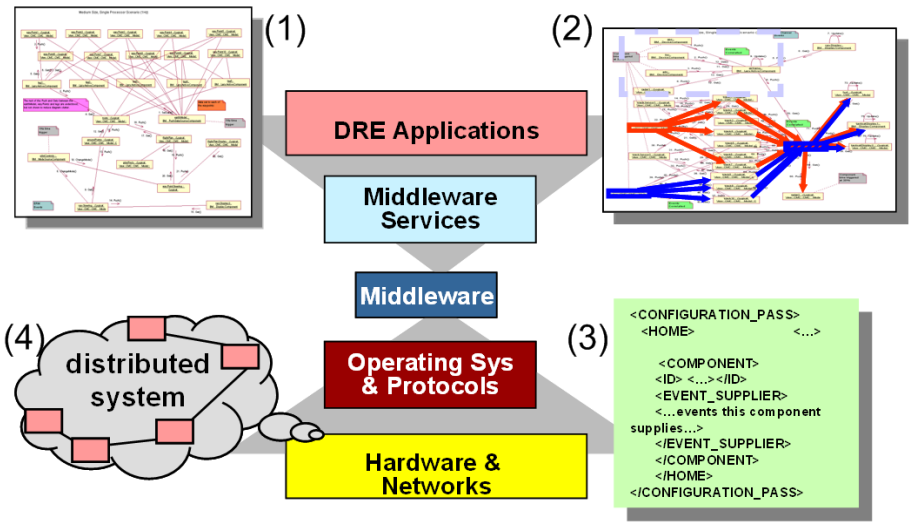

Fig. 7. **Model Driven Middleware Process**

• **Model** different functional and systemic properties of DRE systems via separate middleware- and platform-independent models [14]. Domain-specific aspect model weavers [15] 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 [16]. These tools can check whether the requested behavior and properties are feasible given the specified application and resource constraints. Tool-specific model analyzers [17,18] can also analyze the models and predict [19] 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 [20,21].
• **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 [22].

OMG MDA technologies initially focused largely on enterprise applications [23]. More recently, MDA technologies have emerged to customize QoS-enabled component middleware for DRE systems, including aerospace [24], telecommunications [25], and industrial process control [26]. 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 8, CoSMIC consists of an integrated collection of modeling, analysis, and synthesis tools that address key lifecycle challenges of DRE middleware and applications.

The CoSMIC MDM toolsuite provides the following capabilities:
– *Specification and implementation*, which enables application functionality specification, partitioning, and implementation as components.
– *Packaging*, which allows bundling a suite of software binary modules and metadata representing application components.
– *Installation*, which involves populating a repository with the packages required by the application.
– *Configuration*, which allows configuration of the packages with the appropriate parameters to satisfy the functional and systemic requirements of application without constraining to any physical resources.

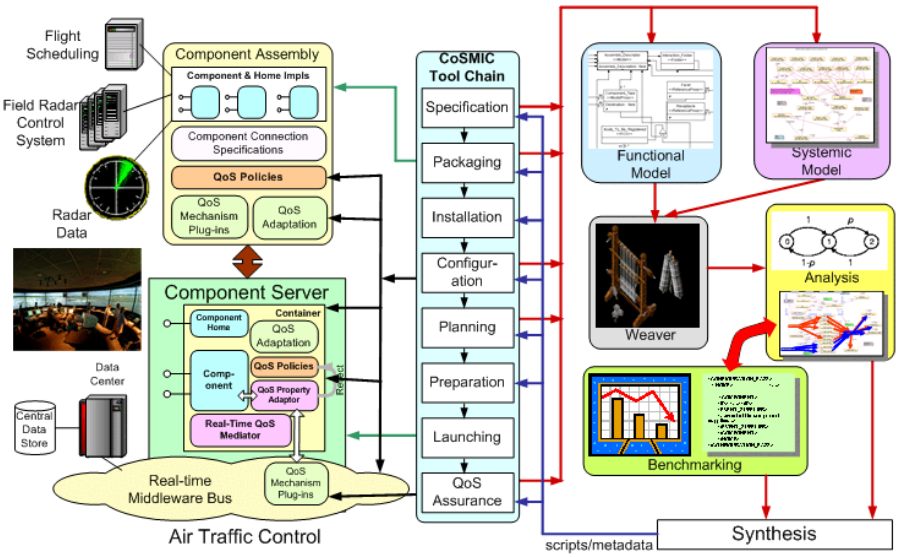

Fig. 8. **CoSMIC Model Driven Middleware Toolsuite**

– *Planning*, which makes appropriate deployment decisions including identifying the entities, such as CPUs, of the target environment where the packages will be deployed.
– *Preparation*, which moves the binaries to the identified entities of the target environment.
– *Launching*, which triggers the installed binaries and bringing the application to a ready state.
– *Adaptation*, which enables run-time reconfiguration and resource management to maintain end-to-end QoS.

The CoSMIC MDM toolsuite also provides the capability to interwork with third party model checking tools, such as Cadena [11], and aspect model weavers, such as C-SAW [27].

The CoSMIC tools are based on the Generic Modeling Environment (GME) [10], which is a metamodeling environment that defines the modeling paradigms [2] for each stage of the CoSMIC tool chain. These per-stage paradigms are aggregated within the context of multiple domain-specific modeling languages (DSMLs) [28] that are the cornerstone of CoSMIC. The CoSMIC DSMLs 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, source-level debuggers, 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 for use in the underlying middleware.

The CoSMIC toolsuite currently uses a *platform-specific model* (PSM) approach that integrates the modeling technology with our CIAO QoS-enabled component middleware [5]. The platform-specificity stems primarily from our model interpreters, which are targeted towards a single middleware platform. However, the modeling abstractions provided by the DSMLs are predominantly platform-independent. 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. This example demonstrates the middleware-based DRE system development challenges described in Section 2. We use this example to describe how the individual CoSMIC tools help address these challenges.

[2] A *modeling paradigm* defines the syntax and semantics of a modeling language [14].

3.1. *Demonstrating CoSMIC via Boeing Avionics Scenarios*

Our representative DRE system is drawn from the avionics mission computing domain. In particular, we chose a product scenario called *Basic Single Processor* (BasicSP) from the Boeing Bold Stroke component avionics mission computing product suite. Bold Stroke uses a *push event/pull data* publisher/subscriber communication paradigm [29] atop the Prism QoS-enabled component middleware platform [30].

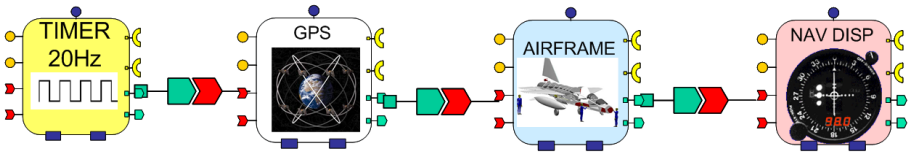

Fig. 9. **Navigation Display Collaboration Example**

As shown in Figure 9, BasicSP comprises four avionics mission computing components that are assembled to form a product line architecture where a navigation display simulation receives the global positions from a GPS device and displays them at a GUI display in a periodic manner that has stringent timing constraints. The desired data request and the display frequencies run at 20 Hz.

Figure 9 also shows the component interaction for the navigation display example. This scenario begins with the GPS being invoked by the TAO ORB's Real-time Event Service [31], shown as a `Timer` component. After receiving a pulse event from the `Timer`, the `GPS` generates its data and pushes a data available event to the airframe. TAO's Real-time Event Service then forwards the event on to the `Airframe` component, which pulls the data from 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 from the `GPS`, updates its state, and displays it.

The remainder of this section describes the design of individual tools of CoSMIC and illustrate how they resolve the deployment and configuration challenges of DRE systems in the context of the BasicSP representative scenario.

3.2. *Model-driven Component Packaging: Resolving Component Packaging Challenges*

The most important DSML provided by CoSMIC is the *Platform-independent Component Modeling Language* (PICML) [32]. PICML incorporates multiple individual tools by aggregating their modeling paradigms. One such tool is called the *COMPosable Adaptive Software Systems* (COMPASS) 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 aspects of the application, validate syntactic, semantic, and binary compatibility of the assembled components, and generate the systemic metadata as descriptors, as explained later in this section.

Figure 10 illustrates shows the sequence of steps involved in component packaging and application assembly. These steps include collecting information about properties and requirements of a single component, assembling a set of components into an application assembly (which satisfies the set of constraints that determines a valid assembly), and creating component packages from either the assembly created in the previous step or just from the information about individual components collected in the first step. Below we describe the key elements of the COMPASS tool from the perspective of the modeling paradigm, constraint specification, and model interpreter aspects. We also show how COMPASS can be applied to the BasicSP scenario described in Section 3.1.

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.

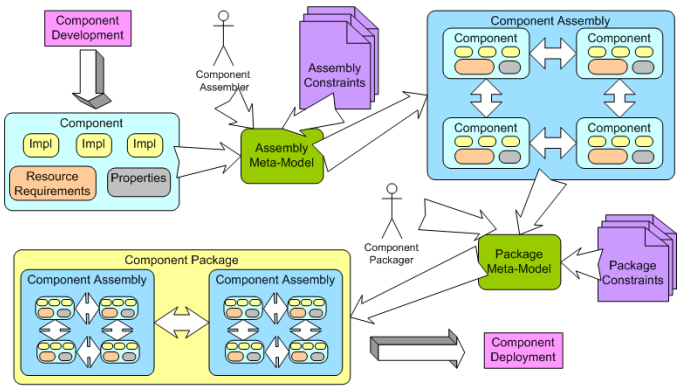

Fig. 10. **Component Packaging & Application Assembly**

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 [33], 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).

For example, the components of the BasicSP scenario (*i.e.*, `Timer`, `GPS`, `Airframe` and `Navdisplay`) can be modeled using COMPASS. Properties such as the name, unique identifiers for the components and the list of implementation artifacts that each component is composed of can be specified for each of these components. COMPASS also allows specification of the assembly information that describes the connections between components, *e.g.*, the connection between the `Timer` and `GPS` component can be captured in a COMPASS model.

3.2.2. *Constraint Specification*

COMPASS provides a constraint checker to ensure that the 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 metamodel using the Object Constraint Language (OCL) [34], 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 metamodel 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. For example, in the context of the scenario described in the Section 3.1, the constraints defined in COMPASS, will disallow connections between incompatible ports (such as the data ports of the `GPS` component and the control interface of the `Airframe` component) in the model.

3.2.3. *Model Interpretation*

The COMPASS model interpreter translates the various packaging and configuration information captured in the models constructed using its metamodel into a set of *descriptors*, which are files containing metadata 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 [35,36]. To ensure interoperability with other CoSMIC modeling tools, COMPASS synthesizes descriptors conforming to the XML schema defined by the OMG D&C specification [33], 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.

For the scenario described in Section 3.1, COMPASS generates XML descriptors that describe the connections between different components, a portion of which is shown below:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Deployment:ComponentImplementationDescription>
  <label>HUDDisplay Implementation</label>
  <UUID>effd4bd0-6db0-4c50-9bb7-db9decebae1c</UUID>
  <assemblyImpl>
    <instance xmi:id="a_GPS">
      <name>GPS</name>
      <reference href="GPS.cpd"/>
    </instance>
    <instance xmi:id="a_RateGen">
      <name>RateGen</name>
      <reference href="RateGen.cpd"/>
    </instance>
    ...
    <connection>
      <name>GPS-NavDisplay</name>
      <internalEndpoint>
        <portName>MyLocation</portName>
        <instance xmi:idref="a_NavDisplay"/>
      </internalEndpoint>
      <internalEndpoint>
        <portName>GPSLocation</portName>
        <instance xmi:idref="a_GPS"/>
      </internalEndpoint>
    </connection>
    ...
  </assemblyImpl>
</Deployment:ComponentImplementationDescription>
```

The XML fragment shown above describes an assembly that contains `RateGen`, `GPS`, `NavDisplay`, and `Timer` (not shown above) components, and describes the connections to be made among the ports of these components. This information is used by MDM tools in the planning stage to perform the actual connections between the components.

3.3. *Model-driven Middleware Configuration: Resolving Configuration Challenges*

CoSMIC provides two DSMLs and their associated tools to address the problem of multi-layer middleware configuration discussed in Challenge 3 of Section 2. The tools are the *Option Configuration Modeling Language* (OCML) [37,38] tool that handles ORB-level configurations and the *Event QoS Aspect Language* (EQAL) [39] tool that addresses container- and application-level configurations. We discuss each of these tools below from the perspective of the mod-

eling paradigm, constraint specification, and model interpreter aspects. We also show how OCML and EQAL can be applied to the BasicSP scenario described in Section 3.1.

3.3.1. *Modeling paradigms.*

The metamodel 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 two tools that help resolve middleware configuration challenges.

• **OCML.** To address the middleware level configuration challenges we have developed Option Configuration Modeling Language (OCML) tool [37]. OCML is a GME-based modeling paradigm for configuring QoS-enabled middleware and alleviating accidental complexities involved in this process. As shown in Figure 11 OCML is designed for use by both (1) *middleware developers*, who use OCML to define the constraints and dependencies of the middleware options, and (2) *application developers*, who use OCML and its constraints to specify semantically compatible middleware configuration options.

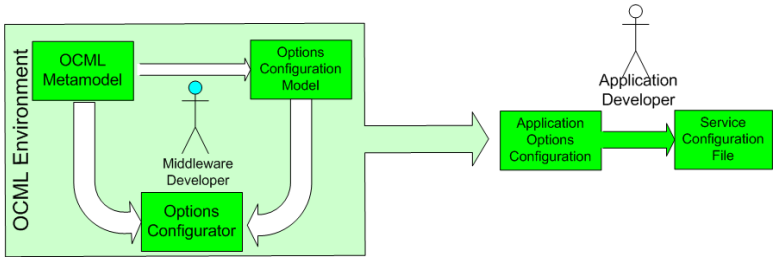

Fig. 11. **OCML Process**

The OCML language defines two different artifacts: (a) the *structure* artifact, which contains the hierarchical organization of the middleware configuration options a DRE system will require (*e.g.*, OCML has been used to model the configuration options provided by the TAO [40] ORB) and (b) the *rules* artifact, which constrains the available combination of these options. In the following we describe the artifacts defined by the OCML language.

In the *structure* artifact the middleware configuration options are arranged hierarchically within different option categories. An option category may include other option categories or may include options. Options are categorized according to the type of the values which they have (*e.g.* numeric, strict, enumerated, etc.) Using the *rules* artifact, rule categories can be defined, which represents the dependency information of certain options on other options. A rule definition element contains logical expressions. The operands of these logic expressions are both the references to the options which is modeled in the hierarchical option and other rules. Logical "and," "or," and "not" operations are provided as the operations.

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 11 the OCML tool is intended to be used by both middleware developers and application developers.

• **EQAL.** The Event QoS Aspect Language (EQAL) tool is designed to address container- and application-level configurations. The EQAL architecture is show in Figure 12. EQAL allows DRE system modelers to specify three types of CORBA event services: the OMG standard CORBA Event and Notification Services and the TAO's proprietary Real-time Event Service. These services allow components to asynchronously and anonymously send and receive customized data structures called events.

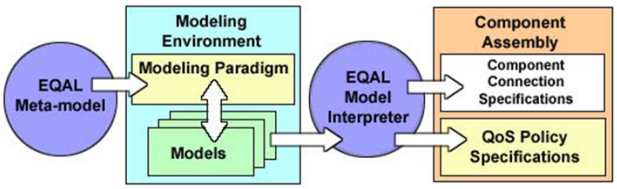

Fig. 12. **The Event QoS Aspect Language Architecture**

The EQAL modeling paradigm consists of two parts, the *configuration part* and the *deployment part*. For the *configuration part*, modelers can specify policies and strategies that include (but are not limited to) event filtering, event correlation, timeouts, locking, disconnect control, and priority. Each service policy can have different scopes, ranging from a single port to an entire event channel. EQAL's configurations can therefore be provisioned at the following three levels of granularity:

- **Channel scope**, which applies to all components using the channel. Each event channel must be specified with a number of policies that control its behavior. These policies control the way that the channel handles all connections and events.
- **Proxy scope**, which applies to a single component port. Each event port is associated with a proxy object. A number of QoS policies are configured at the proxy level. QoS parameters are provided for each connection by configuring the proxy. It follows that connection-level parameters must be coherent with channel-level policies.
- **Event scope**, which applies to an event instance. A limited number of QoS settings, such as timeout, can be specified for an individual event instance.

The EQAL modeling paradigm allows modelers to provision reusable and sharable configurations at each level of granularity outlined above. Modelers assign configurations to individual event connections and then construct filters for each connection. EQAL supports two forms of event generation using the push model: (1) a component may be an exclusive supplier of an event type or (2) a component may supply events to a shared channel.

To address the scalability problem in any large-scale event-based architecture, QoS-enabled component middleware such as CIAO provides event services that supports event channel federations. The federation could share filtering information to minimize or eliminate the transmission of unwanted events to a remote entity. Moreover, the 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 [41]. Connecting event channels from different systems together allows event information to be interchanged, providing a level of integration among the systems.

The EQAL *deployment part* specifies how components and event channels are assigned to hosts on a target network. For example, collocating a gateway with its consumer event channel (*i.e.*, the one it connects to as a supplier) eliminates the need to transmit events that are not subscribed to by the consumer event channel. Application developers can also choose different types of gateways based on different application deployment scenarios with different networking and computing resources. These deployment decisions have no coupling with, or bearing on, component application logic. The same set of components can therefore be reused and deployed into different scenarios without modifying application code manually.

The EQAL modeling paradigm allows three types of federations (*i.e.*, CORBA IIOP, UDP, or IP multicast) to be deployed. For event channel federation models, the EQAL modeling paradigm defines two levels of syntactic elements:

- The **outer-level**, which contains the host elements as basic building blocks and allows users to define the hosts present in the DRE system and
- The **inner-level**, which represents a host containing a set of syntactic elements (including event channels, CORBA IIOP gateways, UDP senders and receivers, IP multicast senders and receivers, and event type references) that allow users to configure the deployment of these artifacts inside a host.

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 accidental complexity in component middleware. One of CoSMIC's primary benefits is the prevention of inconsistent combinations of QoS parameters during modeling time through constraint checking. Constraints ensure that only valid models can be constructed and interpreted. This section describes the constraint checkers in the OCML and EQAL tools that ensure compatibility and validity of configuration options.

• **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. For example, TAO ORB developers use OCML to define rules that constrain the permissible combinations of ORB level configuration options. DRE system developers are thus constrained to use only valid

combinations of configuration parameters for their applications.
• **EQAL.** EQAL automatically verifies the validity of different types 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. Also, 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 tools provides model interpreters that synthesize the target middleware configuration files and component descriptor files. Below we describe the model interpreters in the OCML and EQAL tools, which support the synthesis of configuration metadata for the TAO ORB and CIAO component middleware that uses the TAO ORB.
• **OCML.** The middleware-specific options configuration language is validated against the OCML metamodel 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 11.
• **EQAL.** EQAL encompasses two model interpreters. The first interpreter generates XML descriptor files that conform to the Boeing Prism XML schema for Event Services component configuration. These descriptor files identify the real-time requirements of individual connections and event channel federations. 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. The information captured in the descriptor files include the relationship between each artifacts, the physical location of each supplier, consumer, event channel and CIAO Gateway.

3.3.4. *Resolving Middleware Configuration Challenges for BasicSP Scenario using OCML and EQAL*

In this section we demonstrate how the middleware configuration challenges are resolved for the BasicSP application scenario using OCML and EQAL tools.
• **OCML.** As discussed in the previous section a model for TAO ORB Configuration Options is designed within the GME modeling environment using the OCML modeling language and interpreted to generate the TAO specific Configuration File Generator application. The Configuration File Generator is used to configure middleware for each component of the BasicSP scenario. For example, the OCML tool is used to configure ORB level configuration options including server concurrency mechanism, protocol factories so that BasicSP can operate over distinct transport protocols, and real-time CORBA threading policies so that these can support the 20Hz periodic real-time requirements of BasicSP.
• **EQAL.** The BasicSP components transmit and receive events at specified real-time rates. Consequently, the event propagation mechanism must be capable of delivering events in a timely manner. For the BasicSP, therefore, we use EQAL to configure and deploy the TAO real-time event service that can provide guaranteed timely delivery of events. Using EQAL, a developer or deployer of the BasicSP application can rapidly specify the timing requirements (20Hz in our case) for each component, while being assured of semantic compatibility among event dependencies.

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

As part of the PICML DSML, CoSMIC provides two additional tools: the *Model Integrated Deployment and Configuration Environment for Composable Software Systems* (MIDCESS) and the *CCM Performance* (CCMPerf)

tools [42] 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:

a. **Nodes**, where the individual components and component packages are loaded and used to instantiate those components.
b. **Interconnects** among nodes, to which inter-component software connections are mapped, to allow the instantiated components to intercommunicate.
c. **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. In the context of the BasicSP scenario described in Section 3.1, various components need to collaborate to complete the GPS application. Compatible component implementations of GPS and airframe functionality need to be chosen depending on the target environment specified. For example, components may be implemented in different programming languages because the target environment in which these components will execute may not be known at design-time. MIDCESS helps in this deployment planning process by specifying the target environment in which these components will execute.

Using the target environment information available from MIDCESS, CCMPerf [42] can then be used to synthesize 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 given 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 thereby guaranteeing the 20Hz rate end to end. The experiments in CCMPerf can be divided into the following three experimentation categories:

a. *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,
b. *Common middleware services* tests that quantify the suitability of using different implementations of CORBA services, such as using Real-time Event [43] service against the Notification Services [44] for delivering periodic trigger updates to the `Nav_Display` component, and
c. *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.

Figure 13 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. We discuss each of these tools below from the perspective of the modeling paradigm, constraint specification, and model interpreter aspects. We also show how MIDCESS and CCMPerf can be applied to the BasicSP scenario described in Section 3.1.

3.4.1. *Modeling Paradigm*

This section describes the modeling paradigm supported by the MIDCESS and CCMPerf tools.

• **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

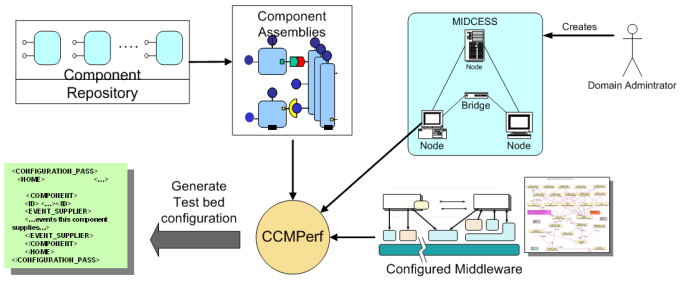

Fig. 13. **Deployment Planning Process**

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:

a. Specification of node elements and the interconnections between the node elements, *e.g.*, specifying the node that will host the `Nav_Display` component and how it will be connected to other nodes hosting other components, *e.g.*, `Airframe`.
b. Specification of the attributes of each of the nodes, *e.g.*, specifying the name of the node.
c. Hierarchical modeling of the individual nodes that share certain basic attributes (such as their type), but vary in the processing power, supported OS etc.
d. Hierarchical modeling of the interconnects to specify the different varieties of connections possible in the target environment.
e. 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

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

3.4.2. *Constraints Specification*

This section describes the constraint checking capabilities of MIDCESS and CCMPerf.

• **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) [34], 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. For example, MIDCESS will flag an error if the binary format of a component does not match the target node's supported format. Similarly, it can flag errors if the underlying network cannot support the bandwidth requirements.

• **CCMPerf** also contains a constraint checker that validates the experiment to preclude invalid configuration, such as (1) *conflicting metrics*, such as using both back box and white box metrics in a given experiment, (2) *invalid connections*, such as not connecting a required interface with the corresponding provides interface (*e.g.*, in the BasicSP

scenario this constraint violation corresponds to connecting `Nav_Display` ports directly to the `GPS` component instead of the `Airframe` component), and (3) *incompatible exchange format*, such as connecting a point-to-point entity with a point-to-multipoint entity, *e.g.*, connecting `Timer` refreshes (events) to position updates generated for the `Nav_Display` ("pull" operations). Constraints are defined in the CCMPerf meta model are defined using OCL [34]. The use of constraints ensure that the experiment is correct *a priori* minimizing errors at run-time.

3.4.3. *Model Interpretation*

This section describes the artifacts of the model interpretation process in MIDCESS and CCMPerf.

• **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* [33]. The output of MIDCESS can therefore be validated by running the descriptor through a tool that supports XML schema validation. The generated descriptor is then used by the CIAO deployment run-time infrastructure, which uses information in the planning descriptor to make deployment decisions.

• **CCMPerf** 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 MIDCESS and CCMPerf tools described above. For example, providing application developer with QoS metrics (such as latency, throughput, and jitter) for the scenario on a target platform at design-time helps them make intelligent decisions on mapping components to appropriate nodes in the domain. To achieve this goal, developers can use CCMPerf to compose a representative test application to be run of a target environment modeled using MIDCESS, and associate certain QoS requirements, such as minimizing latency. Figures 14 and 15 show how the component interaction and QoS association can be modeled using CCMPerf.

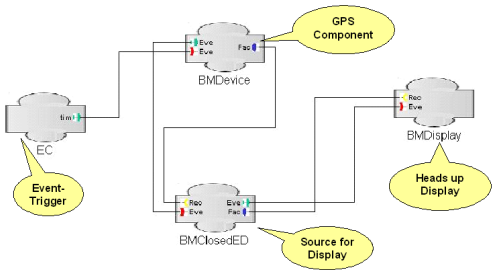

Fig. 14. **Modeling Component Interaction using CCMPerf**

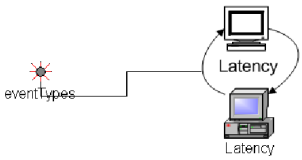

Fig. 15. **Associating QoS Attributes with BasicSP Scenario**

The following steps help resolve configuration and deployment challenges for DRE applications:

a. Determine the QoS expected from the middleware.
b. 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 middleware developers will have the appropriate insights to select the right options.
c. Use CCMPerf to generate a testsuite for evaluating QoS delivered by the middleware. The CCMPerf interpreter will generate the scaffolding code required to set up, run and tear down the experiment.
d. Use MIDCESS to model the target configuration and synthesize the necessary descriptors for component deployment.
e. For each configuration option discussed in step 4, run the generated benchmarking tests to evaluate the QoS. [3]

[3] The challenges arising from the explosion in the configuration space can be alleviated using pruning techniques discussed in research [45,46].

f. 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, 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) [7,47,48,8] 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 [49] (used primarily in the real-time and embedded domain) and MDA [9] based on UML [50] and XML [51] (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 leverages 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 [52], the UML profile for quality of service [53], and UML profile for schedulability, performance and time [54]) to support DRE applications. Moreover, CoSMIC applies the MIC principles to large-scale network-centric DRE systems as opposed to standalone embedded platforms restricted to digital signal processors.

The Virginia Embedded System Toolkit (VEST) [55] 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) [56]. 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.

In this regard the Cadena effort is complementary to the CoSMIC effort since the former can be used to model-check properties of the system modeled in CoSMIC. We have used the Open Tool Integration Framework (OTIF) [57] to build model translators that will allow our CoSMIC models to communicate with Cadena and VEST thereby leveraging their model checking capabilities for validating properties, such as end-to-end rates in a component assembly [58].

Commercial successes in model-based software development include the Rational Rose [59] 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 [22] 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 [22]. 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) [60] 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 [21] 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 [61] and AspectC++ [62].

CoSMIC has been developed to interwork with model-level aspect weaving tools like C-SAW [15] to weave into models crosscutting properties like host assignment.

5. Concluding Remarks

Large-scale distributed real-time and embedded (DRE) systems 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 by applying 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 middleware 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.

The CoSMIC MDM toolsuite is 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: Software Product Lines: Experience and Research Directions, Vol. 576, 2000, pp. 353–370.

[3] J. K. Cross, P. 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, 2004, pp. 413–438.

[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, 2004, pp. 131–162.

[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 - The Journal of Defense Software Engineering (2001) 10–16.

[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, Communications of the ACM (2001) 87–93.

[9] Object Management Group, Model Driven Architecture (MDA) Guide V1.0.1, OMG Document omg/03-06-01 Edition (Jun. 2001).

[10] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, G. Karsai, Composing Domain-Specific Design Environments, IEEE Computer (2001) 44–51.

[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, pp. 160–172.

[12] R. Rajkumar, C. Lee, J. P. Lehoczky, D. P. Siewiorek, Practical Solutions for QoS-based Resource Allocation Problems, in: IEEE Real-time Systems Symposium (RTSS 98), IEEE, Madrid, Spain, 1998, pp. 296–306.

[13] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, J. Hansen, A Scalable Solution to the Multi-Resource QoS Problem, in: Proceedings of the IEEE Real-time Systems Symposium (RTSS 99), Phoenix, AZ, 1999, pp. 315–326.

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

[15] 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, 2004, pp. 681–709.

[16] A. Bondavalli, I. Mura, I. Majzik, Automated Dependability Analysis of UML Designs, in: Proceedings of the Second IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC 99), 1999, pp. 139–144.

[17] S. Gokhale, J. R. Horgan, K. S. Trivedi, Integration of Specification, Simulation and Dependability analysis, Workshop on Architecting Dependable Systems (May 2002).

[18] S. Gokhale, "Cost–constrained reliability maximization of software systems", in: Proc. of Annual Reliability and Maintainability Symposium (RAMS 04), Los Angeles, CA, 2004, pp. 195–200.

[19] 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, pp. 64–75.

[20] 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, pp. 236–251.

[21] 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.

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

[23] P. Allen, Model Driven Architecture, Component Development Strategies 12 (1).

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

[25] Looking Glass Networks, Optical Fiber Metropolitan Network, `www.omg.org/mda/mda_files/LookingGlassN.pdf` (Jan. 2003).

[26] Austrian Railways, Success Story OBB, www.omg.org/mda/mda_files/SuccessStory_OeBB.pdf/ (Jan. 2003).
[27] J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. Gokhale, B. Natarajan, An Approach for Supporting Aspect-Oriented Domain Modeling, in: Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE'03), Erfurt, Germany, 2003, pp. 151–168.
[28] J. Gray, J. Tolvanen, S. Kelly, A. Gokhale, S. Neema, J. Sprinkle, Domain-Specific Modeling, in: CRC Handbook on Dynamic System Modeling, (Paul Fishwick, ed.), CRC Press, 2007, pp. 7.1–7.20.
[29] D. C. Sharp, Avionics Product Line Software Architecture Flow Policies, in: Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC), 1999, pp. 9.C.4–1–9.C.4–8.
[30] D. C. Sharp, W. C. Roll, Model-Based Integration of Reusable Component-Based Avionics System, Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003 (May 2003).
[31] 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, Atlanta, GA, 1997, pp. 184–199.
[32] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, D. C. Schmidt, A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems, in: RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium, Los Alamitos, CA, USA, 2005, pp. 190–199.
[33] OMG, Deployment and Configuration of Component-based Distributed Applications, v4.0, Document formal/2006-04-02 Edition (Apr. 2006).
[34] Object Management Group, Unified Modeling Language: OCL version 2.0 Final Adopted Specification, OMG Document ptc/03-10-14 Edition (Oct. 2003).
[35] 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/
[36] P. V. Biron, A. M. et al., XML Schema Part 2: Datatypes, W3C Recommendation (2001).
URL www.w3.org/TR/xmlschema-2/
[37] E. Turkay, A. Gokhale, B. Natarajan, Addressing the Middleware Configuration Challenges using Model-based Techniques, in: Proceedings of the 42nd Annual Southeast Conference, ACM, Huntsville, AL, 2004, pp. 166–170.
[38] A. S. Krishna, E. Turkay, A. Gokhale, D. C. Schmidt, Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems, in: Proceedings of the 11th Real-time Technology and Application Symposium (RTAS '05), IEEE, San Francisco, CA, 2005, pp. 180–189.
[39] G. Edwards, G. Deng, D. C. Schmidt, A. Gokhale, B. Natarajan, Model-driven Configuration and Deployment of Component Middleware Publisher/Subscriber Services, in: Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE), Vancouver, CA, 2004, pp. 337–360.
[40] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, C. Gill, TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems, IEEE Distributed Systems Online 3 (2).
[41] 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) (2002) 115–132.
[42] 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, pp. 140–147.
[43] Object Management Group, Event Service Specification Version 1.1, OMG Document formal/01-03-01 Edition (Mar. 2001).
[44] Object Management Group, Notification Service Specification, Object Management Group, OMG Document telecom/99-07-01 Edition (Jul. 1999).
[45] 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, Edinburgh, Scotland, 2004, pp. 459–468.
[46] A. S. Krishna, D. C. Schmidt, A. Porter, A. Memon, D. Sevilla-Ruiz, Validating Quality of Service for Reusable Software via Model-integrated Distributed Continuous Quality Assurance, in: Proceedings of the 8th International Conference on Software Reuse, ACM/IEEE, Madrid, Spain, 2004, pp. 286–295.
[47] 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
[48] 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
[49] 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.
[50] Object Management Group, Unified Modeling Language (UML) v1.4, OMG Document formal/2001-09-67 Edition (Sep. 2001).
[51] Extensible Markup Language (XML) 1.0 (Second Edition), www.w3c.org/XML (Oct. 2000).
[52] Object Management Group, UML Profile for CORBA, OMG Document formal/02-04-01 Edition (Apr. 2002).
[53] 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).
[54] Object Management Group, UML Profile for Schedulability, Final Draft OMG Document ptc/03-03-02 Edition (Mar. 2003).
[55] 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: RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, Washington, DC, USA, 2003, p. 58.
[56] 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).

[57] Institute for Software Integrated Systems, Open Tool Integration Framework, www.isis.vanderbilt.edu/Projects/WOTIF/.

[58] G. Trombetti, A. Gokhale, D. C. Schmidt, J. Hatcliff, G. Singh, J. Greenwald, An Integrated Model-driven Development Environment for Composing and Validating Distributed Real-time and Embedded Systems, in: S. Beydeda, M. Book, V. Gruhn (Eds.), Model Driven Software Development- Volume II of Research and Practice in Software Engineering, Springer-Verlag, New York, 2005, pp. 329–361.

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

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

[61] 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

[62] O. Spinczyk, A. Gal, W. 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), Sydney, Australia, 2002, pp. 53–60.