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

Krishnakumar Balasubramanian [a], Jaiganesh Balasubramanian [a], Arvind S. Krishna [a], George Edwards [a], Gan Deng [a], Emre Turkay [a], Jeffrey Parsons [a], Aniruddha Gokhale [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) applications 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). DRE applications are increasingly composed of multiple systems that are interconnected via wireless and wireline networks to form systems of systems. A challenging requirement for DRE applications 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 a growing number of DRE applications 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 application developers. For example, substantial time and effort is spent retrofitting DRE applications when their COTS technology infrastructure changes.

This paper provides three contributions to improving the development and validation of DRE applications throughout their lifecycles. First, we illustrate the challenges in developing and deploying QoS-enabled component middleware-based DRE applications and outline our solution approach to resolve these challenges. Second, we describe a new software paradigm called Model Driven Middleware (MDM) that combines model-based software development techniques with QoS-enabled component middleware to address key challenges faced by developers of DRE applications - particularly composition, integration, and assured QoS for end-to-end operations. Finally, we describe our progress on a MDM tool-chain, called CoSMIC that addresses key DRE application and middleware lifecycle challenges, including developing component functionality, partitioning the components to use distributed resources effectively, validating the software, assuring multiple simultaneous QoS

Preprint submitted to Science of Computer Programming *14 November 2003*

properties in real-time, and safeguarding against rapidly changing technology.

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

---

1 Introduction

1.1 Emerging Trends

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

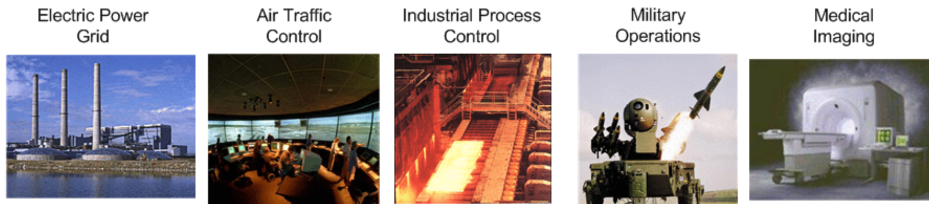

Fig. 1. **Example Large-scale Distributed Real-time and Embedded Applications**

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

2. Deeply embedded properties. DRE applications are frequently composed of multiple embedded subsystems. For example, an antilock 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 applications. These controllers introduce many challenging – often simultaneous – QoS constraints, including (1) *real-time*

⋆ Work supported by AFRL Contract#F33615-03-C-4112 for DARPA PCES Program

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

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 applications makes it infeasible to deploy them in disconnected, standalone configurations. The functionality of DRE applications is therefore partitioned and distributed over a range of networks. For example, an urban bioterrorist evacuation capability requires highly distributed functionality involving networks connecting command and control centers with biosensors that collect data from police, hospitals, and urban traffic management systems.

5. Dynamic operating conditions. Operating conditions for large-scale DRE applications can change dynamically resulting in the need for appropriate adaptation and resource management strategies for continued successful system operation. In civilian contexts, for instance, the recent power grid failure in the northeastern United States underscores 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 applications described above has grown significantly, DRE software remains harder to develop, maintain, and evolve [2,3] than mainstream desktop and enterprise software due in large part to their reliance on proprietary hardware and software technologies and development techniques. Unfortunately, proprietary solutions often fail to address the needs of large-scale DRE applications over their extended lifecycles. As DRE applications grow in size and complexity, moreover, 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 challenges outlined in the previous paragraph. As shown in Figure 2, middleware is systems software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware and provides the following capabilities:

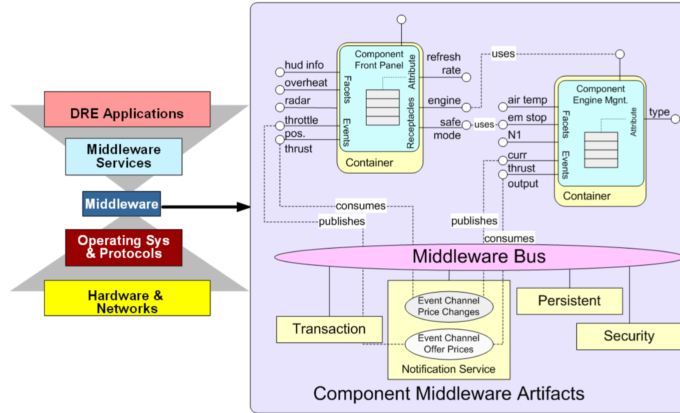

Fig. 2. **Component Middleware Layers and Architecture**

1. Control over key end-to-end QoS properties. A hallmark of DRE applications 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 applications.

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 *container* 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 implementation. By reusing the middleware's communication mechanisms and containers, developers of DRE applications 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 the significant advances in standards-based QoS-enabled component middleware, however, there remain significant technology gaps to effectively support large-scale DRE applications 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. 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.

2. Effective composition of DRE application components. DRE component middleware enable application developers to develop standalone QoS-enabled components that can be composed together into *assemblies* to form semi-complete DRE applications. Although this approach provides the ability to use "plug&play" components into DRE applications, system integrators must now face the daunting task of composing the right set of compatible components that will deliver the desired semantics and QoS to an entire application.

3. Accidental complexities in configuring and deploying middleware. In QoS-enabled component middleware, both the components and the underlying component middleware framework may have a large number of configurable attributes and parameters that can be set at various stages of development lifecycle, such as composing an application or deploying an application in a specific environment. It is tedious and error-prone, however, to manually ensure that all these parameters are semantically consistent throughout an application. Moreover, such *ad hoc* approaches have no formal basis for validating and verifying that the configured middleware will indeed deliver the end-to-end QoS requirements of the application.

4. Effective deployment decisions of DRE component assemblies on heterogeneous target platforms. The component assemblies described in bullet 2 above must be deployed in the distributed target environment before an application can start running. Application deployers must therefore perform the complex task of selecting from amongst a range of component assemblies and deploy them on the right entities in a 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 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.

1.4 Paper Organization

The remainder of paper is organized as follows: Section 2 describes key R&D challenges associated with large-scale DRE applications and outlines how the Model Driven Middleware 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 DRE Application R&D Challenges and Resolutions

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

(1) Safeguarding DRE applications against technology obsolescence
(2) Packaging of component functionality
(3) Configuration of middleware
(4) Planning for deployment

For each of these challenges we describe the context in which they arise, the specific technology problems that need to be solved, and outline how Model Driven Middleware (MDM) can be applied to help resolve these problems. 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).

2.1 Challenge 1 - Safeguarding DRE Applications Against Technology Obsolescence

• **Context.** Component middleware refactors what was often historically *ad hoc* application functionality into reusable, composable, and configurable standalone 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. The goal is to provide different implementations that are tailored 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 multitude of middleware stacks, such as those shown in Figure 3. This heterogeneity often makes

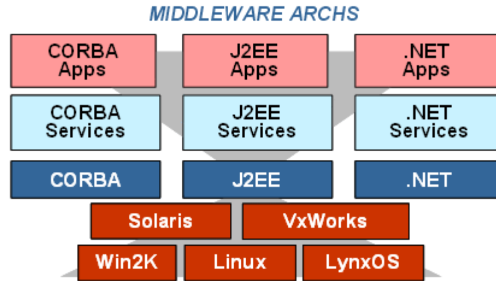

Fig. 3. **Multiple Middleware Stacks**

it hard, however, to identify the right middleware for a given application domain. Moreover, there exist limitations on how much application code can be factored out as reusable patterns and components in various layers for each middleware stack. This limit on refactoring in turn affects the optimization possibilities that can be implemented in different layers of the middleware. A challenge for DRE application developers is therefore to choose the right middleware technology that can provide the desired levels of end-to-end QoS. It is also desirable to safeguard applications from technology obsolescence so that the application software can incorporate newer technologies with minimal effort.

• **Solution approach.** One way to safeguard DRE applications from technology obsolescence is to apply the MDM paradigm to separately model the functional and systemic (*i.e.*, QoS) requirements of components at higher levels of abstraction than that provided by conventional programming languages or scripting tools. MDM analysis and synthesis tools can then make suitable decisions on the choice of the appropriate middleware platform and its configuration and deployment for the target environment. Section 3 describes the architecture of CoSMIC, which is an integrated suite of MDM tools we are developing to address this challenge.

2.2 Challenge 2 – Packaging of Component Functionality

•**Context.** As illustrated in Figure 4, *packaging* involves bundling a suite of software binary modules and metadata representing application components, where a component can be monolithic (standalone) or an assembly of subcomponents. Packaged components exist in "passive mode," *i.e.*, all their functionality is present, but they are inert object code. To perform their functionality at run-time, components must transition to "active mode," where the interconnections between components are established. Deployment mechanisms are responsible for transitioning components from passive to active mode.

• **Problem – Accidental complexities in composing and integrating software systems.** Composing an application from a set of components with syntactically consistent interface signatures simply ensures they can be connected together. To function correctly, however, collaborating components

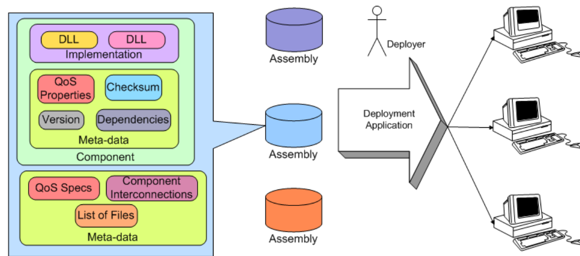

Fig. 4. **Packaging of components**

must also have compatible semantics and invocation protocols, which are hard to express via interface signatures alone. For example, if a component developer has provided different implementations of the same functionality, it is necessary to be able to assemble components that are semantically and binary compatible with each other. It is also essential that the assembled packages provide the desired systemic (QoS) properties.

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

• **Solution approach.** Our approach to addressing Challenge 2 involves developing MDM tools to represent component assemblies using the modeling techniques described in Section 3.1. These models are amenable to model checking [11], which in turn can ensure semantic and binary compatibility.

2.3 Challenge 3 – Configuration of Middleware

• **Context.** Assuming a suitable component packaging capability exists, the next challenge involves configuring packages in accordance with the appropriate configuration parameters of the middleware platform. Configuration of middleware 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, and data marshaling strategies. These parameters can be set at various stages of development lifecycle, such as during composing an application or deploying an application in a specific environment. The goal is to satisfy the functional and systemic requirements of DRE applications by taking into

account the properties of the target environment, without prematurely committing to physical resources, such as a specific host machine or choice of a network.

• **Problem – Accidental complexities in configuring middleware.** It is tedious and error-prone to manually ensure that all the configurable parameters provided by the middleware are semantically consistent throughout an application. Moreover, such *ad hoc* approaches have no formal basis for validating and verifying that the configured middleware will indeed deliver the end-to-end QoS requirements of the application.

• **Solution approach.** To address Challenge 3, we are 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 3.2 describes MDM tools that help ensure the configuration parameters at different layers of the middleware are compatible with each other.

2.4 Challenge 4 – Planning for Deployment

• **Context.** DRE applications often possess multiple QoS requirements that the middleware must help to enforce simultaneously on the target platform. This enforcement process involves both planning and preparing for the deployment of components and their assemblies. Planning involves decisions specifying the target environment, making appropriate component deployment decisions, such as identifying the packages that will be deployed in the hosts specified in the target environment. After these decisions are made, the preparation activity involves moving the selected package binaries to the specified entities of the target environment and triggering the necessary scripts to launch the application.

• **Problem – Satisfying multiple QoS requirements simultaneously.** Due to the uniqueness and complexity of DRE application QoS requirements, the heterogeneity of the environments in which they are deployed, and the need to interface with legacy systems and data, it is hard to develop a universal middleware solution and deployment decision that can address these requirements. It is also hard to integrate highly configurable, flexible, and optimized components from different providers while still ensuring that application QoS requirements are delivered end-to-end. Due to the functional and systemic partitioning of these DRE applications, it is therefore necessary to have a carefully choreographed sequence of deployment planning steps that will ensure that functional dependencies are met and the systemic requirements are satisfied after deployment.

• **Solution approach.** To address challenge 4, our solution approach described in Section 3.3 requires developing MDM tools that accurately depict the target environment and that determine how appropriate deployment decisions can be made based on an analysis of expected end-to-end QoS of packages

to be deployed in a given target environment. For example, the target environment modeling includes the network topology, the network technology and the available bandwidth, the CPUs, the OS they run and the size of RAM, among others that are used to make the right deployment decisions. Moreover, as described in Section 3.3, the models of target environment when combined with the models of the packages are used to synthesize test suites that are customized to benchmark different aspects of DRE application and component middleware performance. Empirical benchmark data for individual assemblies is used in end-to-end QoS prediction analysis tools to guide the deployment of components throughout a distributed 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 DRE application and middleware software. These methods must enforce the physical constraints of the system. Moreover, they must satisfy stringent functional and systemic QoS requirements within an entire system. Achieving these goals requires a set of standard integrated 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. These tools must be able to analyze the requirements and synthesize the required metadata that will compose applications from the right set of middleware components.

A promising way to address the DRE application lifecycles challenges described in Section 2 is *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. 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). In the context of DRE middleware and applications, MDM tools can be applied to:
• **Modeling** different functional and systemic properties of DRE applications in separate middleware- and platform-independent models [12]. Domain-specific aspect model weavers [13] can integrate these different models into composite models that can be further refined by incorporating middleware and platform-specific properties.
• **Analyzing** different—but interdependent—characteristics and requirements of application behavior specified in the models, such as scalability, predictability, safety, schedulability, and security. Model interpreters [10] translate the

information specified by models into the input format expected by model checking [11] and analysis tools [14]. These tools can check whether the requested behavior and properties are feasible given the specified application and resource constraints. Tool-specific model analyzers [15,16] can also analyze the models and predict [17] expected end-to-end QoS of the constrained models.

• **Synthesizing** [18,19] platform-specific code and metadata that is customized for a particular component middleware and DRE application properties, such as end-to-end timing deadlines, recovery strategies to handle various runtime failures in real-time, and authentication and authorization strategies modeled at a higher level of abstraction.

• **Provisioning** 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 automated invasive changes to existing components in order to provide the hooks that will adapt to the metadata. These invasive changes can be instrumented using a program transformation system, such as DMS [20].

• **Assuring** runtime QoS properties are delivered to the DRE application, which can be achieved via modeling dynamic adaptation and resource management strategies that use hybrid control-theoretic [21] techniques.

Initially, OMG MDA technologies focused largely on enterprise applications. More recently, MDA technologies have emerged to customize QoS-enabled component middleware for DRE applications, including aerospace [22], telecommunications [23], and industrial process control [24]. This section describes our R&D efforts that are integrating the MDA paradigm with QoS-enabled component middleware to create a Model Driven Middleware toolsuite called CoSMIC (Component Synthesis using Model Integrated Computing). As shown in Figure 5, the CoSMIC toolsuite consists of an integrated collection of modeling, analysis, and synthesis tools addressing key lifecycle challenges of DRE applications and middleware. The CoSMIC tools are all based on the Generic Modeling Environment (GME) [10], which is a meta-modeling environment that defines the modeling paradigms for each stage of the CoSMIC tool chain. The CoSMIC tools use GME to enforce their "correct by construction" techniques, as opposed to the "construct by correction" techniques commonly used by post-construction tools such as compilers and script validators. CoSMIC ensures that the rules of construction – and the models constructed according to these rules – can evolve together over time. Each CoSMIC tool synthesizes metadata in XML and passes this XML to the next stage of the tool chain.

The initial CoSMIC toolsuite uses a *platform-specific model* (PSM) approach that integrates the modeling technology with our CIAO QoS-enabled component middleware [5] since CIAO is targeted to meet the QoS requirements of

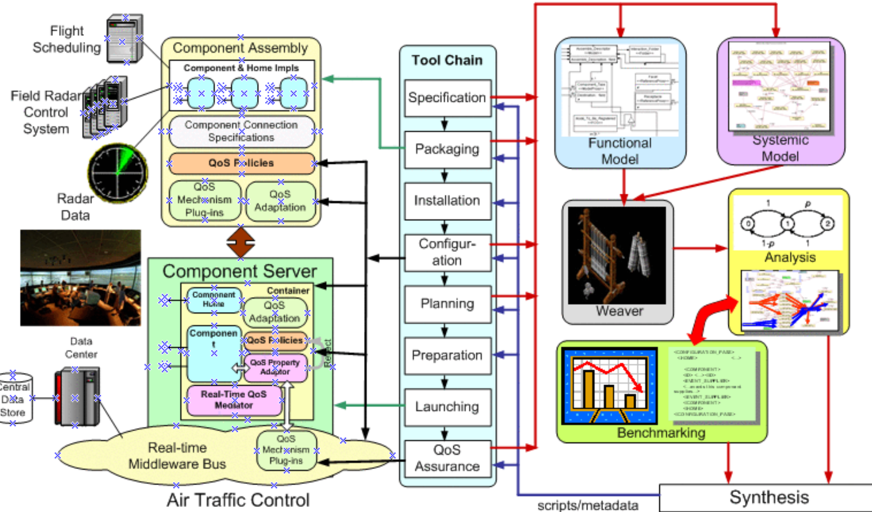

Fig. 5. **CoSMIC Model Driven Middleware Toolsuite**

DRE applications. As other component middleware platforms (such as J2EE and .Net) mature and become suitable for DRE applications, however, 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 each tool in our CoSMIC toolsuite, the modeling paradigms we have developed for that tool, and how the tool resolves the R&D challenges described in Section 2.

3.1 Model-driven Component Packaging: Resolving Component Packaging Challenges

CoSMIC provides the Composable Adaptive Software Systems (COMPASS) tool to resolve the problem of packaging component functionality described in Challenge 2 of Section 2.2. COMPASS defines a modeling paradigm that allows DRE application integrators to model the component assembly and packaging aspect. COMPASS also provides built-in constraint checkers that check for syntactic, semantic and binary compatibility of the assembled components. Moreover, the COMPASS model interpreter enables the synthesis of metadata describing component packages.

Figure 6 illustrates how COMPASS fits into the overall CoSMIC tool chain, which enables application developers to model, synthesize, and deploy DRE applications. Below we describe how the COMPASS tool is used by DRE application integrators.

Package modeling. The modeling paradigm of COMPASS comprises different packaging and configuration artifacts and also legal domain-specific associations between the various artifacts. The modeling paradigm is defined

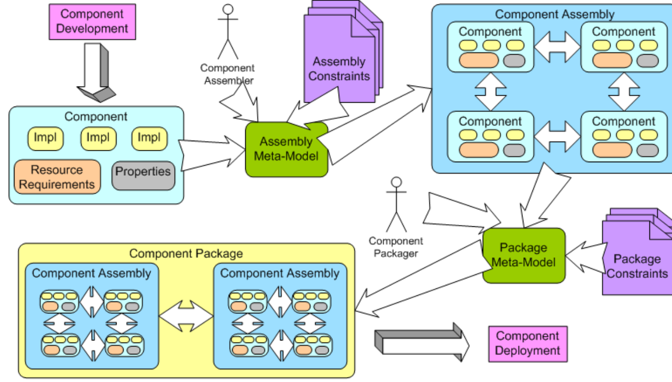

Fig. 6. **COMPASS**

such that the application integrator is able 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.

Configuration modeling. Since components can be composed out of assemblies of components, the individual components need to carry 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 can be done using metrics defined by the OMG *Deployment and Configuration of Component-based Distributed Applications Specification* [25], 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).

Constraint specification. COMPASS provides a constraint checker to ensure that the packages it creates are valid. This checker plays a crucial role in enforcing the CoSMIC "correct by construction" techniques. Constraints are defined on elements in the COMPASS meta-model using the Object Constraint Language (OCL) [26], which is a strongly typed, declarative, query and constraint language that has formal mathematical semantics domain experts can use to describe their domain constraints. COMPASS defines constraints

to capture the restrictions that exist in the context of component packaging and configuration, including (1) creation of component packages, (2) interconnection of component packages, (3) composition of packages, (4) creation of component assemblies, (5) interconnection of component assemblies, (6) composition of assemblies, (7) creation of components, and (8) interconnection of components.

Adding constraints to the COMPASS meta-model ensures that illegal connections are not made among the various modeling elements, which also catches errors early in the component development cycle. Since COMPASS performs static modeling 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.

Interpretation. The COMPASS model interpreter translates the various packaging and configuration information captured in the models constructed using the meta-model into a set of descriptors, which are files containing metadata that describes the systemic information of component-based DRE applications. The output of the COMPASS model interpreter serves as input to another tool downstream, such as the deployment planner described in Section 3.3, that will use the information in the descriptors to deploy the components.

The descriptors generated by COMPASS model interpreter are XML documents that conform to a XML Schema [27,28]. To ensure interoperability with other CoSMIC modeling tools, COMPASS synthesizes descriptors conforming to the XML schema defined by the *Deployment and Configuration of Component-based Distributed Applications Specification.* COMPASS generates 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 like component ports

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

3.2 Model-driven Middleware Configuration: Resolving Configuration Challenges

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

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

• **OCML.** The OCML modeling paradigm addresses middleware level configuration options. OCML contains artifacts to define and categorize the middleware options and to configure the middleware with these options. OCML also generates the documentation for the middleware options. The user interface of the OCML is based on the Graphical Modeling Environment (GME).

The OCML tool is intended to be used by both the middleware developer and the application developer. The middleware developer uses the OCML metamodel to create the middleware options configuration model. The options configuration model contains two artifacts: (a) the *structure* artifact, which contains all the available options categorized hierarchically in different folders. For example, we have used OCML to model all the configuration options provided by the TAO [29] ORB; (b) The *rules* artifact contains the dependency relations among these options.

An application developer uses OCML to model a permutation of ORB configuration options to be used for configuring the ORB. After validating the compatibility of the selected options, a ORB-specific service configuration file is generated. Figure 7 illustrates how OCML can be used for middleware configuration.

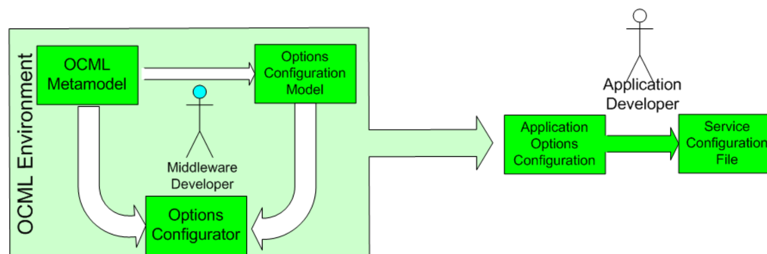

Fig. 7. **OCML Process**

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

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

• **FESML.** One strategy by which a real-time event service can be configured to minimize network traffic is building federations of event services that share filtering information to minimize or eliminate the transmission of unwanted events to a remote entity. Moreover, federation of event services allows events that are being communicated in one channel to be made available on other channels. The channels could communicate with each other through CORBA IIOP Gateways, UDP, or IP Multicast [30]. Connecting event channels from different systems together will allow event information to be interchanged, providing a level of integration among the systems.

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

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

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

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

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

Interpretation. The CoSMIC middleware configuration toolset provides model interpreters that synthesize middleware configuration files and component descriptor files.

• **OCML.** The middleware specific options configuration language is validated against the OCML meta-model and when interpreted generates the following:

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

This procedure is illustrated in Figure 7.

• **EQAL.** EQAL encompasses two model interpreters. The first interpreter generates XML descriptor files that conform to the Boeing Bold Stroke XML schema for component RT event service configuration. These descriptor files identify the RT requirements of individual connections. The second interpreter generates the service configuration files that specify event channel policies and strategies. The component deployment framework parses these files, creates event channels, and configures each connection, while shielding the actual component implementations from the lower-level middleware services. Currently, these files must be written by hand a tedious process that is repeated for each component deployment. Accordingly, the automation of this process, and the guarantee of model validity, improves the reusability of components across diverse deployment scenarios.

• **FESML.** FESML includes a model interpreter that generates XML files to specify the configuration of the federation of event channels. The information captured in the descriptor files include the relationship between each artifacts, the physical location of each supplier, consumer, event channel, CIAO Gateway, etc. This file will be then be further fed into the CIAO assembly and deployment tool to deploy the system.

3.3 Model-driven Configuration and Deployment of Components : Resolving the Deployment Planning Challenge

CoSMIC provides the Model Integrated Deployment and Configuration Environment for Composable Software Systems (MIDCESS) and the CCM Performance (CCMPerf) tools to resolve the problem of deployment planning described in challenge 4 of Section 2.4. MIDCESS is used to specify the target environment for deploying the packages. CCMPerf uses the aforementioned target information, and uses empirical benchmarking in identifying the packages to be configured and deployed in the target environment. These two tools in concert help in creating the deployment plan for the final deployment of packages.

A target environment is a model of the computing resource environment (such as processor speed and type of operating system) in which a component-based application will execute. The various entities of the target model include:

(1) **Nodes**, where the individual components and component packages are loaded and used to instantiate those components,
(2) **Interconnects** among nodes, to which inter-component software connections are mapped, to allow the instantiated components to intercommunicate, and
(3) **Bridges** among interconnects. Interconnects provide a direct connection between nodes, while bridges provide a routing capability between interconnects.

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

Using the target environment information available from MIDCESS, CCMPerf can be used to create experiments that measure black box, *e.g.*, latency, throughput and white-box *e.g.*, jitter, context-switch overhead, metrics that can be used to know the consequences of mixing and matching component assemblies on a target environment. CCMPerf can also be used to synthesize experiments on a per component basis, the motivation being to generate unit and regression tests. The experiments in CCMPerf can be divided into the following three experimentation categories:

(1) *Distribution middleware* tests that quantify the performance of CCM-based applications using black box and white box techniques
(2) *Common middleware services* tests that quantify the suitability of using different implementations of CORBA services, such as the Real-time Event [31] and Notification Services [32].
(3) *Domain-specific middleware* tests that quantify the suitability of CCM implementations to meet the QoS requirements of a particular DRE application domain, such as static linking and deployment of components

in the Boeing Bold Stroke avionics mission computing architecture [2].

A model based approach to planning the deployment allows the modeler to get information about the target environment, get the middleware configuration information and generate tests at the push of button. Without modeling techniques, these tedious and error-prone code would have to be written by hand. In a hand-crafted approach, changing the configuration would entail re-writing the benchmarking code. In a model based solution, however, the only change will be in the model and the necessary experimentation code will be automatically generated. A model based solution also provides the right abstraction to visualize and analyze the overall planning phase rather than looking at the source code. In the ensuing paragraphs we describe the design of MIDCESS and CCMPerf.

Figure 8 illustrates how MIDCESS and CCMPerf are designed to be a link in the CoSMIC tool chain that enables developers to model the planning phase of the component development process. Below we describe how the MIDCESS

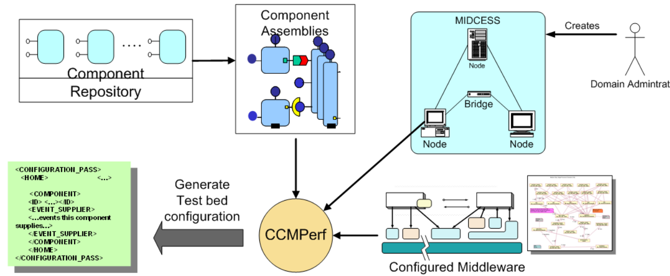

Fig. 8. **PLANNING**

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

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

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

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

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

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

(1) *Configuration Aspect*, that defines the interface that are provided and required by the individual component,
(2) *Metric Aspect*, that defines the metric captured in the benchmark, and
(3) *Inter-connection Aspect*, that defines how the components will interact in the particular benchmarking experiment.

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

• **MIDCESS.** MIDCESS contains a constraint checker to ensure that the target environments specified by the tool are semantically compatible. Constraints are defined using the Object Constraint Language (OCL) [26], which is a strongly typed, declarative, query and constraint language. MIDCESS defines constraints to enforce restrictions in the (1) specification of node elements, (2) specification of interconnect elements, (3) specification of bridge elements, (4) specification of resource elements, and (5) interconnection of various elements of the domain.

• **CCMPerf.** Additionally, a constraint checker validates the experiment precluding the possibility of invalid configuration, such as: (1) conflicting metrics (*e.g.*, using both back box and white box metrics in a given experiment), (2) invalid connections (*e.g.*, not connecting a required interface with the corresponding provides interface), and (3) incompatible exchange format (*e.g.*, connecting a point-to-point entity with a point to multi point entity). Constraints are defined in the CCMPerf meta model are defined using OCL [26]. The use of constraints ensure that the experiment is correct a priori minimizing errors at run-time.

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

• **MIDCESS.** MIDCESS generates a *domain descriptor* that describes the domain aspect of the target model environment of composable software systems. This descriptor is an XML document that conforms to a XML Schema defined by the *Deployment and Configuration of Component-based Distributed Applications Specification* [25]. The output of MIDCESS can therefore be validated by running the descriptor through a tool that supports XML schema validation. The generated descriptor is also used by the CoSMIC run-time infrastructure, which uses information in the descriptor to make deployment planning decisions.
• **CCMPerf.** From the CCMPerf meta-model, an interpreter generates the necessary descriptor files that provide meta-data to configure the experiment. In particular, the interpreter embellishes the following descriptor files generated by COMPASS (described in Section 3.1) including:

(1) *Component package descriptor*, by selecting the component implementation used in the test,
(2) *Component implementation descriptor*, by specifying the interactions between the components, and
(3) *Component interface descriptor*, by associating metrics with the corresponding interfaces.

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.

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) [7,33,34,8] 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 [35] (used primarily in the real-time and embedded domain) and MDA [9] based on UML [36] and XML [37] (which have been used primarily in the business domain). Our work on CoSMIC combines the GME tool and UML modeling language to model and synthesize QoS-enabled component middleware for use in provisioning DRE applications. In particular, CoSMIC is enhancing GME

to produce domain-specific modeling languages and generative tools for DRE applications, as well as developing and validating new UML profiles (such as the UML profile for CORBA [38], the UML profile for quality of service [39], and UML profile for schedulability, performance and time [40]) to support DRE applications.

As part of an ongoing collaboration [41] between ISIS, University of Utah, and BBN Technologies, work is being done to apply GME techniques to model an effective resource management strategy for CPU resources on the Timesys Linux real-time OS [42]. Timesys Linux allows applications to specify CPU reservations for an executing thread, which guarantee that the thread will have a certain amount of CPU time, regardless of the priorities of other threads in the system. Applying GME modeling to develop the QoS management strategy simplifies the simulation and validation necessary to assure end-to-end QoS requirements for CPU processing.

The Virginia Embedded System Toolkit (VEST) [43] 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 lightweight formal methods to CCM-based DRE applications. The Cadena tools are implemented as plug-ins to IBM's Eclipse integrated development environment (IDE) [44]. This architecture provides an IDE for CCM-based DRE systems that ranges from editing of component definitions and connections information to editing and debugging of auto-generated code templates.

Commercial successes in model-based software development include the Rational Rose [45] 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, in-

cluding control systems, digital signal processors (DSPs), and telecommunication systems. Simulink is capable of simulating the modeled system's behavior, evaluating its performance, and refining the design. Stateflow is an interactive design tool for modeling and simulating event-driven systems. Stateflow is integrated tightly with Simulink and Matlab to support designing embedded systems that contain supervisory logic. Simulink uses graphical modeling and animated simulation to bridge the traditional gap between system specification and design.

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

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

Generative Programming (GP) [46] is a type of program transformation concerned with designing and implementing software modules that can be combined to generate specialized and highly optimized systems fulfilling specific application requirements. The goals are to (1) decrease the conceptual gap between program code and domain concepts (known as achieving high intentionality), (2) achieve high reusability and adaptability, (3) simplify managing many variants of a component, and (4) increase efficiency (both in space and execution time).

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

Yet another type of program transformation is aspect-oriented software development (AOSD). AOSD is a new technology designed to more explicitly

separate concerns in software development. The AOSD techniques make it possible to modularize crosscutting aspects of complex DRE systems. An aspect is a piece of code or any higher level construct, such as implementation artifacts captured in a MDA PSM, that describes a recurring property of a program that crosscuts the software application *i.e.*, aspects capture crosscutting concerns). Examples of programming language support for AOSD constructs include AspectJ [47] and AspectC++ [48].

5 Concluding Remarks

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

It is in this context that the OMG's Model Driven Architecture (MDA) is an effective paradigm to address the challenges described above. The MDA is a software development paradigm that applies domain-specific modeling languages systematically to engineer computing systems. This paper provides an overview of the emerging paradigm of *Model Driven Middleware* (MDM), which applies MDA techniques to help configure and deploy QoS-enabled component middleware and large-scale DRE applications and systems of systems. The MDM analysis-guided composition and deployment of DRE applications helps to provide a verifiable and certifiable basis for ensuring the consistency and fidelity of DRE applications, such as those deployed in safety-critical domains like avionics control, medical devices, and automotive systems.

To illustrate recent progress on MDA technologies, this paper describes CoSMIC, which is an MDM toolsuite that combines the power of domain-specific modeling, aspect-oriented domain modeling, mathematical analysis, generative programming, QoS-enabled component middleware, and run-time dynamic adaptation and resource management to resolve key challenges that occur throughout the DRE application lifecycle. CoSMIC currently provides platform-specific metamodels that address the packaging, middleware configuration, deployment planning and runtime QoS assurance challenges. The middleware platform we use to demonstrate our MDM R&D efforts is the Component-Integrated ACE ORB (CIAO) [5], which is QoS-enabled imple-

mentation of the CORBA Component Model (CCM). As other component middleware technologies mature to the point where they can support DRE applications, the CoSMIC tool-chain will be enhanced to support platform-independent models and their mappings to various platform-specific models.

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[26] Object Management Group, Unified Modeling Language: OCL version 2.0, OMG Document ptc/03-08-08 Edition (Aug. 2003).

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

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

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

[30] 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).

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

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

[33] 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

[34] 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

[35] 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.

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

[37] W. A. Domain, Extensible Markup Language (XML), http://www.w3c.org/XML.

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

[39] 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).

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

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

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

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

[44] 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).

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

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

[47] 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`

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