Applying MDA and Component Middleware to Large-scale Distributed Systems: A Case Study

Andrey Nechypurenko
Siemens Corporate Technology
Munich, Germany
andrey.nechypurenko@siemens.com

Tao Lu, Gan Deng,
Douglas C. Schmidt, Aniruddha Gokhale
Vanderbilt University, Nashville, TN, USA
{tao.lu, gen.deng, d.schmidt, a.gokhale}@vanderbilt.edu

Abstract

Despite advances in hardware and software technologies, it remains challenging to develop large-scale distributed systems that are correct, efficient, and flexible. Some challenges arise from increasingly demanding end user requirements for quality and functionality. Other challenges arise from complexities associated with integrating large-scale distributed systems composed of modular components. This paper provides two contributions to R&D efforts that address these challenges. First, it motivates the use of an integrated Model Driven Architecture (MDA) and component middleware approach to enhance the level of abstraction at which distributed systems are developed to (1) improve software quality and developer productivity and (2) reduce the complexity of component integration. Second, we present our experience gained applying MDA and component middleware software techniques to develop an Inventory Tracking System that monitors and controls the flow of goods and assets in warehouses. Our preliminary results show that using MDA tools and component middleware as the core elements of software composition leads to reduced development complexity, improved system maintainability, and increased developer productivity.

Keywords: Model Driven Architecture, Component Middleware, CORBA Component Model (CCM).

1. Introduction

During the past five decades the IT industry has experienced a steady increase in the complexity of both problem and solution spaces. In the problem space of various domains (such as telecom, enterprise business, aerospace, and industrial process control systems) software-intensive systems developed today are often considerably larger and more complicated than those developed two decades ago. In the solution space, gigabytes of documentation, source code, and binaries are supplied by providers of infrastructure software (such as operating systems, database management systems, graphical user interface packages, and component middleware), which suggests their complexity has grown beyond the ability of most developers to comprehend many aspects of these popular technologies.

The IT industry has historically addressed the growth in complexity by raising the level of abstraction at which software systems are developed, integrated, and validated. For example, the growing complexity of larger-scale assembly language programs in the 1960s motivated the creation and adoption of the next generation of higher-level programming languages (such as Pascal and C++), which raised the abstraction level and helped improve the efficiency and quality of software development. Likewise, the growing complexity of developing large-scale systems from scratch motivated the creation and adoption of frameworks and patterns as a way to provide semi-complete applications and factor out reusable structures and behaviors in mature domains (such as network programming, database access, and GUI creation).

More recently, component middleware technologies (such as J2EE, .NET and CCM) have factored out key functional and non-functional aspects (such as component lifecycle management, authentication/authorization, and remoting) to shield application developers from low-level, non-portable platform details (such as socket-level programming). As a result, a growing number of large-scale distributed systems are being assembled from modular components – many of which are available from commercial-off-the-shelf (COTS) providers – rather than developed manually from scratch using proprietary, monolithic software technologies.

Although the capabilities of higher-level component middleware can help to alleviate many complexities associated with lower-level platforms and tools, the complexity of today's component middleware technologies yields new challenges to application developers. For example, considerable effort must now be expended to integrate business logic with the set of rules and behavior dictated by component models. A concrete example is: component containers in J2EE and CCM. They can activate or passivate components according to a lifecycle management strategy that is independent from the business logic implemented by the components. This process, however, imposes non-trivial restrictions and rules on component developers. Moreover, different component middleware platforms implement lifecycle management differently, which incurs additional complexities for applications that must run on multiple platforms.

Another challenge confronting developers of large-scale COTS-based component systems is that few software developers have an *integrated* view of all the subsystems and libraries in large-scale systems. Instead, they are only familiar with a subset of the characteristics of the subsystems and libraries they use regularly, which

makes it hard for developers to know which portions of the system functionality are influenced by changes arising from bug fixes, new requirements, new platforms, etc. The lack of an integrated view – coupled with the danger of unforeseen side-effects – often force developers to implement suboptimal solutions that duplicate code unnecessarily, violate key architectural principles, and complicate system maintenance.

As a result of these challenges, it is not surprising that large-scale distributed systems often have many defects and are chronically over budget and behind schedule, even when they are based on the most advanced software technologies. In particular, despite improvements in third-generation programming languages (such as Java or C++) and run-time platforms (such as component middleware), the level of abstraction at which business logic is integrated with the set of rules and behavior dictated by component models is still too low. For example, the components and the underlying component middleware framework often 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 to use third-generation languages to write programs that manually ensure all these parameters are semantically consistent throughout an application. Moreover, there is no formal basis for validating and verifying that middleware configured via such *ad hoc* approaches will deliver the intended behaviors. In addition, the level of abstraction supported by third-generation languages does not intuitively reflect the concepts used by today's cutting-edge software developers, who are using higher level concerns (such as persistence, remoting, and synchronization) to express their system architectures.

A promising way to alleviate these problems with low-level abstractions and tools is to apply *Model Driven Architecture (MDA)* techniques [MDA] that express application functional and non-functional requirements at higher levels of abstraction beyond third-generation programming languages and conventional component middleware. MDA tools help to improve the understanding of software-intensive systems using higher-level *models* that (1) standardize the process of capturing business logic and quality of service (QoS)-related requirements and (2) ensure the consistency of software implementations with analysis information associated with functional and systemic QoS requirements captured by models. A key role in reducing software complexity via MDA tools is played by *meta-modeling* [GME], which defines a semantic *type system* that precisely reflects the subject of modeling and exposes important constraints associated with specific application domains.

To evaluate the extent to which MDA technologies actually improve development productivity, quality, and understanding, we have developed a prototypical Inventory Tracking System (ITS). The ITS is a warehouse management system that monitors and controls the flow of goods and assets. Users of an ITS include couriers, such as UPS, FedEx, DHL, as well as airport baggage handling systems.

This paper uses a portion of our ITS prototype as a case study to illustrate the benefits of integrating MDA and component middleware by focusing on two of the fundamental aspects of ITS, "Warehouse Configuration" and "Component Assembly and Configuration". We will illustrate (1) how our MDA tool suite with two aspects we developed could capture end user's concerns in a ITS system, (2) how the concerns are mapped to the actual artifacts which are used by the run time framework -- CIAO[1].

The remainder of this paper is organized as following: Section 2 outlines our Inventory Tracking System (ITS) prototype; Section 3 describes how we applied MDA tools and techniques to generate (a) warehouse domain-specific models for ITS and (b) middleware configuration tools used to generate CIAO configuration artifacts; and Section 4 summarizes our lessons learned and presents concluding remarks.

2. Overview of the ITS Case Study

A key goal of an Inventory Tracking System (ITS) is to provide convenient mechanisms that manage the movement and flow of inventory in a timely and reliable manner. For instance, an ITS should enable human operators to configure warehouse storage organization criteria, maintain the set of goods known throughout a highly distributed system (which may span organizational and even international boundaries), and track warehouse assets using GUI-based operator monitoring consoles. This section presents an overview of the behavior and architecture of our ITS prototype. Section 3 then uses this prototype to illustrate how MDA can be integrated with component middleware and applied to large-scale distributed system development.

2.1 ITS System Behavior

Figure 1 shows a UML use case diagram for our ITS prototype. As shown in the figure, there are three primary *actors* in the ITS system.

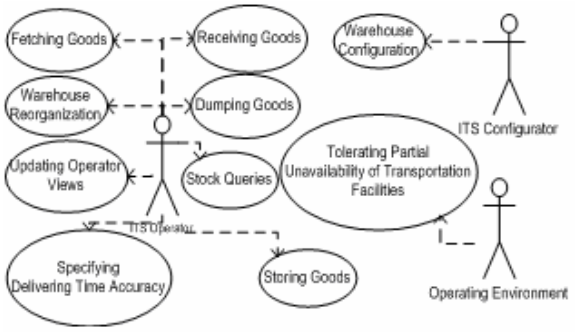

Figure 1. Use Case Diagram for the ITS Prototype

[1] Creating generators for J2EE and .NET component middleware remains as future work.

For the *Configurator* actor, the ITS provides the ability to configure the set of available facilities in certain warehouses, such as the structure of transportation belts, routes used to deliver goods, and characteristics of storage facilities (e.g., whether hazardous goods are allowed to be stored, maximum allowed total weight of stored goods, etc.). For the *Operator* actor, the ITS provides the ability to reorganize the warehouse to fit future changes, as well as dealing with other use cases, such as receiving goods, storing goods, fetching goods, dumping goods, stock queries, specifying delivery time accuracy, and updating operator console views. For the *Operating Environment* actor, the ITS provides the ability to tolerate partial failures due to transportation facility problems, such as broken belts. To handle these partial failures the ITS dynamically recalculates the delivery possibilities based on available transportation resources and delivery time requirements.

2.2 Architecture

The ITS architecture is based on component middleware developed in accordance with the OMG's CORBA Component Model (CCM) [CCM]. A component is a basic meta-type in CCM that consists of a named collection of features – known as *ports*, i.e., event sources/sinks, facets, and receptacles – that can be associated with a single well-defined set of behaviors. In particular, a CCM component provides one or more ports that can be connected together with ports exported by other components. CCM also supports the hierarchical encapsulation of components into *component assemblies*, which export ports that allow fine tuning of business logic modeling.

Figure 2 illustrates the key components that form the basic implementation and integration units of our ITS prototype. Some ITS components (such as the Operator

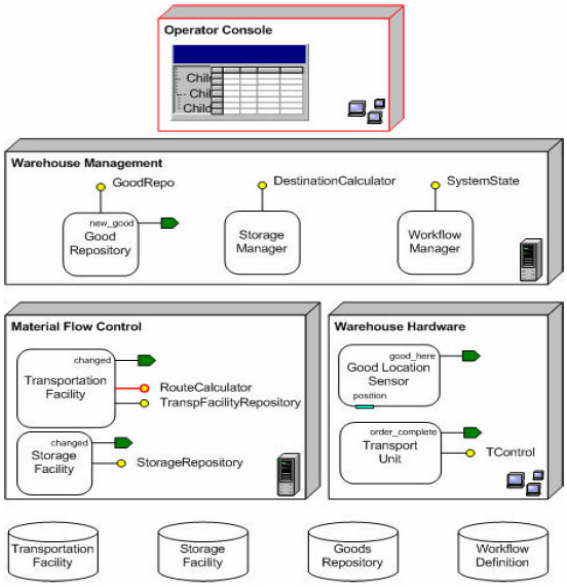

Figure 2. Key CCM ITS Architecture Components

Console component) expose interfaces to end users, i.e., ITS operators. Other components represent warehouse hardware entities (such as cranes, forklifts, and shelves) and expose interfaces to manage databases (such as Transportation Facility component and the Storage Facility component). Yet another set of components (such as the Workflow Manager and Storage Manager components) coordinate and control the event flow within the ITS system.

As illustrated in Figure 2, the ITS architecture consists of the following three subsystems:

1. **Warehouse Management (WM) subsystem**, which is a set of high-level functionality and decision making components. This level of abstraction calculates the destination location and delegates the rest of the details to the Material Flow Control (MFC) subsystem. In particular, the WM does not provide capabilities such as route calculation for transportation or reservation of intermediate storage units.
2. **Material Flow Control (MFC) subsystem**, which is responsible for executing high-level decisions calculated by the WM subsystem. The primary task of the MFC is to deliver goods to the destination location. This subsystem handles all related details, such as route (re)calculation, transportation facility reservation, and intermediate storage reservation.
3. **Warehouse Hardware (WH) subsystem**, which is responsible for dealing with physical devices, such as sensors and transportation units (e.g., belts, forklifts, cranes, pallet jacks, etc.).

The functionality of these three ITS subsystems is monitored and controlled via an Operator Console. All persistence aspects are handled via databases that can be managed either by the centralized DBMS or distributed DBMS over different DB servers. A typical interaction scenario between these three subsystems is illustrated by the following action sequence:

1. The new good arrives at the warehouse entrance and is entered into the ITS either automatically or manually.
2. The WM subsystem calculates the final destination for storing the good by querying the Storage Facility for a list of available free locations. The final destination is passed to the MFC subsystem.
3. The MFC subsystem calculates the transportation route and assigns required transportations facilities.
4. The MFC subsystem interacts with the WH subsystem to control the transportation process and if necessary adapt to changes, such as failures or the appearance of higher priority tasks.

For the technical infrastructure of our initial ITS prototype, we selected the Component Integrated ACE ORB (CIAO) [CIAO1, CIAO2], which is QoS-enabled CCM middleware built atop the *The ACE ORB* (TAO) [TAO1, TAO2]. TAO is a highly configurable, open-source[2]

[2] CIAO and TAO can be downloaded from http://deuce.doc.wustl.edu/Download.html.

Real-time CORBA Object Request Broker (ORB) that implements key patterns [POSA2] to meet the demanding QoS requirements of distributed real-time and embedded (DRE) systems. CIAO extends TAO to provide the component-oriented paradigm to developers of DRE systems by abstracting critical systemic aspects (such as QoS requirements, real-time policies) as installable/configurable units supported by the CIAO component framework. Promoting these DRE aspects as first-class metadata disentangles (1) code for controlling these non-function aspects from (2) code that implements the application logic, ideally making DRE system development more flexible and productive as a result.

3. Model Driven ITS Development

To evaluate how MDA technologies can help improve productivity by enabling developers to work at a higher abstraction level than components and classes, we developed and applied a set of modeling tools to automate the following two aspects of ITS development:

1. Warehouse modeling, which simplifies the warehouse configuration aspect of the ITS system according to the equipment available in certain warehouses, including moving conveyor belts and various types of cranes. These modeling tools can synthesize the ITS database configuration and population.
2. Modeling and synthesizing the deployment and configuration (D&C) aspects of the components that implement the ITS functionality. These modeling tools use MDA technology in conjunction with the CCM to develop, assemble, and deploy ITS software components.

This section describes these two modeling aspects, focusing on the domain models and model interpreters. We also explore the relationship between these aspects to show how multiple layers of MDA are applied in ITS.

3.1 Modeling an ITS Warehouse with MDA Tools

Warehouse modeling consists of designing the warehouse configuration model by mapping from concrete warehouse structures perceived from a physical standpoint. This is the first phase which must be accomplished prior to setting up an ITS. The following are the two main concerns of a warehouse model in this phase:

1. **Transportation facility network**, which includes position information (e.g., the physical location and reachable areas) and properties, (e.g., the capacity and toxicity of items transported in the network).
2. **Appropriate available storage places**, which includes their physical locations and properties (e.g., storage capacity and type of goods they can store).

In the ITS warehouse model these two concerns are blended together to give the warehouse model developers a convenient overview of the warehouse setup, which is similar to the architectural blue print of the warehouse. Mapping from the architectural blue print to the warehouse model should be intuitive to domain experts, as well as to model developers, so they can reuse the warehouse knowledge efficiently and conveniently.

3.1.1 Choosing the Modeling Tool

After evaluating the requirements of our partners in Siemens business units, we have selected Microsoft® Visio® as our warehouse modeling tool. Visio is a commercially supported graphic drawing tool with meta-modeling capabilities, as well as the following desirable features:

- **Full range of technical diagramming capabilities**. Numerous drawing related features are provided by Visio. For instance, it supports grid, docking point, and object manipulation (e.g., resize, rotation, connection routing), which are valuable for warehouse modeling by domain experts who work on large-scale commercial ITS deployments.
- **Integrated model interpreter with embedded debugging environment**. Unlike traditional tools that focus on a discrete segment of information, Visio offers an integrated toolset for applications, development, and data modeling. Visio is shipped with an embedded Visual Basic® editor and debugging environment that simplifies interpreter writing. C++/COM objects can also be plugged in, if desired.
- **Extensibility**. Visio supports database modeling, which includes complete database design, database schema, and Data Definition Language (DDL) script generation from conceptual and physical models. For example, in the warehouse configuration domain, we can connect the physical model to the associated database. Moreover, Visio ships with many domain-specific paradigms (known as drawing types in Visio). Besides the major building blocks needed by warehouse management systems, Visio also provides many other modeling paradigms, such as UML diagrams. The meta-modeling capability makes it possible to extend Visio's modeling paradigm to suit the domain more effectively.

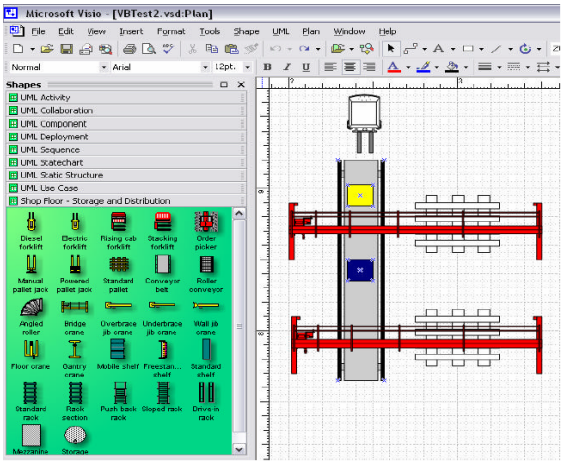

Figure 3. Microsoft Visio ITS Model Example

Figure 3 illustrates a Visio screenshot, where warehouse model elements are available from the master panel (left-side) and the right-side contains the drawing repre-

senting a warehouse fragment consisting of a moving belt, two cranes, storage rack, and a forklift. Modeling a warehouse graphically is therefore as straightforward as mapping/drawing the concrete warehouse physical structure in the Microsoft Visio drawing panel.

3.1.2 Implementing the Model Interpreter

After creating the complete model for a desired warehouse configuration, the corresponding configuration artifacts are generated automatically by using our domain-specific model interpreter. The model interpreter we developed for the warehouse model contains a set of Visual Basic macros that can be executed within Visio to generate corresponding data model as part of component synthesis. In the model interpreter, certain analysis and validation steps are applied to the warehouse model to validate the correctness of the data model. Once validated, C++ code is generated and used at runtime to bootstrap the ITS components, as described below:

1. Certain location-related constraints can be checked by the model interpreter to validate the model to ensure that the physical layout and configuration of the warehouse is valid and meaningful. For example, when a crane is on top of a storage place, the model interpreter can ensure that the crane is capable of reaching all the storage cells of the place. Upon discovering potential conflicts, error or warning messages will be issued to a domain expert.
2. Different domain-specific aspects captured by the graphical model can be extracted from the model to populate the warehouse system databases. The generated artifacts include the classes used to populate the databases and some initialization steps of the databases.

After running the model interpreter, the system is ready to start the component-based deployment and configuration process described in Section 3.2.

3.2 Modeling ITS Component Deployment and Configuration with MDA Tools

As discussed in section 2.2, ITS is developed using CIAO, which is a CCM implementation. As a result, ITS has a standardized way to configure the functional and systemic QoS behavior of its software components and map them to the underlying hardware and software infrastructure in a highly flexible manner. In ITS, component deployment and configuration is performed via the *Component Synthesis using Model Integrated Computing* (CoSMIC) toolsuite [CoSMIC], which is an MDA open-source [3] toolsuite targeted for component-based distributed applications.

At the heart of CoSMIC is the Component Assembly and Deployment Modeling Language (CADML), which automates the deployment and configuration aspects of distributed applications. CADML is a visual language tool developed using the Generic Modeling Environment (GME) framework [GME], which supports the following features:

- GUI interface supporting all general GUI application features with very generic semantic mapping.
- Library importing and exporting capability.
- Type system defined in the meta-model, which supports inheritance and instantiation. This introduces object-oriented design (OOD) in the modeling paradigm.
- Formalized constraints specified in the meta-model to validate the model.
- Plug-in of analysis and synthesis tools that interpret the models

The current release of CoSMIC's CADML tool supports the CCM Deployment and Configuration standard [D&C] and works out-of-the-box with CIAO. The CADML modeling paradigm allows developers of CIAO-based application to model component assemblies that capture the connections between different application components.

The CADML model interpreter synthesizes component assembly metadata as XML descriptors, which are then used by CIAO middleware deployment tools. Different descriptor files represent different application scenarios. With the support of a component repository, application developers can configure and deploy different application scenarios by providing the required descriptors. For example, Figure 4 presents a screen shot that illustrates how the deployment and configuration of ITS components are modeled in the CADML modeling environment.

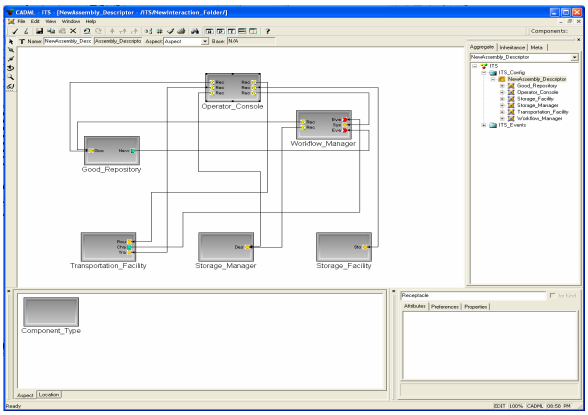

Figure 4. ITS CCM Component Assembly Model

The generated XML descriptors are fed into the CIAO component middleware runtime environment, which then deploys the components into the containers throughout the ITS distributed system. This MDA-based modeling approach is essential to the CCM D&C process. In particular, it automates the descriptor generation to avoid errors that arise when the *ad-hoc* handcrafting approach is used. Moreover, analysis is performed on the D&C models to ensure semantic correctness of the

[3] CoSMIC can be downloaded from http://www.dre.vanderbilt.edu/cosmic/.

configurations, e.g., only the ports with the same interface or event type could be connected.

3.3 Relations between the Warehouse Model and the Component D&C Model

As discussed above, there are two types of modeling aspects in ITS: (1) warehouse modeling and (2) component deployment and configuration (D&C) modeling. These two aspects are semi-orthogonal to each other in terms of aspect separation, i.e., they depict the overall system from different perspectives, yet they are complementary to each other. For example, Figure 5 shows how the system modeler and warehouse modeler are different roles in the ITS development process.

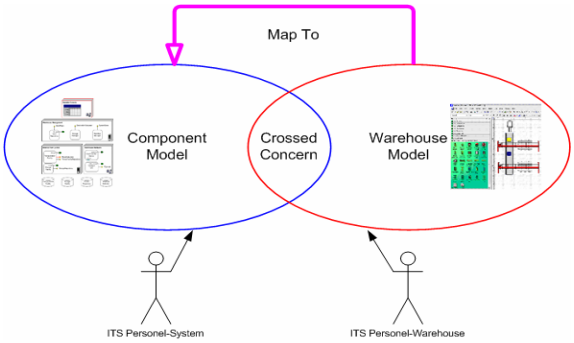

Figure 5. ITS Modeling Aspects

The system modeler studies the business logic of general ITS and produces a model describing the software aspect of the system, including CCM component, deployment/assembly specification, and QoS requirements. The warehouse modeler, in contrast, is responsible for modeling one or a group of specific warehouses.

The warehouse and component model aspects can be implemented separately during system development, i.e., the warehouse model can be mapped to the CCM and D&C model by means of MDA-based code generation to fully materialize an ITS system. There exist, however, some concerns that span these two aspects. For example, the number of components and the way they are communicate with each other can influence the configuration of different infrastructural aspects, such as real-time event channels [Harrison]. In ITS, however, a warehouse modeler often needs to fine tune the configuration on the base of warehouse model. In these cases, different actions are applied according to the nature of the concern after necessary analysis.

4. Concluding Remarks

Advances in hardware and software raising the level of abstraction at which distributed systems are developed. With each increase in abstraction comes a new set of complexities that must be mastered to reap the rewards of the higher-level technologies. A key challenge associated with higher-level software abstractions is that the *integration complexity* makes it hard to assure the overall quality of the complete system. To explore the benefits of applying Model Driven Architecdture (MDA) technologies to address these challenges, we have developed an Inventory Tracking System (ITS) prototype, which is a distributed system that employs MDA tools and component middleware to address key requirements from the warehouse management application domain.

The lessons we have learned applying MDA and component middleware technologies thus far include:

- The component middleware paradigm elevates the abstraction level of middleware to enhance software developer quality and productivity. It also introduces extra complexities, however, that are hard to handle in an *ad-hoc* manner for enterprise application. For example, the CCM requires many configuration files due to its large number of configuration points.
- The MDA paradigm expedites application development with the proper tool support. In the ITS project, if the warehouse model is the only missing or changing aspect in the system (which is typical for end users), little new application code must be written. Likewise, in the case when the software model is missing or changes, application developers must write the component implementation code and finish the component model. Even in this latter case, however, the amount of effort required is significantly less than starting from the raw component middleware.
- Domain-specific modeling techniques can help to reduce the learning curve for end users. For example, warehouse modelers in our ITS project need little or no knowledge of how to write conventional software since they interact with the system entirely through models and visual modeling environments.
- Models at different abstraction layers or reflecting different aspects often exist in the large-scale MDA-based systems. Weaving the models together to form the overall system is very important. In ITS, this is currently done in a *ad-hoc* manner. To solve this problem an even higher level of abstraction is needed based on the concept of "concern" as a fundamental building block. For example, we could define yet another modeling paradigm to capture the meta-models of both the warehouse model and the component model, as well as important correlations between these two meta-models. A model weaving process [Gray] could then be captured in the model and automated.

In future work we plan to implement an integrated concern modeling and manipulation environment to achieve the benefits outlined in the last bullet point above. We also plan to extend our MDA modeling tools so they can perform a two step mapping from (1) the domain-specific model to the platform-independent component-based architecture presented in Figure 2 and (2) the platform-independent model to a CCM-specific implementation.

References

[CCM] BEA Systems, et al., CORBA Component Model Joint Revised Submission, Object Management Group, OMG Document orbos/99-07-01 edition, July 1999.

[CIAO1] Nanbor Wang, Krishnakumar Balasubramanian, and Chris Gill, "Towards a Real-time CORBA Component Model," in OMG Workshop On Embedded & Real-Time Distributed Object Systems, Washington, D.C., July 2002, Object Management Group.

[CIAO2] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Christopher D. Gill, Balachandran Natarajan, Craig Rodrigues, Joseph P. Loyall and Richard E. Schantz, "Total Quality of Service Provisioning in Middleware and Applications," *Microprocessors and Microsystems*, vol. 26, no. 9-10, Jan 2003.

[COSMIC] Aniruddha S. Gokhale, Douglas C. Schmidt, Tao Lu, Balachandran Natarajan, Nanbor Wang: "CoSMIC: An MDA Generative Tool for Distributed Real-time and Embedded Applications," Middleware Workshops 2003.

[D&C] Object Management Group: "Deployment and Configuration of Component-based Distributed Applications", *An Adopted Specification of the Object Management Group, Inc.* June 2003 Draft Adopted Specification ptc/July 2002.

[GME] Akos Ledeczi "The Generic Modeling Environment", Workshop on Intelligent Signal Processing, accepted, Budapest, Hungary, May 17, 2001.

[Gray] Jeff Gray, Janos Sztipanovits, Ted Bapty Sandeep Neema, Aniruddha Gokhale, and Douglas C. Schmidt, "Two-level Aspect Weaving to Support Evolution of Model-Based Software," *Aspect-Oriented Software Development*, Edited by Robert Filman, Tzilla Elrad, Mehmet Aksit, and Siobhan Clarke, Addison-Wesley, 2003.

[Harrison], Tim Harrison, David Levine, and Douglas C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service, Proceedings of OOPSLA '97, Atlanta, Georgia, Oct 1997.

[MDA] OMG: "Model Driven Architecture (MDA)" Document number ormsc/2001-07-01 Architecture Board ORMSC1July 9, 2001.

[POSA2] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Volume 2, Wiley & Sons, New York, 2000.

[TAO1] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

[TAO2] Douglas C. Schmidt et. al, "TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems," *IEEE Distributed Systems Online*, vol. 3, no. 2, Feb. 2002.

[Voelter] Markus Völter, Alexander Schmid, Eberhard Wolff. *Server Component Patterns: Component Infrastructures Illustrated with EJB*, Wiley & Sons. The ISBN is 0-470-84319-5.