Modularizing Variability and Scalability Concerns in Distributed Real-time and Embedded Systems with Modeling Tools and Component Middleware: A Case Study

Gan Deng, Douglas C. Schmidt, Aniruddha Gokhale
EECS Dept., Vanderbilt University
2015 Terrace Place
Nashville, TN 37203, USA
{dengg,schmidt,gokhale}@dre.vanderbilt.edu

Andrey Nechypurenko
Corporate Technology, Siemens AG
Wittelsbacherplatz 2
Munich, D-80333, Germany
andrey.nechypurenko@siemens.com

Abstract

Developing real-time software for large-scale distributed real-time and embedded (DRE) systems is hard due to variabilities that arise from (1) integration with various subsystems based on different programming languages and hardward, OS, middleware platforms, (2) fine tuning the system to satisfy a range of customer requirements, such as various quality-of-service (QoS) properties, and (3) changing functional and QoS properties of the system based on available system resources. This paper describes our experience applying model-driven development (MDD) tools and QoS-enabled component middleware technologies to address domain- and middleware-specific variability challenges in an inventory tracking system, which manages the storage and flow of items in warehouses. Our results show that (1) coherent integration of MDD tools and component middleware can provide a productive software process for developing DRE systems by modularizing and composing variability concerns and (2) significant challenges remain that must be overcome to apply these technologies to a broader range of DRE systems.

Keywords: Model-Driven Development, Domain-Specific Modeling Languages, Component Middleware, and Inventory Tracking Systems

1 Introduction

Emerging trends and challenges. Developing software for large-scale distributed, realtime and embedded (DRE) systems of systems, such as a warehouse inventory tracking system (ITS), is hard due to the numerous factors that must be addressed. For example, ITS software must provide reliable, efficient, and convenient mechanisms that manage warehouses and the movement of inventory items in a timely and reliable manner. An ITS should enable human operators to configure warehouse storage organization criteria and warehouse transportation facility criteria, maintain the set of items known throughout a distributed environment (which may span organizational and even international boundaries), and track warehouse assets using GUI-based operator monitoring consoles. Addressing these challenges is crucial since it impacts the large set of users of an ITS, which includes couriers (such as UPS, FedEx, and DHL), airport baggage handling systems, and large trading and manufacturing companies (such as Wal-Mart and Target).

Standards-based quality-of-service (QoS)-enabled *object-oriented* middleware technologies, such as Real-time CORBA [1] and Real-time Java [2], have been successful in small- to medium-scale DRE systems [3], where they support the provisioning of key QoS properties, such as (pre)allocating CPU resources, reserving network bandwidth, and monitoring/enforcing the proper use of distributed real-time and embedded DRE system resources at runtime to meet end-to-end QoS requirements, such as throughput, latency, and jitter. Since object-oriented technologies tend to tangle functional (*e.g.*, application business-logic) aspects and QoS (*e.g.*, end-to-end latency and jitter requirements) aspects [4], however, it remains hard to develop larger-scale DRE systems, such as an ITS. In recent years, therefore, QoS-enabled *component* middleware [5–8] has emerged to help developers of DRE systems enhance reuse by factoring out reusable concerns, such as component lifecycle management, system resource reservation and allocation, system authentication/authorization, and remoting. As a result, software for large-scale DRE systems is increasingly being assembled from reusable and configurable components.

Although QoS-enabled component middleware technology provides powerful capabilities, however, it also yields the following challenges for developers of DRE systems [9, 10]:

1. Increased scale. As DRE subsystems are joined together to form large-scale systems, developers rarely have in-depth knowledge of the entire system or an integrated view of all subsystems and libraries, which may cause them to implement suboptimal solutions that duplicate code unnecessarily, complicate system evolution, affect system QoS, and violate architectural principles. For example, in an ITS new warehouses may be added at remote locations, which will require reuse of existing ITS software assets to work for the new warehouses and their incorporation within the global ITS system. It is hard enough to satisfy ITS QoS requirements independently at remote locations, and even harder to satisfy them in concert.

2. Increased variability. Additions to the features of the system and/or availability of better implementations of the same type of systems can further increase functional and QoS variability. For example, new types of warehouse transportation facilities, such as forklifts or cranes, which may have more sophisticated features and timeliness requirements may be introduced in a warehouse, hence requiring appropriate changes in the ITS software. It is hard to accommodate these changes within individual components without complicating the solution and affecting the overall QoS of the entire system.

To maximize software reuse and productivity, therefore, increased scale and variability must be addressed by more effectively combining technologies and tools that support system configuration and integration.

Promising approach → Integrating model-driven development and QoS-enabled component middleware. A promising way to alleviate the challenges of DRE system scale and variability described above is to integrate model-driven development (MDD) [10–13] techniques with QoS-enabled component middleware [6,8]. MDD helps resolve key software development challenges by combining (1) *metamodeling*, which defines type systems that precisely express key abstract syntax characteristics and static semantic constraints associated with application domains, such as software defined radios, avionics mission computing, and warehouse inventory tracking, (2) *domain-specific modeling languages* (DSMLs), which provide programming notations that are guided by certain metamodels to formalize the process of specifying application logic and QoS-related requirements in a particular domain, and (3) *model transformations and code generation* that help automate repetitive, tedious and error-prone tasks in the software lifecycle to ensure the consistency of software implementations with analysis information associated with functional and QoS requirements captured by structural and behavioral models.

In prior work, we developed (1) a MDD toolsuite called *Component Synthesis using Model Integrated Computing* (CoSMIC) (`www.dre.vanderbilt.edu/cosmic`), which is an integrated collection of DSMLs that support the specification, analysis, development, configuration, deployment, and evaluation of DRE systems, and (2) a QoS-enabled component middleware platform called *Component-Integrated ACE ORB* (CIAO) [6], which combines Lightweight CORBA Component Model (CCM) [5] capabilities with Real-time CORBA features [1]. To evaluate how well the integration of MDD tools and QoS-enabled component middleware helps resolve the scale and variability challenges described above, we developed an ITS case study that provides logistics support to manage the flow of items and assets in and across warehouses in a distributed and timely manner.

This paper presents our experience gained and lessons learned while integrating MDD and QoS-enabled component middleware to address two key variability concerns in designing the ITS: (1) warehouse configuration and management concerns and (2) component management, assembly, QoS configuration, and deployment concerns. The goal of our case study was to evaluate how well these technologies could be integrated together to (1) modularize key functional and real-time QoS concerns of ITS at higher levels of abstractions than third-generation programming languages, such as Java and C++, (2) handle variabilities at different levels of abstractions, *e.g.*, by assembling a set of components to provision ITS functionality based on particular warehouse requirements, and configuring middleware and services via DSMLs, and (3) automate key steps in the software lifecycle, such as automated ITS product instance software assembly, deployment, and configuration based on warehouse-specific deployment requirements.

Paper organization. The remainder of this paper is organized as follows: Section 2 provides an overview of the ITS case study; Section 3 describes how we integrated and applied MDD tools together with QoS-enabled middleware to resolve key technical problems faced in our case study; Section 4 further evaluates our approach; Section 5 compares our work with related efforts; and Section 6 presents concluding remarks and summarizes lessons learned.

2 Overview of the ITS Case Study

This section provides an overview of the ITS case study, focusing on its component architecture, as well as the scale and variability of its requirements.

2.1 ITS Component Architecture

Figure 1 illustrates the components that form the core implementation and integration units of our ITS case study. Some ITS components (such as the OperatorConsole) expose interfaces to end users, *i.e.*, ITS operators. Other components (such as TransportationUnit) represent hardware entities, such as cranes, forklifts, and belts. Yet other database management components (such as ItemRepository and StorageFacility) expose interfaces to manage external backend databases (such as those tracking items inventory and storage facilities). Finally, the sequences of events within the ITS is coordinated by control flow components (such as the WorkflowManager and StorageManager).

An ITS needs capabilities to tolerate partial failures due to transportation hardware facility problems, such as broken belts. To handle such failures, the software entities associated with hardware devices must alert the ITS work flow manager in real-time, *i.e.*, with low latency delay, and higher processing priority. The ITS must then recalculate the delivery possibilities dynamically based on available transportation resources and delivery time requirements and then reschedule the delivery.

The capabilities shown in Figure 1 are used in the context of their associated ITS subsystems as follows: (1) the *Warehouse Management subsystem* consists of a set of high-level functionality and decision making components that calculate the destination location and delegates other details other subsystems, (2) The *Material Flow Control subsystem* executes high-level decisions calculated by the Warehouse Management subsystem to deliver items to the destination location, *e.g.*, it (re)calculates routes, schedules transportation facilities, and reserves storage facilities, (3) the *Warehouse Hardware subsystem* handles physical devices, such as sensors and transportation units (*e.g.*, belts, forklifts, cranes, and pallet jacks) that correspond to various component types, such as ItemLocationSensor and TransportUnit.

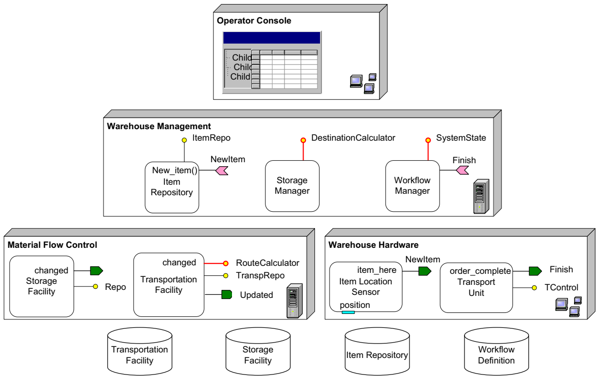


Figure 1: ITS Subsystems and Key Components

The functionality of the ITS subsystems shown in Figure 1 can be monitored and controlled by one or more OperatorConsole components, and all persistence concerns are handled via databases. The Material Flow Control subsystem requires high throughput of continuously refreshed data, soft real-time processing of regular tasks. Components in the Warehouse Hardware subsystem require hard real-time deadlines associated with periodic processing.

Our ITS component architecture is implemented using the CIAO QoS-enabled component middleware, which combines Lightweight CCM features (such as standard mechanisms for specifying, implementing, packaging, assembling, and deploying components) with Real-time CORBA features (such as thread pools, portable priorities, synchronizers, priority preservation policies, and explicit binding mechanisms). ITS also use CIAO's *Deployment And Configuration Engine* (DAnCE) [14], which implements the OMG Deployment and Configuration (D&C) specification [15] to help developers deploy and configure pre-built components and component assemblies. Finally, ITS uses the CoSMIC MDD tools work in conjunction with CIAO and DAnCE to separate the application business logic from system deployment and configuration concerns to make ITS development more rapid and flexible.

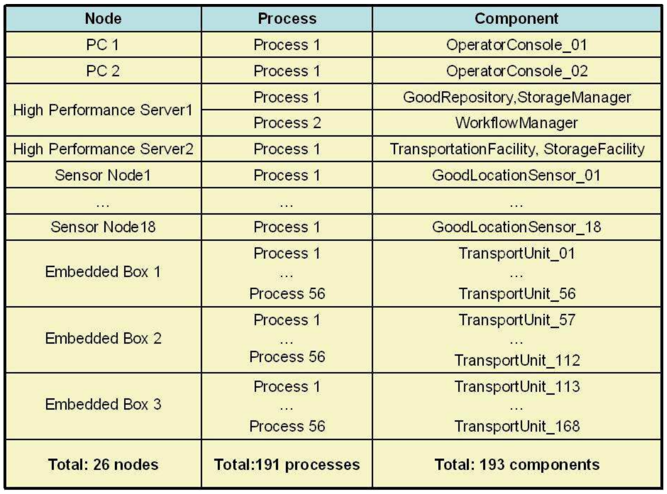
2.2 Scale and Variability in the ITS Case Study

Although the ITS component archihtecture described in Section 2.1 are present in most warehouses, there may be significant differences in customer needs, warehouse specific requirements, task specific QoS requirements, and integration with other subsystems. Implementing an ITS properly therefore requires a thorough understanding of the scalability and variability manifested in the system. For example, the warehouse automation hardware and software infrastructure is often supplied by multiple vendors who select different hardware and software platforms and tools. The resulting heterogeneity yields integration and deployment challenges over an ITS lifetime since various components may be removed or replaced by components from other vendors, which often requires system-wide reconfiguration of system resources to ITS components to improve overall system QoS.

In general, variabilities resulting from different warehouse configurations, hardware/software platforms, and QoS requirements yield much diversity in ITS implementations, particularly for large-scale warehouses that consists of thousands of software/hardware components. An ITS could have a diverse set of characteristics and QoS requirements including – but not limited to – high throughput of continuously refreshed data, hard real-time deadlines associated with periodic processing, well defined computational paths traversing multiple components, soft real-time processing of regular tasks, and operator display and control requirements.

Figure 2 shows the ITS product instance we developed for this case study. Based on requirements mined from Siemens warehouse management business units our case study consists of ~120,000 lines of C++ source code and contains 8 component types and 193 component instances deployed into a warehouse, including 2 OperatorConsoles, 1 TransportationFacility, 1 ItemRepository, 1 StorageManager, 1 WorkflowManager, 1 StorageFacility, 18 ItemLocationSensors and 168 TransportUnits.

These 193 components are deployed into 191 processes, which in turn are hosted in 26 physical nodes, with 2 single-CPU PCs running on Windows XP and 24 nodes running on various versions of Linux OS with some of which having Timesys Linux real-time scheduling class. All the nodes are connected through 100 MB Fast Ethernet and all nodes are installed with the ACE 5.4.7, TAO 1.4.7, and CIAO 0.4.7 middleware. The database management system for ITS case study is MySQL 5.0.



Node	Process	Component
PC 1	Process 1	OperatorConsole_01
PC 2	Process 1	OperatorConsole_02
High Performance Server1	Process 1	GoodRepository,StorageManager
	Process 2	WorkflowManager
High Performance Server2	Process 1	TransportationFacility, StorageFacility
Sensor Node1	Process 1	GoodLocationSensor_01
...	...	...
Sensor Node18	Process 1	GoodLocationSensor_18
Embedded Box 1	Process 1 ... Process 56	TransportUnit_01 ... TransportUnit_56
Embedded Box 2	Process 1 ... Process 56	TransportUnit_57 ... TransportUnit_112
Embedded Box 3	Process 1 ... Process 56	TransportUnit_113 ... TransportUnit_168
Total: 26 nodes	**Total:191 processes**	**Total: 193 components**

Figure 2: Characteristics of an Example ITS System

All components run in separate processes except in two collocated cases: ItemRepository/StorageManager and TransportationFacility/StorageFacility. One should note that even for the same set of components, however, the composition and configuration of an ITS deployment may vary significantly across different warehouses, depending on the availability of actual transportation facilities, computing hardware and software resources.

3 Resolving ITS Challenges via an Integrated MDD Tool and Component Middleware Solution

This section describes how we applied the CoSMIC MDD tools and CIAO QoS-enabled component middleware to help simplify and automate the following activities related to developing the ITS described in Section 2:

- **Modeling and synthesizing warehouse configurations,** which involves simplifying and automating physical layout configuration and transportation facility network design of warehouses, as well as automatically populating databases to capture the above information. Section 3.1 describes the *Warehouse Modeling and Generation Language* (WMGL) MDD tool we developed which raises the abstraction layer of warehouse structures and behaviors to higher-level models.
- **Modeling and synthesizing component software deployment and configuration concerns,** which involves simplifying and automating various configuration concerns of middleware and applications that implement ITS functionality. Section 3.2.1 describes how the CoSMIC MDD tools were integrated to help develop, assemble, and configure ITS software components.

- **Automation of system deployment and configuration,** which involves (1) deploying component assemblies into the appropriate ITS warehouse target nodes, (2) activating and deactivating component assemblies automatically, (3) initializing and configuring component server resources to enforce end-to-end QoS requirements of component assemblies, and (4) simplifying the configuration, deployment, and management of common services used by applications and middleware. Section 3.2.2 describes how the DAnCE middleware was used to help deploy an ITS system onsite.

The remainder of this section describes key problems we faced when addressing these concerns, presents our solutions, and evaluates these solutions qualitatively and/or quantitatively in the context of the ITS case study.

3.1 Addressing ITS Warehouse Configuration Concerns

A key challenge in designing an ITS is to provide a generic, reconfigurable architecture that can be deployed rapidly in different warehouse configurations or redeployed to adapt to reconfiguration needs of existing warehouses. The proper configuration of an ITS depends heavily on the physical layout of transportation facilities and storage facilities of a warehouse, which may vary in different circumstances. A good warehouse physical layout configuration design should enable timely delivery of messages across ITS components, since the "logical" connections of ITS CCM components must map to "physical" layout of a warehouse. The layout information that is specified during the warehouse design phase should therefore be amenable to changes both before and after the warehouse is deployed.

For example, when deploying an ITS in a specific warehouse, all transportation facility units should be mapped to their corresponding software entities, *i.e.*, TransportationUnit components, as described in Section 2.2. Likewise, backend databases should capture and store warehouse physical layout information (*e.g.*, represented by the physical locations of transportation and storage facilities), as well as the reachable range of each transportation facility (*i.e.*, the range within which a transportation unit can pickup and transport items).

Problem: *Ad hoc*, tightly coupled warehouse design. ITS developers have historically relied on *ad hoc* approaches (*i.e.*, manually writing programs from scratch) to (1) create software components that correspond to transportation facility units and (2) populate physical warehouse layout configurations into databases. Moreover, they often hard code such information using third-generation programming languages or scripts, which overly couples their solutions to particular warehouse configurations and technologies. Such tight couplings make an ITS software product hard to evolve after the initial deployment since changes in the warehouse configuration require modification, reverification, recompilation, and redeployment throughout the code.

Solution: A DSML for warehouse configuration. To address the problems described above, we developed the *Warehouse Modeling and Generation Language* (WMGL), which is a DSML in CoSMIC that represents warehouse structures and behaviors as higher-level models. WMGL allows developers to visually depict and manipulate the transportation facility network, which includes position information (*e.g.*, the physical location and reachable areas) and properties (*e.g.*, the type, capacity and toxicity of items each transportation unit could transport in the network). It can also be used to visually depict and manipulate the available storage facilities, which include their physical position information and properties (*e.g.*, storage capacity and type of items they can store).

By capturing the physical position information of the transportation facilities and storage facilities in visual models, WMGL can automatically deduce the topology of the warehouse and generate a warehouse connectivity graph, which is a directed weighted graph that represents the connectivity among transportation facilities and storage facilities. The WorkflowManager component can then apply any pluggable customized path finding algorithm on this graph to determine the optimal transportation path to transfer a particular item from a source (*e.g.*, loading dock or gate) to its destination (*e.g.*, a storage unit). Whenever the warehouse is reorganized or a new transportation facility or storage facility is added, therefore, the graph can be (re)generated automatically from the model, and all other information associated with such change are updated automatically.

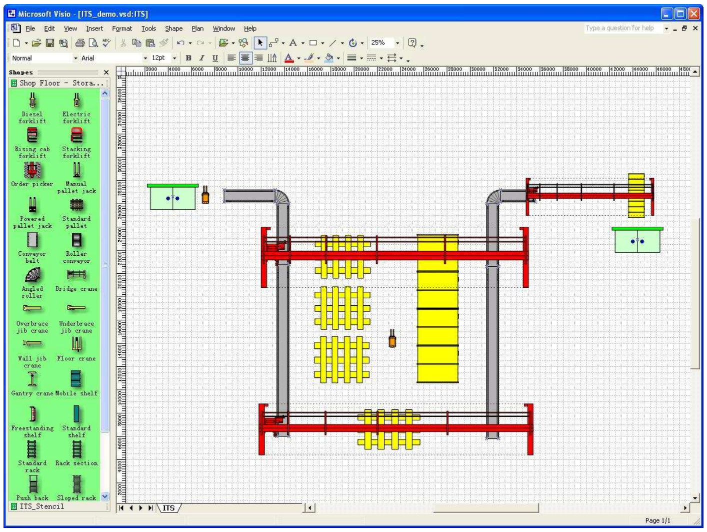


Figure 3: A Warehouse Configuration in WMGL

We selected Microsoft Visio to build WMGL since it supports a wide range of sophisticated graphics capabilities, an embeddable programming environment that enables developers to build custom tools, and integration with popular database management systems, such as Oracle and MySQL. Figure 3 illustrates a Visio screenshot of a partial ITS WMGL model, where warehouse model elements are available from the left-side master panel and the right-side panel contains a drawing that represents a warehouse configuration consisting of two moving angle belts, three cranes, four storage racks, two folk lifts and two gates. Modeling a warehouse in WMGL involves drawing the concrete warehouse physical structure and then adding customized properties (such as capacity, size, etc) to transportation and storage facilities model elements. Warehouse modelers can also specify the reachable range of particular transportation units (*e.g.*, forklifts and cranes) visually and define various properties (*e.g.*, capacity, heating or cooling) of storage locations.

One benefit of WMGL is its ability to validate location-related constraints automatically to ensure that the physical layout and configuration of the warehouse is valid at design-time, rather than run-time. For example, when a crane is positioned over a storage location, the WMGL model interpreter can ensure that the crane is capable of reaching all the storage cells of the location. When warehouse modelers mistakenly model a transportation facility or a storage facility that is isolated from the rest of the warehouse transportation facility network, the WMGL model checker will warn the modelers before generating code.

After WMGL has validated a model, it can generate C++ or Java code for warehouse configuration artifacts, such as lookup tables for transportation route calculation, lookup data for storage facility utilization planning, and schedules for warehouse maintenance. Different domain-specific concerns captured by WMGL can be extracted from the model and used to generate code artifacts that ITS components based on CIAO middleware can subsequently use to populate the databases, construct the warehouse connectivity graph, and initialize the backend databases by using generic database access libraries, such as the Open Database Connectivity (ODBC) Template Library (OTL). After running the WMGL model interpreter, the ITS can proceed with component deployment and configuration process described in Section 3.2.

Evaluating WMGL for ITS. WMGL significantly increased the productivity of application developers. For example, in our ITS product instance scenario shown in Figure 2, the WMGL model interpreter automatically synthesizes ∼6,300 lines of C++ code (which is over 90% of the total code) that describes the warehouse layout and artifact property information. The generated code is then used by CIAO components to populate databases before the ITS system is actually deployed.

3.2 Addressing ITS Component Deployment and Configuration Concerns

As discussed in Section 2, our ITS case study is a DRE system created using components developed to run on CIAO QoS-enabled component middleware. QoS-enabled component middleware introduces new complexities, however, that stem from the need to deploy component assemblies into the appropriate DRE system target nodes while simultaneously initializing and configuring components to enforce end-to-end QoS requirements of component assemblies. Below, we discuss how we applied CoSMIC's MDD tools to resolve key deployment and configuration challenges that arose when developing our ITS case study using CIAO.

3.2.1 Simplifying ITS Deployment and Configuration Profile Design

Deploying an ITS product instance into a warehouse involves configuring the functional and QoS behavior of its software components and deploying them throughout the underlying hardware and software infrastructure. Like most other DRE systems, an ITS is assembled from many independently developed reusable components, as described in Section 2.2. These components must be deployed and configured so that (1) assemblies meet ITS operational requirements and (2) interactions between the components meet ITS QoS requirements.

Developers must address a number of crosscutting concerns when deploying and configuring component-based ITS applications, including (1) identifying dependencies between component implementation artifacts, such as the OperatorConsole component having dependencies on other ITS components (*e.g.*, the WorkflowManager component) and other third-party libraries (*e.g.*, the QT library, which is a cross-platform C++ GUI library compatible with the Embedded Linux OS), (2) specifying the interaction behavior among ITS components, (3) specifying components to configure and control various resources, including processor resources, communication resources, and memory resources, and (4) mapping ITS components and connections to the appropriate nodes and networks in the target environment where the ITS will be deployed.

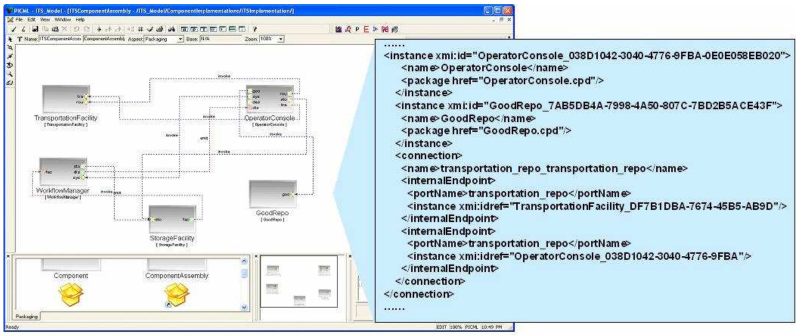
Problem: Ad hoc deployment and configuration design for diverse system requirements. A large-scale DRE system like ITS often require the creation of assemblies containing thousands of components. Conventional techniques for deploying and configuring such component-based systems can incur both inherent and accidental complexities. Common inherent complexities involve ensuring syntactic and semantic compatibility, *e.g.*, only connecting ports of components in an ITS assembly with matching types. Common accidental complexities stem from using *ad hoc* techniques for writing and modifying middleware and application configuration files, such as handcrafting XML files describing component metadata (*e.g.*, the hundreds of connections between components in ITS assemblies), which are very large, even for relatively simple groups of connected components. *Ad hoc* techniques are tedious and error-prone, making it hard to adapt the ITS to new deployment and configuration requirements, such as another warehouse that may have different types of transportation units or ITS operator console GUI terminals.

Solution: MDD-based deployment and configuration of ITS components. In our ITS project, system deployment and configuration is performed via the *Platform-Independent Component Modeling Language* (PICML) [16], which is a DSML in the CoSMIC toolsuite that works together with the DAnCE middleware to implement the OMG D&C specification [15]. PICML provides capabilities to handle complex component engineering tasks, such as multi-aspect visualization and manipulation of components and the interactions of their subsystems, component deployment planning, and hierarchical modeling and generation of component assemblies. PICML itself uses the Generic Modeling Environment (GME) [17], which is a metaprogrammable development environment for building and processing DSMLs.

PICML allows modelers to define component interfaces and component compositions, and establish connections among components visually. In our ITS, for example, PICML is used to model the interaction behavior among the ITS components, such as the facet/receptacle interface port connection between WorkflowManager component and the StorageFacility component, and the event source/sink connections between the WorkflowManager component and TransportationUnit components. These interacting components are connected together to form a valid component assembly. The semantic rules associated with component assemblies are enforced by constraints defined in PICML's metamodel and model interpreter. Its metamodel defines static semantic rules that determine valid connections between components. PICML's model interpreters ensure the dynamic semantics of models specified by users, *e.g.*, they can analyze models for various well-formedness properties and synthesize code for components and their XML descriptors that convey metadata needed by DAnCE.

In the context of ITS, a major cause of missed deadlines is priority inversions, where lower priority requests access a resource at the expense of higher priority requests. Priority inversions must be prevented or bounded since they can cause some critical paths in the ITS system to miss their deadlines. To reduce priority inversions, we use PICML to configure Real-time CORBA policies of the ITS component instances, which include (1) processor resources via priority mechanisms, thread pools, and synchronizers, for real-time components with fixed priorities, (2) communication resources via protocol properties and explicit bindings to server objects using priority bands and private connections, and (3) memory resources via bounding the size of request buffers and thread pools.

After using PICML to create component assemblies for our ITS based on warehouse-specific deployment and configuration requirements, we used it to generate the metadata needed to deploy the ITS assemblies. As shown in Figure 4, this metadata includes the list of implementation artifacts associated with each component instance, the list of connections between the different component instances, the organization of the application into different levels of hierarchy, and the default properties with which each component instance is initialized. PICML generates the different types of metadata in the form of XML descriptors that are tedious and error-prone to write manually. This metadata is used by DAnCE to drive the deployment of the complete ITS applications, which is explained in Section 3.2.2.



```
......
<instance xmi:id="OperatorConsole_038D1042-3040-4776-9FBA-0E0E058EB020">
   <name>OperatorConsole</name>
   <package href="OperatorConsole.cpd"/>
  </instance>
  <instance xmi:id="GoodRepo_7AB5DB4A-7998-4A50-807C-7BD2B5ACE43F">
   <name>GoodRepo</name>
   <package href="GoodRepo.cpd"/>
  </instance>
  <connection>
   <name>transportation_repo_transportation_repo</name>
   <internalEndpoint>
    <portName>transportation_repo</portName>
    <instance xmi:idref="TransportationFacility_DF7B1DBA-7674-45B5-AB9D"/>
   </internalEndpoint>
   <internalEndpoint>
    <portName>transportation_repo</portName>
    <instance xmi:idref="OperatorConsole_038D1042-3040-4776-9FBA"/>
   </internalEndpoint>
  </connection>
</connection>
......
```

Figure 4: Partial PICML Assembly Model for ITS

Evaluating PICML for ITS. We applied PICML to a warehouse scenario where 193 ITS components are deployed across 26 physical nodes, as described in Figure 2. Based on the deployment decisions discussed earlier, ∼400 connections must be established among these component ports. All these connections are specified by using two types of XML descriptor files: component interface descriptors and component implementation descriptors. To create a deployment profile for this scenario is prohibitively tedious and error-prone without tool support, *i.e.*, the XML files are hard to write manually since cross-referenced identifiers specify the component connections in accordance with the OMG's D&C standard.

In contrast, it was much easier to create a PICML model for these ITS connections visually, rather than writing XML files manually. The PICML packaging interpreter generates all six types of descriptor files described above, with a total number of 582 XML files averaging ∼25 lines per file. Generating the deployment automatically via PICML's model interpreter enforces the correct-by-construction paradigm in component-based application development, which eliminates a common source of errors [13]. The effort we saved in applying PICML to a warehouse deployment scenario shown in Figure 2 was more than 300 developer hours, which freed us to focus on more strategic aspects of our ITS case study. Moreover, the diversity of services and variability of QoS requirements supported by the ITS requires tremendous development effort. PICML shields developers from the inherent complexities of the configuration code and allows them to concentrate on the business logic of application components.

3.2.2 Automating ITS Deployment Process to Ensure QoS Requirements

Deployment is the sequence of activities that occurs between (1) the acquisition of software and its associated configuration and deployment metadata and (2) the actual execution of software in a target environment based on the acquired software and associated metadata. Likewise, configuration is the process of mapping known variations in the application requirements space to known variations in the software (and particularly the middleware) solution space. To complete the deployment and configuration of an ITS application, DAnCE uses the metadata generated by PICML (Section 3.2.1) to describe the concerns from multiple actors and combines them in the target environment to enforce overall system QoS requirements.

To deploy an ITS assembly, ITS developers must perform four tasks based on the deployment profile, including (1) *preparation*, which takes the pre-built ITS software package and brings it into a component software repository under the deployer's control, (2) *installation*, which downloads the ITS components to component server processes that run in each node in the target environment, including embedded system nodes used to host TransportUnit components and PC nodes that host other types of ITS components, such as WorkflowManager and OperatorConsole, (3) *configuration*, which customizes QoS properties of components and containers on each node, and (4) *launching*, which connects the ports of ITS components that are distributed throughout the target environment and bring the entire system into running mode.

Problem: *Ad hoc* deployment mechanisms for variable ITS deployment requirements. In large-scale DRE systems, QoS requirements (such as low latency and bounded jitter) are often important considerations during the deployment process since component (re)deployment may occur throughout the lifecycle of a large-scale system. To enforce these QoS requirements, components, containers, and component servers must be configured in accordance with real-time QoS properties. Component deployment and configuration tools must therefore be able to (1) specify the middleware configurations needed to configure components, containers, and component servers and (2) set the QoS policy options provided by the underlying middleware into semantically consistent configurations. For instance, in our ITS case study, whenever a ConveyorBelt component's hardware fails, it should notify the WorkflowManager in real-time to minimize/avoid damage. Likewise, ITS ConveyorBelt and Crane components may need to be collocated with WorkflowManager in some assemblies to minimize latency.

The existing OMG D&C specification does not support real-time QoS policies. Moreover, QoS-enabled object-oriented middleware, such as Real-time CORBA and Real-time Java, lacks the flexible higher level abstractions that component middleware provides to separate real-time policy configurations from application functionality. As a result, conventional object-oriented middleware platforms for deploying large-scale DRE systems incur inherent complexities stemming from *ad hoc*, tightly coupled deployment mechanism to ensure system QoS. To ensure real-time QoS requirements for large-scale DRE systems manually and programmatically is not only tedious and error-prone, but also makes deployment artifacts and effort hard to reuse, *e.g.*, there is no easy way to reconfigure a warehouse to accommodate the variability. It is therefore hard for ITS developers to configure, validate, modify, and evolve their systems consistently using QoS-enabled object-oriented middleware.

Solution: Extending the OMG D&C specification to support real-time QoS policies. DAnCE extends the OMG D&C specification to enable the configuration of real-time QoS policies through metadata generated by PICML. The overall architecture of DAnCE is shown in the right side of Figure 5. This figure also shows how ITS developers can model various warehouse D&C concerns via PICML, which then automatically generates the corresponding D&C profile for the designated system. DAnCE then takes the generated profile and automatically deploys the system into CIAO, thereby bridging the gap between higher-level MDD tools and lower-level middleware runtime platforms.

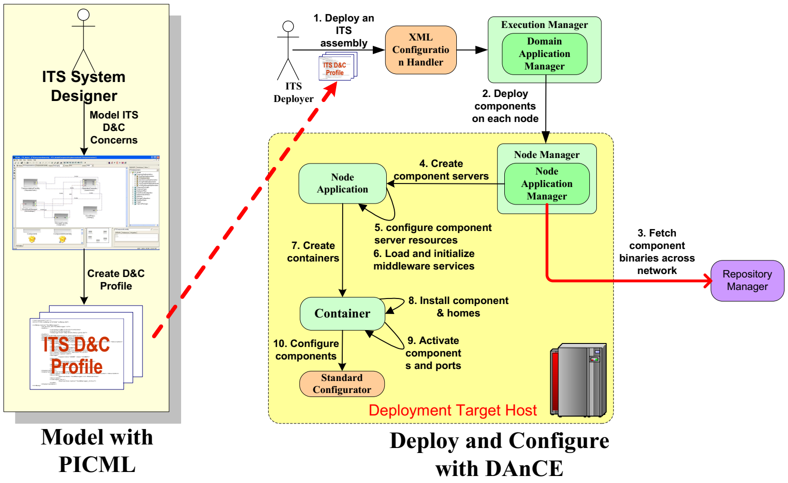


Figure 5: DAnCE Architecture and PICML Relationship

As shown in Figure 5, DAnCE consists of implementations of a set of runtime services that deal with the instantiation, installation, configuration, monitoring, and termination of components on the nodes of the target environment. Some services (such as ExecutionManager and DomainApplicationManager) run at the global domain level, whereas others (such as NodeManager and NodeApplicationManager) run on each node. These services together manage the lifecycle of the ITS deployment process to help configure component servers on the individual nodes, install components into containers, and set up connections among components.

When an ITS deployer instructs a global interface to deploy an ITS assembly, he/she must give the XML-based deployment profile generated by the PICML MDD tool. The global interface then uses this profile to populate a global deployment plan that describes a mapping of a configured ITS assembly into a target domain. This plan includes information about nodes where components will be deployed, the mapping of component to nodes, the QoS configuration of components, the connections among component instances, and the process collocation strategy settings. Depending on the total number of nodes needed for a particular deployment, the global deployment service then split the global plan into multiple local (node-level) deployment plans and passes them to each node-level service, based on the specification in the PICML model of the ITS.

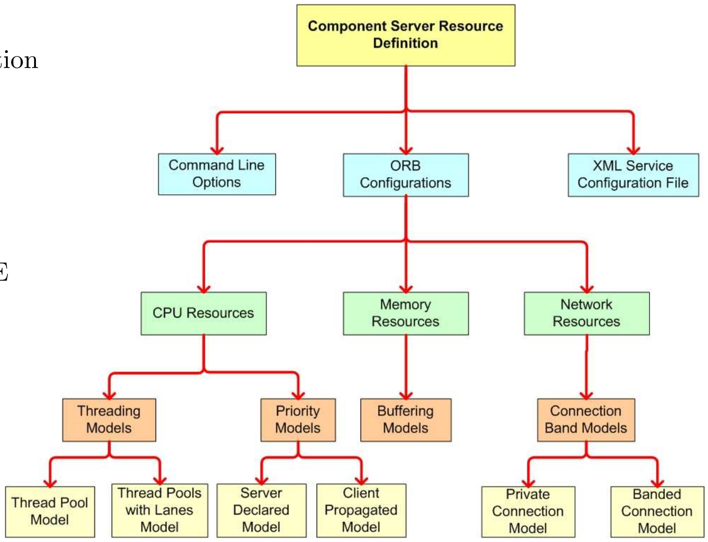


Figure 6: Specifying Real-time QoS Requirements

To enforce QoS requirements, DAnCE extends the OMG D&C specification to define Node-Application server resource configurations, which heavily influence end-to-end QoS behavior.

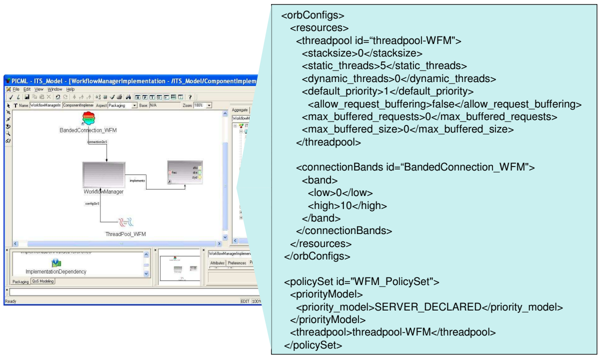
Figure 6 shows the different categories of server configurations that can be specified using the DAnCE *server resources XML schema*, which are related to system end-to-end QoS enforcement. Each server resources specification can set the following options: (1) *ORB command-line options*, which control TAO's connection management models, protocol selection, and optimized request processing, (2) *ORB service configuration option*, which specify ORB resource factories that control server concurrency and demultiplexing models. Using this XML schema, a system deployer can specify the designated ORB configurations.

ITS components are hosted in containers created by the NodeApplication process, which provides the run-time environment and resources for components to execute and communicate with other components in a component assembly. The ORB configurations defined by the *server resources XML schema* are used to configure NodeApplication processes that host components, thereby providing the necessary resources for the components to operate. For example, some ITS components, such as `ItemLocationSensor` and `WorkflowManager`, have stringent

QoS requirements since they handle real-time item delivery activities.

Figure 7 shows an example XML document that specifies the server resource configurations defined by an ITS deployer. To honor the specified real-time configurations, the component servers hosting these components are configured with server-declared priority model with the highest CORBA priority, thread pools with a preset number of static real-time threads, as well as priority-banded connections.

Evaluating DAnCE for ITS. Without DAnCE tool support it is prohibitively hard to manually (re)install all 193 components on 26 nodes in our ITS case study shown in Figure 2 while taking into consideration the heterogeneous software/hardware platforms, and variable component QoS configuration. It would require ∼20,000 lines of code to accomplish this task, and the code would be scattered throughout the ITS system software since many QoS configuration tasks are embedded into component implementation code, hence it is extremely hard to accommodate any warehouse reconfiguration changes.



```
<orbConfigs>
  <resources>
    <threadpool id=“threadpool-WFM">
      <stacksize>0</stacksize>
      <static_threads>5</static_threads>
      <dynamic_threads>0</dynamic_threads>
      <default_priority>1</default_priority>
        <allow_request_buffering>false</allow_request_buffering>
      <max_buffered_requests>0</max_buffered_requests>
      <max_buffered_size>0</max_buffered_size>
    </threadpool>

    <connectionBands id=“BandedConnection_WFM">
      <band>
        <low>0</low>
        <high>10</high>
      </band>
    </connectionBands>
  </resources>
 </orbConfigs>

 <policySet id="WFM_PolicySet">
  <priorityModel>
    <priority_model>SERVER_DECLARED</priority_model>
  </priorityModel>
  <threadpool>threadpool-WFM</threadpool>
 </policySet>
```

Figure 7: Example Resource Configuration

In contrast, by using DAnCE in conjunction with PICML, the entire component deployment process can be automated and simplified for ITS deployers. Moreover, when the warehouse is reconfigured, deployers need only extend the existing ITS WMGL and PICML models. The new ITS can then be redeployed and configured correctly without tedious manual activities.

4 Evaluation

Our motivation for integrating MDD tools with QoS-enabled component middleware in the ITS case study was to (1) construct software-intensive DRE systems using higher-level visual models that facilitated more effective analysis, verification, and better productivity than handcrafting software using third-generation languages and (2) provide more reusable and composable abstractions than using conventional object-oriented real-time middleware platforms for DRE systems. This section evaluates how well we achieved these goals.

4.1 Verifiability

When developing large-scale DRE systems – especially mission- or safety-critical systems – it is essential that system be verified and validated against certain pre-defined domain rules before the product is deployed. As described in Section 3.1, the WMGL DSML generated and ensured the syntactic correctness of warehouse initialization code that precisely define the physical layout of the warehouse, characteristics of warehouse storages and facilities. Moreover, the embedded model checker of WMGL performed semantic analysis, such as reachability analysis, of the warehouse configuration to ensure that each warehouse storage cell could be reached by least one transportation unit. In addition, WMGL encapsulated domain knowledge from warehouse domain experts in the form of additional advanced warehouse model checkers. Prior to this MDD tool support, warehouse initialization and configuration scripts were handcrafted manually, which was tedious and error-prone.

Prior work [7, 10, 13] has shown that it is hard for DRE developers to keep track of many complex dependencies when configuring and deploying large-scale systems, even when using component middleware. Without MDD tool support, therefore, the effort required to deploy large-scale DRE systems like ITS involves hand-crafting deployment descriptor metadata in an *ad hoc* manner. Addressing this challenge effectively requires techniques that can analyze, validate, and verify functional and QoS system properties.

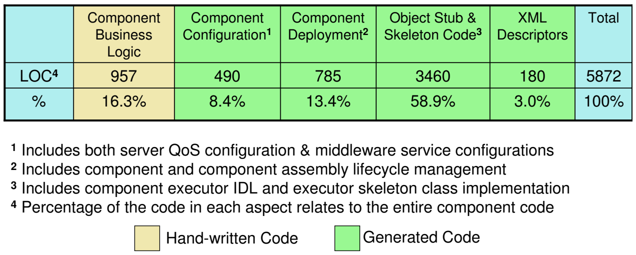
In our ITS case study, simple CORBA objects and more sophisticated CCM components must coexist, which introduces complexities in interface definition of components and interaction definitions between components. For example, while CORBA object interfaces can support multiple inheritance, CCM components can only have only a single component parent, so equivalent units of composition (*i.e.*, interfaces in CORBA objects and components in CCM) can have subtle semantic differences. Without automated support from MDD tools like PICML, developers would have no systematic way to specify interconnections and configurations. Manually configuring the systems via *ad hoc* techniques can overwhelm developers since many configuration aspects are tangled with each other which spread throughout different subsystems and layers of the ITS and component middleware itself. When using PICML, however, the problems associated with multiple inheritance and semantics of tangled configuration can be detected and resolved at design time.

4.2 Reusability and Composability

Reusability and composability are important requirements, particularly for developing large-scale DRE systems like ITS. Diversified operating policies of ITS components can be customized based on a particular warehouse deployment scenario. In QoS-enabled object-oriented middleware, such as Real-time CORBA and Real-time Java, these configurations are made imperatively via invocations on programmatic configuration interfaces. The drawbacks with object-oriented approaches to reusability, however, are (1) impeded reusability due to tight coupling of application logic with specific QoS properties, such as event latency thresholds and priorities [18], and (2) reduced composability due to hard coded application interfaces, such as those integrated with particular types of middleware services.

In contrast, component middleware, such as CIAO and DAnCE, enhances reusability and composability by using metaprogramming techniques (such as XML descriptor files) to specify component configuration and deployment concerns declaratively. This approach enables QoS requirements to be specified later (*i.e.*, just before runtime deployment) in a system's lifecycle, rather than earlier (*i.e.*, during component development). When deploying an ITS product instance, DAnCE parses the given XML configuration files and make appropriate invocations on the corresponding service configuration interfaces, which is useful for DRE systems like ITS that require changeable customized QoS configurations for variable target OS, network, and hardware platforms with different capabilities and properties.

Figure 8 shows the weight distribution of a typical ITS component WorkflowManager based on its functionality and measured through *lines of code* (LOC) and the percentage of code relative to the entire component lifecycle. Other types of ITS components have similar weight distributions. As illustrated in Figure 8, the amount of code related to the WorkflowManager component can be classified into five categories: *Component Business Logic*, *Component Configuration*, *Component Deployment*, *Object Stub & Skeleton Glue code*, and *XML descriptors*. Each category implements different aspect of components during their lifecycles.



	Component Business Logic	Component Configuration[1]	Component Deployment[2]	Object Stub & Skeleton Code[3]	XML Descriptors	Total
LOC[4]	957	490	785	3460	180	5872
%	16.3%	8.4%	13.4%	58.9%	3.0%	100%

[1] Includes both server QoS configuration & middleware service configurations
[2] Includes component and component assembly lifecycle management
[3] Includes component executor IDL and executor skeleton class implementation
[4] Percentage of the code in each aspect relates to the entire component code

🟨 Hand-written Code 🟩 Generated Code

Figure 8: Weight of WorkflowManager Component in ITS

With object-oriented middleware, the *Object Stub and Skeleton Code* is generated by the IDL compiler that shields application developers from low-level network programming details. This code accounts for 58.9% of the entire implementation. The rest of the code (more than 40% of the implementation) must be hand written by the application developers. With the component-based approach, conversely, most the code in the *Component Configuration* and *Component Deployment* categories can be factored into reusable CIAO component middleware infrastructure and then configured by MDD tools like PICML and D&C frameworks like DAnCE. For our ITS product instance shown in Figure 2, where there are 8 components types and 193 component instances, more than ~13,500 lines of code that were previously handcrafted are now refactored into reusable component frameworks and configured and deployed using PICML and DAnCE.

5 Related Work

Our work on MDD extends earlier work on Model-Integrated Computing (MIC). Examples of MIC technology used today include GME [17] and Ptolemy [19] (used primarily in real-time and embedded domains) and Model

Driven Architecture (MDA) [20] based on UML and XML. Kennedy Carter's iUML Product Suite [21] supports the Executable UML process from textual requirements management through modeling to complete target code generation. The Rhapsody System Designer tool by I-Logix (www.ilogix.com) is based on OMG's MDA and UML 2.0 specification, and provides complete application generation from UML models, and the generated application could be in multiple programming languages including C/C++, Java or Ada, and multiple middleware platforms including CORBA and COM. Both tools have proved many successes in developing DRE systems. Unlike our approach, however, these MDD tools are based on OMG's UML specification which is still in the low-level solution space. In contrast, our work combines GME metamodeling mechanisms and UML capability to model and synthesize component middleware for configuring and deploying DRE systems, which better unifies the problem space and solution space.

The *Embedded Systems Modeling Language* (ESML) [22] was developed by ISIS at Vanderbilt University to provide a visual metamodeling language that captures multiple views of embedded systems, allowing a diagrammatic specification of complex models. The modeling building blocks include software components, component interactions, hardware configurations, and scheduling policies. The user-created models can be fed to analysis tools (such as AIRES, VEST, and Cadena) to perform schedulability and event analysis. Using these analyses, design decisions (such as component allocations to the target execution platform) can be performed. Unlike PICML, ESML is platform-specific since it is customized for the Boeing Bold Stroke PRiSm QoS-enabled component model [10, 23]. ESML also does not support nested assemblies and the allocation of components are tied to processor boards, which is a proprietary feature of the Bold Stroke component model.

Cadena [24] is a MDD tool developed within Eclipse Modeling Framework (EMF) for building component-based DRE systems, with the goal of applying static analysis, model checking, and lightweight formal methods to enhance these systems. Unlike our work on CoSMIC, however, Cadena does not support activities such as component packaging, generating deployment plan descriptors, and hierarchical modeling of component assembly, thus it introduces additional burden to DRE application developers to accomplish such tasks. In our work, these aspects could be captured through PICML MDD tool and then all the deployment and configuration work can be automated using DAnCE. We are collaborating with the Cadena researchers to create an integrated suite of MDD tools citeSchmidt:04v.

6 Concluding Remarks

This section focuses on our experience gained and lessons learned when integrating MDD tools and QoS-enabled component middleware technologies and applying them to a case study in the warehouse management domain. In general, our development and experiments of ITS case study shows the complementary relationships between

CoSMIC MDD tools (*i.e.*, WMGL/PICML) and underlying component middleware and D&C frameworks (*i.e.*, CIAO/DAnCE). The lessons learned 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 large-scale DRE applications. For example, the OMG Lightweight CCM [5] and Deployment and Configuration (D&C) [15] specifications have a large number of configuration points, which we handle with advanced MDD tools.
- **Large-scale systems are rarely developed from scratch and often are composed of smaller systems.** The MDD paradigm expedites application development with the proper integration of the modeling tool and underlying technical infrastructure, such as the DAnCE D&C framework. In our ITS case study, if the warehouse model is the primary changing concern in the system (which is typical for end users), little new application code must be written, yet the complexity of the generation tool remains manageable due to the limited number of well-defined configuration *hot spots* exposed by the underlying infrastructure. Likewise, when component deployment plans are incomplete or must change, the effort required is significantly less than using the raw component middleware without MDD tool support since applications can evolve from the existing set of PICML and WMGL models.
- **Domain-specific modeling techniques can help reduce the learning curve for end users.** For example, warehouse modelers in our ITS project needed little or no knowledge of how to write component software since they used higher-level models that correspond to the language understood by domain engineers and visual modeling environments, such as WMGL. In addition to the warehouse configuration aspects, WMGL embodies certain assumption and rules about the mapping (usage patterns) from problem domain of warehouse management to the solution domain of component middleware. The mapping rules understood by WMGL are defined by experienced software architects and then enforced by the WMGL modeling and code generation environment. These enforcement mechanisms reduce the probability of architectural rules violation discussed in Section 2 and ensure the proper usage of component middleware. Conversely, our experience applying PICML to model the ITS deployment structure indicated that it raised the level of abstraction at which developers work, enabling them to concentrate on certain aspects (*e.g.*, deployment structure) in the multidimensional problem space associated with applying component middleware for DRE systems. This separation of concerns in turn eliminated many sources of accidental complexities and improved overall system quality.

While we were generally successful in integrating and applying MDD and component middleware into our ITS case study, we also felt addition R&D is warranted to enable broader adoption of these technologies:

- **How to optimally and effectively configure and deploy each product instances.** Despite the fact that PICML facilitates the configuration of enterprise DRE systems based on Real-time CORBA and Lightweight CCM, developers are still faced with the question of what constitutes a "good" configuration and they are still ultimately responsible for determining the appropriate configurations. We observed that scheduling analysis and verification tools would be helpful in performing this evaluation and should be integrated into CoSMIC's MDD toolsuite to help system designers address such challenges. In addition, despite the benefits of using visual MDD tools to describe different aspects of the large scale DRE systems, it is still labor intensive and error-prone to manually show all ∼400 connections for the number of components in our ITS case study. This observation motivates the need for further research in automating the synthesis of large-scale DRE systems based on the different types of meta-information about assembly units, such as components or services.

- **How to reuse legacy code when transitioning from object-oriented to component-based architectures.** Porting existing object-oriented DRE systems to component-based systems is an important concern for production systems that achieve reuse in an *ad hoc* manner at the source code level. We observed that such reuse is ineffective without advanced MDD tools to support this porting effort. Our experience with the ITS case study indicated that these tools should enable application developers to identify (1) which part of the legacy code originally compliant to traditional object-oriented middleware is still compliant to the new component-based middleware, (2) which part of the legacy code is no longer needed since it has been factored into reusable component middleware frameworks, and (3) which part of the legacy code should be refactored and configured through other tools and frameworks, such as PIMCL and DAnCE. Developing these tools requires analysis capability for both the object-oriented and component-based programming models at the source code level.
- **How to handle challenges associated with domain evolution.** Using MDD technologies effectively requires practical and scalable solutions to the domain evolution problem, which arises when existing software products are extended and/or refactored to handle unanticipated requirements or better satisfy existing requirements. For example, changing metamodels can invalidate models based on previous versions of the metamodels. While software developers can manually update their models and/or components developed with a previous metamodel to work with the new metamodel, this approach is clearly tedious, error-prone, and non-scalable. Although MDD tools remove many complexities associated with handcrafted solutions, developers are still faced with the challenge of evolving existing models when the respective domain evolves. Although model evolution tools, such as GReAT [25], exist they are hard to use and only provide partially automated solutions. Since these modifications significantly complicate domain evolution efforts, they can outweigh the advantages of system development compared to traditional development methods based on handcrafting solutions using third generation languages. To rectify these problems, compositional architecture patterns are desired to guide the design of MDD tools component middleware, so they can modularize system concerns and reduce the effort associated with domain evolution.

Our experiences described above motivate the need for further research on automated techniques to uncover effective heuristics to guide the complicated process of developing and evaluating DRE systems, reusing legacy code, and migrating models and metamodels as knowledge of a domain expands. We are exploring all of these open R&D issues in the context of our ITS case study and other large-scale DRE system projects.

References

[1] Object Management Group, *Real-time CORBA Specification*, OMG Document formal/02-08-02 ed., Aug. 2002.

[2] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull, *The Real-time Specification for Java*. Addison-Wesley, 2000.

[3] *Evaluating Real-time Java for Mission-Critical Large-Scale Embedded Systems*, (Washington, DC), IEEE Computer Society, May 2003.

[4] V. Subramonian, L.-J. Shen, C. Gill, and N. Wang, "The Design and Performance of Dynamic and Static Configuration Mechanisms in Component Middleware for Distributed Real-time and Embedded Systems," in *The 25th IEEE Real-time Systems Symposium (RTSS)*, (Lisbon, Portugal), Dec. 2004.

[5] Object Management Group, *Lightweight CCM RFP*, realtime/02-11-27 ed., Nov. 2002.

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

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

[8] T. Ritter, M. Born, T. Unterschütz, and T. Weis, "A QoS Metamodel and its Realization in a CORBA Component Infrastructure," in *Proceedings of the* 36^{th} *Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003*, (Honolulu, HW), HICSS, Jan. 2003.

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

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

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

[12] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proceedings of the IEEE*, vol. 91, pp. 145–164, Jan. 2003.

[13] A. Gokhale, K. Balasubramanian, J. Balasubramanian, A. S. Krishna, G. T. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt, "Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications," *The Journal of Science of Computer Programming: Special Issue on Model Driven Architecture*, 2005 (to appear).

[14] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. Gokhale, "DAnCE: A QoS-enabled Component Deployment and Conguration Engine," in *Proceedings of the 3rd Working Conference on Component Deployment*, (Grenoble, France), Nov. 2005.

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

[16] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems," in *Proceedings of the 11th Real-time Technology and Application Symposium (RTAS '05)*, (San Francisco, CA), pp. 190–199, IEEE, Mar. 2005.

[17] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, pp. 44–51, November 2001.

[18] G. Edwards, D. C. Schmidt, A. Gokhale, and B. Natarajan, "Integrating Publisher/Subscriber Services in Component Middleware for Distributed Real-time and Embedded Systems," in *Proceedings of the 42nd Annual Southeast Conference*, (Huntsville, AL), ACM, Apr. 2004.

[19] J. Liu, X. Liu, and E. A. Lee, "Modeling Distributed Hybrid Systems in Ptolemy II," in *Proceedings of the American Control Conference*, June 2001.

[20] Object Management Group, *Model Driven Architecture (MDA)*, OMG Document ormsc/2001-07-01 ed., July 2001.

[21] K. Carter, "Kennedy Carter iUML 2.2." `www.kc.com`, 2004.

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

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

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

[25] Karsai G. and Agrawal A. and Shi F. and Sprinkle J., "On the use of Graph Transformations in the Formal Specification of Computer-Based Systems," in *Proceedings of IEEE TC-ECBS and IFIP10.1 Joint Workshop on Formal Specifications of Computer-Based Systems*, (Huntsville, AL), IEEE, Apr. 2003.