Developing Product-lines for Distributed Real-time and Embedded Systems with Modeling Tools and Component Middleware: A Case Study

Andrey Nechypurenko
Siemens Corporate Technology
Siemens AG
Wittelsbacherplatz 2
Munich, D-80333 Germany

Gan Deng
Department of EECS
Vanderbilt University
2015 Terrace Place
Nashville, TN 37203 USA

Douglas C. Schmidt
Department of EECS
Vanderbilt University
2015 Terrace Place
Nashville, TN 37203 USA

Aniruddha Gokhale
Department of EECS
Vanderbilt University
2015 Terrace Place
Nashville, TN 37203 USA

ABSTRACT

Developing software for product-line architectures (PLAs) in 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 hardware, programming languages, middleware, and OS platforms, (2) fine tuning each product instance to satisfy customer requirements, such as real-time quality-of-service constraints, and (3) changing functional and non-functional aspects of product instances based on available system resources. This paper presents experience gained and lessons learned addressing domain- and middleware-specific variability when applying model-driven development (MDD) tools and component middleware technologies to develop a PLA for an inventory tracking system, which manages the storage and flow of items in warehouses. Our experience shows that (1) coherent integration of MDD tools and component middleware provides a more productive software process for developing PLA-based DRE systems than using component middleware without MDD tools and (2) significant challenges remain that must overcome to apply these technologies to a broader range of PLA-based DRE systems.

Keywords

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

1. Introduction

Software *product line architectures* (PLAs) [14] are a promising technology for industrializing software development by focusing on the automated assembly and customization of domain-specific components, rather than (re)programming systems manually. Conventional PLAs consist of *component frameworks* [19] as core assets, whose design captures recurring structures, connectors, and control flow in an application domain, along with the points of variation explicitly allowed among these entities. PLAs are typically designed using *commonality/variability analysis* (CVA) [12], which captures key characteristics of software product lines, including (1) *scope*, which defines the domains and context of the PLA, (2) *commonalities*, which describe the attributes that recur across all members of the family of products, and (3) *variabilities*, which describe the attributes unique to the different members of the family of products.

Emerging trends and challenges. Standards-based quality-of-service (QoS)-enabled object-oriented middleware technologies, such as Real-time CORBA [9] and Real-time Java [8], 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. Using these object-oriented middleware technologies to build reusable components for PLAs is still hard, however, because objects based on such technologies often encapsulate tangled functional aspects and non-functional aspects.

In recent years, QoS-enabled *component* middleware [1, 2, 3, 6] has emerged to help developers of DRE systems factor out reusable concerns to enhance reuse, such as component life-cycle management, authentication/authorization, and remoting. As a result, software for large-scale DRE systems is increasingly being assembled from reusable modular components available from commercial-off-the-shelf (COTS) providers, rather than developed manually from scratch. Although QoS-enabled component middleware technology provides powerful capabilities, it also yields the following challenges for developers of PLAs for DRE systems [20, 3]:

- **Increased scale**. As DRE systems are joined together to form large-scale *systems of 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, and violate architectural principles.
- **Increased variability**. Additions to the physical features of the system and/or availability of better implementations of the same type of systems can further increase functional and non-functional variability.

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

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) [11, 5, 3, 13] techniques with QoS-enabled component middleware [2, 6]. 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 PLAs for particular 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 for PLAs in a particular domain, and (3) *model transformations and code generation* that help automate repetitive, tedious and error-prone tasks in the software PLA 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) [13], which is an integrated collection of DSMLs that support the development, deployment, configuration, and evaluation of QoS-enabled component middleware-based DRE systems and (2) a QoS-enabled component middleware platform called *Component-Integrated ACE ORB* (CIAO) [2] that combines Lightweight CORBA Component Model (CCM) [1] capabilities with Real-time CORBA features [9]. To evaluate how the *integration* of MDD tools and QoS-enabled component middleware helps resolve key challenges presented above, we recently developed a software PLA for an *inventory tracking system* (ITS), which is a representative DRE system 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 PLA: (1) warehouse configuration and management concern and (2) component management, assembly, configuration, and deployment concern.

The goal of our case study was to evaluate how well these technologies could be integrated together to (1) modularize key functional and QoS concerns 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 of our ITS case study; Section 4 evaluates our approach and codifies our experience; Section 5 compares our work with related efforts; and Section 6 presents concluding remarks.

2. Overview of the ITS Case Study

An inventory tracking system (ITS) provides reliable, efficient, and convenient mechanisms that manage warehouses and the movement of inventory items in a timely and reliable manner. Users of an ITS include couriers (such as UPS, FedEx, and DHL), airport baggage handling systems, and large trading and manufacturing companies (such as Wal-Mart and Target). 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.

Our ITS component architecture is developed in accordance with Lightweight CCM using the *Component-Integrated ACE ORB* (CIAO) [2], which is QoS-enabled component middleware built atop *The ACE ORB* (TAO) [7] Real-time CORBA Object Request Broker (ORB) that implements key patterns to meet the demanding QoS requirements of DRE systems. CIAO extends TAO by abstracting key QoS concerns (such as thread priority models, concurrency model, thread-to-connection bindings, and timing properties) into elements that can be configured separately from the application business logic to make DRE system development more flexible and productive. In addition, to automate the entire system deployment process and bridge the gap between component middleware with MDD tools, we developed a QoS-enabled *Deployment And Configuration Engine* (DAnCE) [18] that works in conjunction with CIAO to help developers configure and deploy existing pre-built components and component assemblies and customize them into reusable services.

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 Use Cases and Component Architecture

Figure 1 shows the primary actors and use cases in our ITS case study, which performs the following activities:

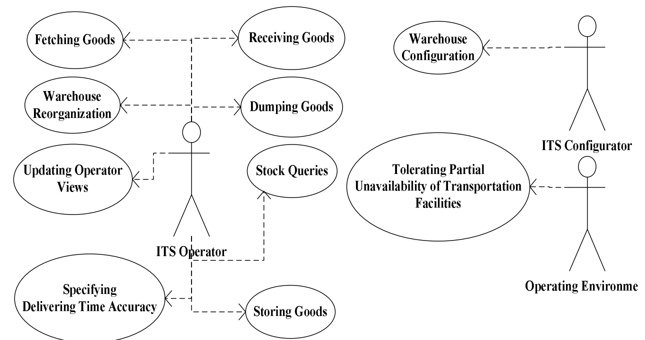

Figure 1: Use Case Diagram for the ITS Case Study

- *ITS Configurator* actors use ITS capabilities to configure the set of available facilities in certain warehouses, such as the structure of transportation belts, routes used to deliver items, and characteristics of storage facilities (e.g., whether hazardous items are allowed to be stored, maximum allowed total weight of stored items, etc).
- *ITS Operator* actors use ITS capabilities to reorganize warehouses to fit future changes, as well as receiving items, storing items into the warehouse, fetching items from the warehouse to certain locations, quering inventory, specifying delivery time deadline, and updating operator console views.
- *Operating Environment* actors use ITS 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.

Although the ITS actors and use cases described above are present in most warehouses, they can have significant variation in customer needs, warehouse specific requirements, and integration with other subsystems. 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. Software PLAs are a promising technology for addressing this variability and heterogeneity [20].

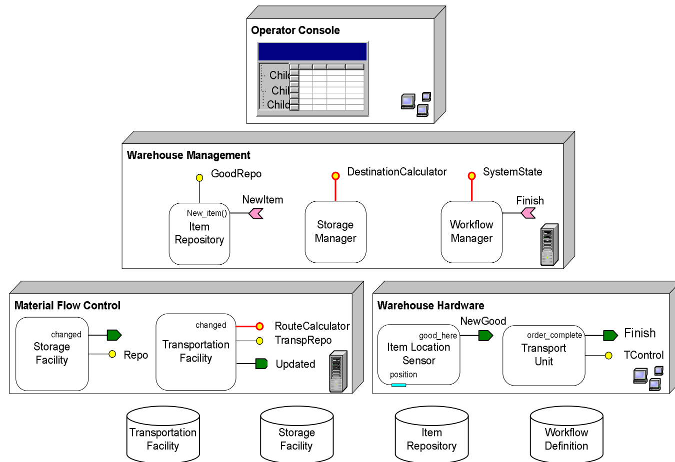

Figure 2: ITS Subsystems and Key Components

Figure 2 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 represent hardware entities, such as cranes, forklifts, and shelves. 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*). These capabilities are used in the context of their associated ITS subsystems as follows:

- The **Warehouse Management subsystem** consists of a set of high-level functionality and decision making components. This subsystem calculates the destination location and delegates other details to the Material Flow Control subsystem described below.
- The **Material Flow Control subsystem** executes high-level decisions calculated by the Warehouse Management subsystem to deliver items to the destination location. This subsystem handles all item delivery details, such as route (re)calculation and reservation of transportation and storage facilities.
- The **Warehouse Hardware subsystem** handles physical devices, such as sensors and transportation units (e.g., belts, forklifts, cranes, and pallet jacks). Each sensor device and transportation unit corresponds to a component type, such as *ItemLocationSensor* and *TransportUnit*.

The functionality of the ITS subsystems shown in Figure 3 can be monitored and controlled by one or more *OperatorConsole* components, and all persistence concerns are handled via databases.

2.2 Scale and Variability in the ITS Case Study

Implementing a PLA for an ITS requires commonality and variability analysis. For example, all transportation facilities could be represented with the same component interface, *i.e.*, *TransportUnit*, while their implementations can vary, however, in response to differences in hardware facilities for transporting certain types of items, as well as different positioning precision and transportation speeds. 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 1,000's of software/hardware components.

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**

Table 1: Characteristics of an Example ITS Product Instance

Table 1 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. It 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 *TransportUnit*s. These 193 components are deployed into 191 processes, which in turn are hosted in 26 physical nodes. All components run in separate processes except in two collocated cases: *ItemRepository/StorageManager* and *TransportationFacility/StorageFacility*. 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. Developing ITS via an Integrated MDD Tool and Component Middleware Solution

This section describes how we applied MDD tools and QoS-enabled component middleware to address scalability and variability issues in our ITS case study by enabling them to work at higher levels of abstraction than components and classes written in third-generation languages and traditional object-oriented middleware platforms, such as Real-time CORBA or Real-time Java. We applied these tools and middleware to help simplify and automate the following two concerns:

- **Modeling and synthesizing warehouse configurations**, which involves simplifying and automating (1) physical layout configuration, and (2) transportation facility network design of warehouses, and automatically populating databases to capture all 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 the configuration of middleware and applications that implement ITS functionality. Section 3.2 describes how the CoSMIC MDD tools and DAnCE were integrated together to help develop, assemble, configure, and deploy ITS software components.

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 reconfigurations of an existing warehouse. The proper configuration of an ITS heavily depends on the physical layout and transportation facilities of a warehouse, which may vary in different circumstances.

The layout information that is specified during the warehouse design phase should therefore be amendable to changes 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, which overly couples their solutions to particular warehouse configurations and technologies. Such tight couplings make an ITS product instance 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 have developed the *Warehouse Modeling and Generation Language* (WMGL), which is a DSML in the CoSMIC MDD toolsuite 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 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). The merits of this approach are that whenever the warehouse is reorganized or a new transportation facility or storage facility is added, the graph can be (re)generated automatically from the model, and all other information associated with such change would also be updated automatically.

We selected Microsoft Visio to build WMGL since it supports a wide range of sophisticated graphics capabilities and provides many pre-developed drawing types, such as graphics elements required to model warehouse transportation units (e.g., forklifts, cranes, and belts). Visio also provides an embeddable programming environment that enables developers to build custom tools, such as writing extensible model interpreters to describe the dynamic behaviors by extending the static Visio model. In addition, Visio supports integration with popular database management systems, such as Oracle and MySQL.

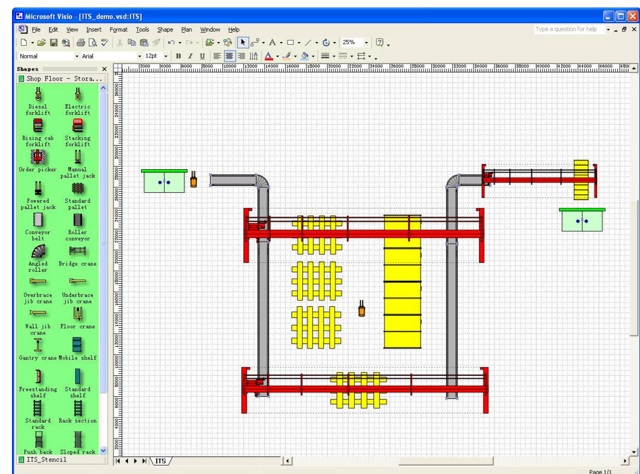

Figure 3. A Warehouse Configuration in WMGL

Figure 3 illustrates a Visio screenshot of an 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. To simplify the use of WMGL during the modeling process, whenever a warehouse artifact (such as a transportation unit or storage facility) is positioned in the warehouse model, WMGL conveys to modelers what other warehouse artifacts are interconnected with it interactively.

A key benefit of WMGL is its ability to automate the correct-by-construction transition from WMGL models to executable warehouse configurations. After creating a WMGL model, the corresponding configuration artifacts (such as lookup tables for transportation route calculation, lookup data for storage facility utilization planning, and schedules for warehouse maintenance) are generated automatically via the WMGL model interpreter.

To validate the correctness of a data model, the WMGL model checker applies analysis techniques to the warehouse model. Certain location-related constraints can be checked automatically to ensure that the physical layout and configuration of the warehouse is valid and meaningful. 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 WMGL discovers potential conflicts, it issues diagnostic messages to users. Likewise, 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 interpreter will warn the modelers before generating code.

After WMGL has validated a model, it can generate C++ or Java code to bootstrap ITS components at runtime. For example, 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 provides several benefits for bridging the gap between software vendors and customers. For example, it visually captures warehouse information (such as positions, sizes, and reachable areas), which helps reduce communication barrier. Moreover, the model analysis in the WMGL interpreter detects warehouse design faults (such as isolated storage facilities) during design rather than runtime.

WMGL significantly increased the productivity of application developers. For example, in our ITS product instance scenario shown in Table 1, the WMGL model interpreter automatically synthesizes ~6,300 lines of C++ code (which is over 90% of the total code) to describe the warehouse layout and artifact property information. This generated code is then used by CIAO components to populate ITS databases before the ITS system actually runs...

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.

3.2 Addressing ITS Component Deployment and Configuration Concerns

As discussed in Section 2, our ITS case study is based on a PLA for DRE systems created using CCM components developed to run on the CIAO Lightweight CCM middlewar platform. We recently enhanced CIAO with the *Deployment And Configuration Engine (DAnCE)* [18], which supports the OMG Deployment and Configuration (D&C) specification [4]. In this specification, deployment is the sequence of activities that occurs between (1) the acquisition of software and its associated 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 [11]. Below, we discuss how we applied MDD tools to resolve key deployment and configuration challenges that arose when developing our ITS case study using QoS-enabled component middleware.

3.2.1 Automating ITS Deployment and Configuration Profile Generation

Deploying an ITS product instance into a warehouse involves configuring the functional and non-functional behavior of its software components and deploying them throughout the underlying hardware and software infrastructure. Like 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 ITS 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, and (3) 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 profile creation for diverse system requirements. Large-scale DRE systems, such an ITS, often require the creation of assemblies containing thousands of components [3, 20]. 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 dozens of connections between components in ITS assemblies), which are very large, even for relatively simple groups of connected components. Such *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 → Model-driven deployment and configuration of ITS components. In our ITS project, system deployment and configuration is performed via the *Platform-Independent Component Modeling Language* (PICML) [13], which is a DSML in the CoSMIC toolsuite that works together with the DAnCE middleware to implement the OMG D&C specification. For example,

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. We developed PICML using the *Generic Modeling Environment* (GME) [5], 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 *OperatorConsole* component. 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 metadata.

PICML contains multiple model interpreters, each performing a particular function. The most commonly used interpreter for our ITS case study is the *packaging interpreter*, which generates XML descriptors to address various concerns in the CCM D&C specification. These XML descriptors include (1) *component interface descriptors*, which capture information about component interfaces including component ports, (2) *component implementation descriptors*, which capture information about component implementations, such as the dependencies and the connections among components, (3) *implementation artifact descriptors*, which capture information about implementation artifacts including dependencies between such artifacts, (4) *component package descriptors*, which capture information about grouping of multiple implementations of the same component interface into component packages, (5) *package configuration descriptors*, which capture information about specific configurations of such component packages, and (6) *component domain descriptors*, which capture information about the target environment in which the component-based application will be deployed.

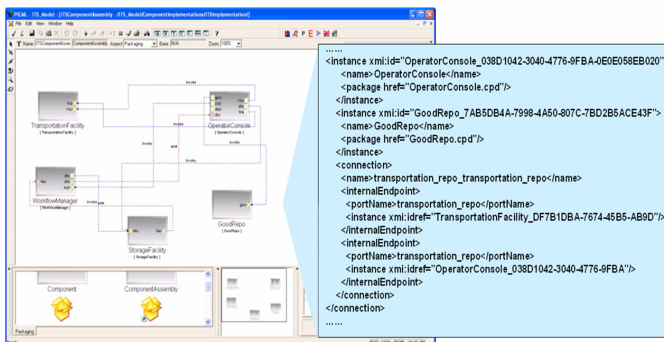

Figure 4. Partial PICML Assembly Model for ITS

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, PICML could be used to configure real-time policies of the ITS component instances, such as those defined in Real-time CORBA.

After using PICML to create component assemblies for our ITS based on warehouse-specific deployment and configuration requirements, we used its packaging interpreter 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's packaging interpreter 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 the subsection 3.2.2 of the paper.

Evaluating PICML for ITS. In our ITS case study, we applied PICML to a warehouse scenario where 193 ITS components are deployed across 26 physical nodes, as described in Table 1. 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 Table 1 was more than 300 developer hours, which freed us to focus on more strategic aspects of our ITS case study.

More generally, 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.

3.2.2 Automate ITS Component Deployment to Target Warehouse Environment

To complete the deployment of an ITS application, it is necessary to take the metadata describing the concerns from multiple actors and bring them together in the target environment. Section 3.2.1 explained how the PICML MDD tool addressed key concerns in the ITS component configuration and assembly phase by automatically and correctly synthesizing various types of XML de-

scriptors. These XML descriptors then form a *profile* that specifies system deployment requirements.

To deploy an ITS assembly, deployers 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 properties of components on each node based on metadata in the deployment profile, and (4) *launching*, which connects the ports of ITS components that are distributed throughout the target environment based on metadata in the deployment profile and executes the component assembly.

Problem → Ad hoc deployment mechanisms for variable ITS deployment requirements. In an ITS environment, each of the four deployment tasks described above can have variations due to differences in the given deployment profiles. For example, depending on the scale and amount of warehouse facilities, different ITS systems can have a wide range of nodes. Moreover, different types of nodes usually have different resources available, such as OS, network interfaces, CPU and memory. Large-scale warehouses usually have hundreds of such nodes that form a heterogeneous distributed environment. Conventional techniques for deploying large-scale component-based DRE systems can incur both inherent and accidental complexities. Common inherent complexities involve (1) ensuring component runtime libraries are compatible with the hosting nodes (e.g., OS compatibility) and (2) creating the correct number of processes to host components based on the specified deployment profile (e.g., some ITS components might be hosted in the same process to improve performance, whereas other ITS components might be hosted in different processes to improve system fault tolerance and reliability). Common accidental complexities stem from using *ad hoc* techniques for moving component runtime binaries and other dependent runtime libraries to the corresponding nodes for deployment. To perform these changes manually is not only tedious and error-prone, but also makes the deployment effort hard to reuse, e.g., there is no easy way to migrate one component running from one node/process to another when a deployment profile changes.

Solution → Standards-based deployment and configuration framework. To automate ITS component deployment and configuration, we developed a run-time framework called DAnCE [18], which 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.

As shown in Figure 5, DAnCE consists of implementations of a set of standards-based runtime interfaces that deal with the instantiation, installation, setting up connections, monitoring, and termination of components on the nodes of the target environment. Some interfaces (such as *ExecutionManager* and *DomainApplicationManager*) run at the global domain level, whereas others (such as *NodeManager* and *NodeApplicationManager*) run on each node. These interfaces 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 that may be distributed across multiple nodes.

When ITS deployers instruct a global interface to deploy an ITS assembly, they must give the XML-based deployment profile generated by the PICML MDD tool. The global interface then uses this profile as input 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, information about connections among component instances, and information about process collocation strategies and attribute configurations of components. Depending on the total number of nodes needed for a particular deployment, global deployment interfaces then split the global plan into multiple local (node-level) deployment plans and passes them to each node-level interface (service), based on the specification in the PICML model of the ITS.

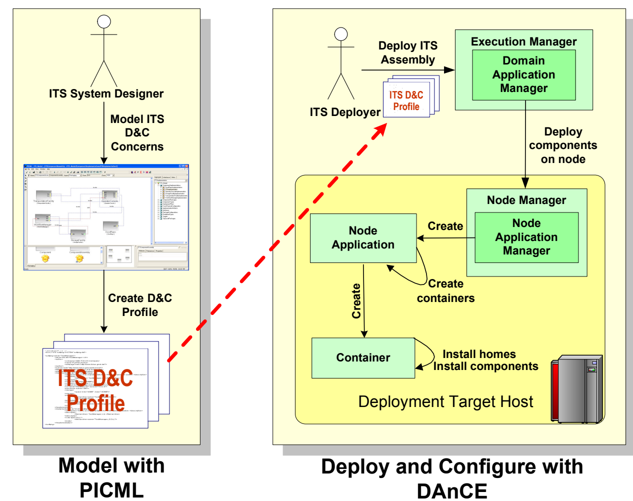

Figure 5. DAnCE Architecture and PICML Relationship

After each node-level deployment plan is dispatched, the node level interfaces will then parse the node-level deployment plan to prepare the components that will be deployed on the node that it manages, and fetch the runtime libraries from a centralized component repository if these libraries are not in the local file system. This deployment process will in turn spawn of one or more node application component servers, depending on the configuration strategies specified in the ITS PICML model. After these node applications are spawned, the specified components will then be installed into the containers and attributes configured in the PICML model will be honored by the container. After components are installed in each individual node, DAnCE creates connections among components, some of which may reside in hundreds of nodes in the warehouse. After the component deployment and application launch is complete, DAnCE also assists in monitoring component assemblies while they are executing and tearing down the application after it finishes executing.

Evaluating DAnCE for ITS. Based on the information captured by PICML, DAnCE maps ITS component software packages onto a running DRE system based on particular deployment profile. In our ITS case study shown in Table 1, without DAnCE tool support it would be prohibitively hard to (re)install all 193 compo-

nents on 26 nodes in our ITS case study manually, while taking into consideration the heterogeneous software/hardware platforms, and variable component configuration, and process collocation strategies. In contrast, by using DAnCE in conjunction with PICML, the whole ITS component deployment process is automated and simplified for 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 via an *OperatorConsole* terminal by DAnCE's *ExecutionManager* without manual intervention.

4. Evaluation

The goals of integrating MDD tools with QoS-enabled component middleware to develop a PLA for our ITS case study were to (1) construct software systems from 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 middleware platforms for PLAs. This section evaluates how well we achieved these goals.

4.1 Verifiability

When developing software products using PLAs, it is essential that product instances be verified and validated against certain domain rules pre-defined for the PLA before the product instance is released to customers or deployed onsite. 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 checkers 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 [20, 3, 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 product instances 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 non-functional system properties.

In our ITS case study, simple CORBA objects and more sophisticated CCM components must coexist, which introduces even more 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 for PLAs like ITS. Diversified operating policies (such as threading and buffering strategies) of ITS components can be customized based on a particular warehouse deployment scenario. Object-oriented middleware, such as CORBA and Java RMI, allows these configurations to be 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, [10] and (2) reduced composability due to hard coded application interfaces, such as those integrated with particular types of middleware services [19].

In contrast, component middleware enhance 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 run-time 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. This approach is particularly useful for DRE systems like ITS, which require customized QoS configurations for variable target OS, network, and hardware platforms that have different capabilities and properties.

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

[1] Includes both component server configuration, and middleware service configuration

[2] Includes component lifecycle management, such as activation, deactivation, remove

[3] Includes component executor IDL, executor implementation skeleton class generated by CIAO CIDL compiler, and XML descriptors generated by PICML MDD tool

[4] Percentage of the code that relates to the entire component lifecycle

Hand-Written   Auto-Generated

Figure 6. Weight of *WorkflowManager* Component in ITS

Figure 6 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. All other types of ITS components have similar weight distributions. As illustrated in Figure 6, 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.

When using 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 compo-

nent-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 Table 1, 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.

4.3 Open Issues

Our experiments with different ITS component deployment scenarios show the complementary relationships between CoSMIC MDD tools (i.e., WMGL/PICML) and underlying component middleware and D&C frameworks (i.e., CIAO/DAnCE). While we were generally successful in integrating and applying MDD and component middleware into our ITS case study, the following were open issues we felt warranted addition R&D to enable broader adoption of these integrated technologies:

1. 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 certain advanced scheduling analysis and verification tools are required to perform these capabilities 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.

2. How to reuse legacy code when transitioning from object-oriented to component-based architectures. Port 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 transition 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.

3. How to handle challenges associated with domain evolution. To effectively use MDD-based PLA technologies requires practical and scalable solutions to the *domain evolution problem* [21], which arises when existing PLAs are extended and/or refactored to handle unanticipated requirements or better satisfy existing requirements. For example, changing metamodels in a PLA 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 [16], exist they are hard to use and only provide partially automated solutions. Since these modifications significantly complicate PLA evolution efforts, they can outweigh the advantages of PLA 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.

These experiences 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.

5. Related Work

Our work on MDD extends earlier work on Model-Integrated Computing (MIC). Examples of MIC technology used today include GME [5] and Ptolemy [15] (used primarily in real-time and embedded domains) and Model Driven Architecture (MDA) based on UML and XML (used primarily in the enterprise business domains). Kennedy Carter's iUML Product Suite (www.kc.com) 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 UML 2.0 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, whereas our tools are based on DSMLs that unify the problem space and solution space, which is particularly useful when developing software PLAs. In contrast, Our work combines GME metamodeling mechanisms and UML to model and synthesize component middleware for configuring and deploying DRE systems.

Cadena [17] is an MDD tool 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 [22].

6. Concluding Remarks

This paper focuses on our experience gained when integrating MDD tools and QoS-enabled component middleware technologies and applying them to a PLA case study in the warehouse management domain. The benefits observed thus far include:

- The component middleware paradigm and implementations such as CIAO, 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 and Deployment and Configuration (D&C) specifications require a lot of configuration effort due to their large number of configuration points, hence we require advanced tools to help with this.
- 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.

Although our integrated approach addresses many hard problems, there is still much room for improvement and future work, as discussed in Section 4.3. CoSMIC, CIAO, and DAnCE are available in open-source format at www.dre.vanderbilt.edu.

References

[1] Object Management Group: "Lightweight CORBA Component Model Revised Submission", *Object Management Group, Inc.* May 2003, realtime/03-05-05

[2] N. Wang, D. Schmidt, A. Gokhale, C. Gill, C. Rodrigues, B. Natarajan, J. Loyall, and R. Schantz, "QoS-enabled Middleware," *Middleware for Communications*, Wiley and Sons, New York.

[3] D. Sharp and W. Roll, "Model-Based Integration of Reusable Component-Based Avionics System", Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS, Washington DC, May 2003.

[4] Object Management Group: "Deployment and Configuration for Component-based Distributed Applications," www.omg.org/docs/ptc/03-07-02.pdf, June 2003.

[5] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-Integrated Development of Embedded Software," *Proceedings of the IEEE,* January 2003.

[6], T. Ritter, M. Born, T. Untersch, T. Weis, "A QoS Metamodel and its Realization in a CORBA Component Infrastructure," Proceedings of the 36 Hawaii International Conference on System Sciences, Honolulu, HW, Jan. 2003.

[7] D. Schmidt, D. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, no. 4, Apr. 1998.

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

[9] Object Management Group: "Real-time CORBA", Adopted Specification of the Object Management Group, Inc. August 2002 Adopted Specification formal/02-08-02.

[10] G. Edwards, G. Deng, D. Schmidt, A. Gokhale, and B. Natarajan, "Model-driven Configuration and Deployment of Component Middleware Publisher/Subscriber Services", Proceedings of the 3rd ACM International Conference on Generative Programming and Component Engineering, Vancouver, CA, October 2004.

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

[12] J. Coplien, D. Hoffman, and D. Weiss, "Commonality and Variability in Software Engineering," IEEE Software, November/December, 1998.

[13] A. Gokhale, K. Balasubramanian., J. Balasubramanian, A. Krishna, G. Edwards, G. Deng, E. Turkay, J. Parsons, and D. 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 [in press].

[14] P. Clements and L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2002.

[15] J. T. Buck and S. Ha and E. A. Lee and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", *International Journal of Computer Simulation*, Special Issue on Simulation Software Development Component Development Strategies, Vol.4, April 1994.

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

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

[18] G. Deng, J. Balasubramanian, W. Otte, D. Schmidt, and A. Gokhale, DAnCE: A QoS-enabled Component Deployment and Conguration Engine, Proceedings of the 3rd Working Conference on Component Deployment, Grenoble, France, November 28-29, 2005.

[19] C. Szyperski, "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, Dec. 1997.

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

[21] R. Macala, L. Stuckey, D. Gross, "Managing Domain-Specific, Product line Development," *IEEE Software*, Vol.14, No. 13, May 1996.

[22] G. Trombetti, A. Gokhale, D. Schmidt, J. Hatcliff, G. Singh, and J. Greenwald, "An Integrated Model-driven Development Environment for Composing and Validating Distributed Real-time and Embedded Systems," *Model Driven Software Development- Volume II of Research and Practice in Software Engineering*, Springer-Verlag, 2005.