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

Gan Deng[1], Douglas C. Schmidt[1], Andrey Nechypurenko[2], Aniruddha Gokhale[1]

[1] Department of EECS, Vanderbilt University
Nashville, Tennessee, USA 37203
{dengg, schmidt, gokhale}@dre.vanderbilt.edu

[2] Siemens Corporate Technology, Munich, Germany
andrey.nechypurenko@Siemens.com

Abstract. Software for distributed real-time and embedded (DRE) systems must handle variabilities arising from (1) integration with various legacy subsystems using different technologies, languages, and platforms, (2) fine tuning needed to satisfy changing customer needs, and (3) appropriate packaging, configuration and deployment of functionality onto available system resources. Developers of applications and middleware must manage these variabilities without overcomplicating their solutions and exceeding project time and effort constraints. This paper presents our experience addressing domain- and middleware-specific variability gained when applying MDD tools and component middleware platforms to an inventory tracking system that manages the storage and flow of goods in warehouses. Our experience shows that integrating MDD tools and component middleware reduces DRE system development complexity, improves reuse and maintainability, and increases developer productivity.

Keywords: Model-Driven Development, Component Middleware.

1 Introduction

Emerging trends and challenges. During the past decade, *quality of service (QoS)-enabled component middleware* has emerged to help developers of distributed real-time and embedded (DRE) systems (1) factor out reusable concerns (such as component lifecycle management, authentication/authorization, and remoting) to enhance reuse and (2) avoid having to deal with low-level, tedious, error-prone, and non-portable platform details, such as socket and thread programming. Standards-based QoS-enabled 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/connections, and monitoring/enforcing the proper use of DRE system resources at runtime to meet end-to-end QoS requirements, such as throughput, latency, and jitter. QoS-enabled *component* middleware technologies, such as Lightweight CCM [1] and Prism [3], further simplify QoS provisioning via metadata and tools that help to (1) automate DRE system development lifecycle phases, such as packaging, assembly, configuration, and deployment, and (2) improve component reusability and performance by preventing premature commitment to specific QoS provisioning decisions, such as allocating components to thread pools and selecting the underlying

transport protocols. 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 many powerful capabilities, it also yields the following challenges for developers of DRE systems:

- **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 the subsystems and libraries. This myopia can cause them to implement suboptimal solutions that duplicate code unnecessarily, complicate system evolution, and violate key architectural principles (e.g., using implementation-specific functionality instead of public interfaces or prematurely committing to non-portable deployment policies).

- **Increased variability**. Functional variabilities include different business-logic implementations of the same interfaces, e.g., cranes, moving belts, and forklifts in our inventory tracking system case study, may vary in their ability to transport certain types of goods, depending on their weight, size, hazard-level, and other properties. Non-functional variabilities include the configuration of middleware services (such as naming, notification, security, or load balancing services), QoS-related configuration policies (such as concurrency and priority policies), and configuration of middleware internals itself (such as which middleware features should be enabled for a particular application in a particular environment).

The increased scale and variability of DRE systems requires developers to integrate different platforms and tools that solve essentially the same types of problems – yet are often non-portable and non-interoperable – without overcomplicating their solutions and exceeding project time and effort constraints. Likewise, developers of reusable middleware must also address these challenges when refactoring common capabilities from applications into effective reusable technologies and providing a portable operating environment for application developers. To maximize software reuse and productivity, therefore, increased scale and variability must be addressed by combining technologies that support alternative configurations and implementations of functionality more effectively than today's third-generation programming languages.

Solution 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 and validation challenges encountered by component middleware and DRE systems by combining (1) *metamodeling*, which defines type systems that precisely express key abstract syntax characteristics and static semantic constraints associated with particular application domains, such as software defined radios, avionics mission computing, and inventory tracking, (2) *domain-specific modeling languages* (DSMLs), which provide programming notations that are guided by and extend metamodels to formalize the process of specifying application logic and QoS-related requirements in a domain, and (3) *model transformations and code generation* that automate and ensure the consistency of software implementations with analysis information associated with functional and QoS requirements captured by structural and behavioral models.

We have created an MDD toolsuite called the *Component Synthesis using Model Integrated Computing* (CoSMIC) [13, 20], which is an integrated collection of DSMLs that support the development, deployment, configuration, and evaluation of QoS-enabled component middleware-based DRE systems. We also created a DSML called the *Warehouse Modeling and Generation Language* (WMGL) that models the physical layout of the warehouse and then generates code to populate database tables that contain the information of warehouse mechanical facilities, storage facilities, and their interconnecting relationships. In addition, we created a QoS-enabled component middleware platform called *Component-Integrated ACE ORB* (CIAO) that combines Lightweight CCM [1] capabilities with Real-time CORBA [9] features, such as thread pools and client-propagated and server-declared priority policies.

To evaluate how the *integration* of MDD tools and QoS-enabled component middleware helps resolve the challenges presented above, we have created an *inventory tracking system* (ITS), which provides logistics support to manage the flow of goods and assets in and across warehouses. 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). This paper presents our experience gained while integrating MDD and QoS-enabled component middleware to address two key concerns of ITS: warehouse configuration and component assembly, configuration, and deployment. The goal of our integration efforts were to help (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 warehouse requirements, and configuring middleware services via DSMLs, and (3) automate key steps in the software lifecycle, such as generating deployment and configuration XML metadata and/or source code from DSMLs, and automating the deployment of components and services on a target running environment 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, focusing on the scale and variability of its requirements, component architecture, and component middleware infrastructure; Section 3 describes how we integrated and applied MDD tools and QoS-enabled middleware to resolve key technical problems of our ITS case study; Section 4 compares our work with related efforts; and Section 5 presents concluding remarks.

2 Overview of the ITS Case Study

An inventory tracking system (ITS) provides logistics support to manage the flow of goods in and between warehouses, such as those shown in Figure 1. A key goal of an ITS is to provide reliable, efficient, and convenient mechanisms that manage the warehouse and the movement of inventory in a timely and reliable manner. For instance, an ITS should enable human operators to configure warehouse storage organization criteria and warehouse transportation facility criteria, maintain the set of goods known throughout a DRE system (which may span organizational and even international boundaries), and track warehouse assets using GUI-based operator monitoring consoles.

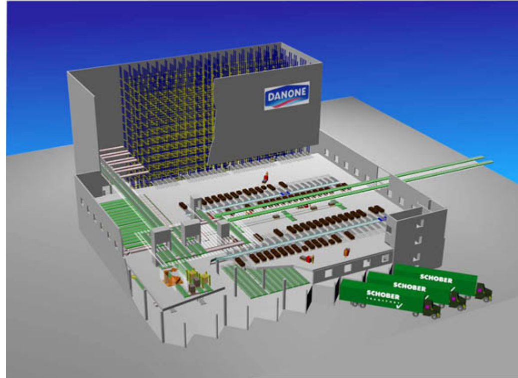

Figure 1: ITS Environment

2.1 ITS Actors and Use Cases

Figure 2 shows the primary actors and use cases in our ITS, which perform the following activities:

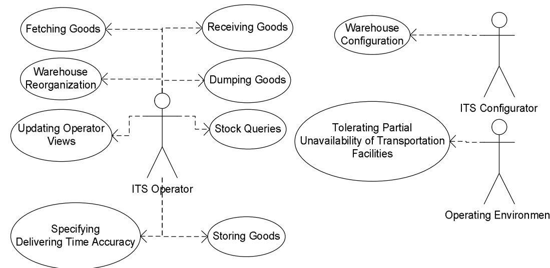

Figure 2: Actors in Our ITS Case Study

- *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 goods, and characteristics of storage facilities (e.g., whether hazardous goods are allowed to be stored, maximum allowed total weight of stored goods, etc.).
- *Operator* actors use ITS capabilities to reorganize warehouses to fit future changes, as well as dealing with other use cases, such as receiving goods, storing goods into the warehouse, fetching goods from the warehouse and delivering to particular location, dumping goods, goods inventory queries, specifying delivery time accuracy, 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 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 in real-time based on available transportation resources and delivery time requirements.

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.

2.2 ITS Component Architecture and Key Variabilities

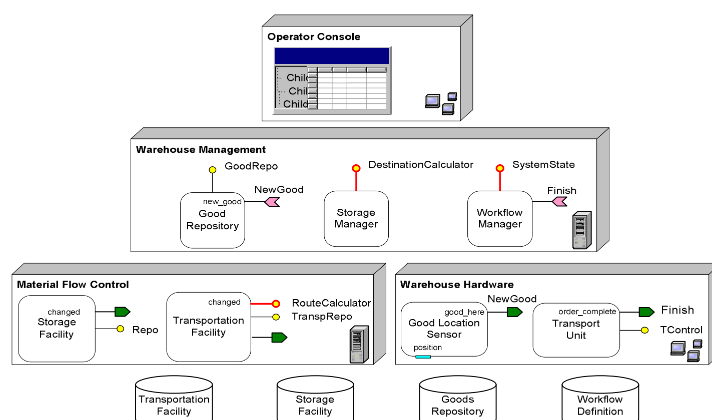

Figure 3: Key ITS Architecture Components.

Figure 3 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 *GoodsRepository* and *StorageFacility*) expose interfaces to manage databases (such as the goods inventory and storage facilities). Finally, the event flow within the ITS is controlled and coordinated by components (such as the *WorkflowManager* and *StorageManager*). These various capabilities are illustrated in Figure 3 and described below in the context of their associated ITS subsystems:

- 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 goods to the destination location. This subsystem handles all related 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 *GoodLocationSensor* and *TransportUnit*.

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

Implementing a large-scale ITS requires commonality and variability analysis [12], e.g., all transportation facilities are represented with the same component interface, *i.e.*, *TransportUnit*. Implementations can vary, however, due to differences in hardware facilities for transporting certain types of goods, 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 deploy 100's–1,000's of components. Section 3 evaluates key variabilities in detail and shows how the integration of MDD tools and component middleware help address them.

2.3 ITS Component Technologies

The ITS component architecture is developed in accordance with the OMG's CORBA Component Model (CCM) [1]. The CCM implementation used for our ITS project is the *Component-Integrated ACE ORB* (CIAO) [2], which is QoS-enabled component middleware built atop *The ACE ORB* (TAO) [7]. TAO is a highly configurable, open-source 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 priority models, thread-to-connection bindings, and timing properties) into elements that can be configured declaratively via metadata. Promoting these QoS concerns as metadata disentangles code for controlling these non-functional concerns from code that implements the application logic, thus making DRE system development more flexible and productive. To integrate component middleware with MDD tools, we developed a QoS-enabled *deployment and configuration engine* (DAnCE) within CIAO that allows application deployers to specify how existing components should be configured, deployed, and customized into reusable services. Section 3.2.2 describes how DAnCE combines MDD tools and component middleware to simplify the development of our ITS case study.

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: ITS Case Study Characteristics

The ITS case study we developed using CIAO, DAnCE, and MDD tools contains ~500 storage facilities and ~200 ITS components deployed in the target environment. As shown in Table 1, there are 2 *OperatorConsoles*, 1 *TransportationFacility*, 1 *GoodRepository*, 1 *StorageManager*, 1 *WorkflowManager*, 1 *StorageFacility*, 18 *GoodLocationSensors* and 168 *TransportUnit*s. The 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 collocation cases: *GoodRepository*/*StorageManager* and *TransportationFacility*/*StorageFacility*. The composition and configuration of other ITS deployments may vary significantly, depending on warehouse facilities, computing hardware, and software resources available in a warehouse.

3. Developing the ITS by Integrating MDD Tools and Component Middleware

This section describes our experience gained when integrating MDD tools and component middleware to address scalability and variability issues by enabling them to work at higher levels of abstraction than components and classes written in third-generation languages and distributed object computing platforms. We developed, integrated, and applied MDD tools and QoS-enabled component middleware to our ITS case study to help simplify and automate the following concerns:

- **Modeling and synthesizing warehouse configurations**, which involve simplifying and automating the configuration of warehouse artifacts and population of ITS databases available in various types of warehouses. Section 3.1 describes the *Warehouse Modeling and Generation Language* (WMGL) MDD tool we developed to represent warehouse structures and behaviors as higher-level models.
- **Modeling and synthesizing component software deployment and configuration concerns**, which involve simplifying and automating the middleware and applications that implement ITS functionality. Section 3.2 describes how the CoSMIC MDD tools and DAnCE were integrated with the CIAO CCM implementation to develop, assemble, and deploy various types of ITS software components.

The remainder of this section describes key problems we faced when addressing these concerns, presents our solutions, and evaluates these solutions in the context of the ITS case study described in Section 2.

3.1 Addressing ITS Warehouse Configuration Concerns

A key challenge in designing an ITS is to provide a generic, reconfigurable DRE system that can be deployed rapidly in different warehouse configurations or redeployed to adapt to reconfigurations of an existing warehouse, taking into account the transportation and storage facilities, as well as the physical warehouse topology. The proper configuration of an ITS depends heavily on the physical layout and transportation facilities of a warehouse, which may vary in different circumstances. For example, it is common to add new transportation facilities into an existing warehouse after it is deployed.

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 goods).

Problem → Ad hoc, tightly coupled warehouse design. ITS developers have historically relied on *ad hoc* approaches (such as manually writing programs from scratch) to (1) create software components that correspond to transportation facility units and (2) store physical warehouse layout configurations into databases. Moreover, they often hard code this information using third-generation programming languages, which overly couples their solutions to particular warehouse configurations and technologies. Such tight couplings make it hard to deploy an ITS in another warehouse with different configurations and to evolve after the initial deployment since changes in the warehouse configuration require modifications, recompilation, and redeployment of 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 MDD tool that represents warehouse structures and behaviors as higher-level models, which allows developers to visually depict and manipulate (1) 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) and (2) the *available storage facilities*, which include their physical position information and properties (e.g., storage capacity and type of goods 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 a shortest path algorithm on this graph to determine the optimal transportation path to transfer a particular good 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.

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 4 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.

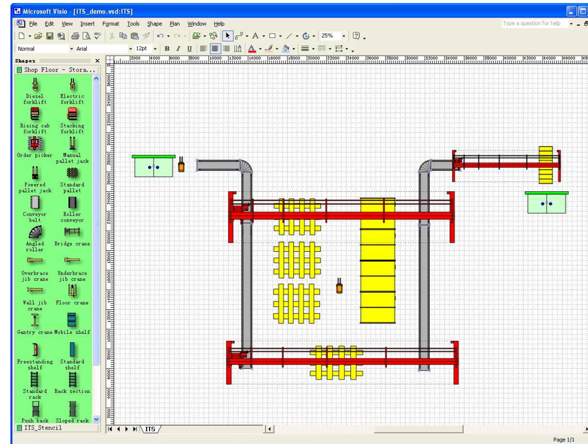

Figure 4. A Warehouse Configuration in WMGL

A key benefit of WMGL is its ability to automate the correct-by-construction transition from WMGL model 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 scheduling information for warehouse maintenance) are generated automatically via the WMGL model interpreter.

To validate the correctness of a data model, the WMGL model interpreter applies analysis and validation 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. Additionally, 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.

Once validated, WMGL can generate C++ or Java code, which is used 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 CIAO components can 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 run-

ning the WMGL model interpreter, the ITS can begin the component-based deployment and configuration process described in Section 3.2.

Evaluating WMGL for ITS. WMGL provides several benefits for our ITS case study. For example, it visually captures warehouse information (such as positions, sizes, and reachable areas), which helps reduce complexity. Moreover, the model analysis in the WMGL interpreter detects warehouse design faults (such as isolated storage facilities) during design rather than runtime. For our ITS case study in Table 1, the WMGL model interpreter automatically generates ~7,000 lines of C++ code to describe the warehouse layout and artifact property information, which could be readily used by CIAO components.

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. These mapping rules 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 1 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 a DRE system composed of CCM components developed using CIAO, which is a QoS-enabled component middleware implementation that supports the OMG Deployment and Configuration (D&C) specification [4]. In this specification, deployment is the sequence of activities 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 resolved key component deployment and configuration challenges that arose when developing the ITS.

3.2.1 Automating ITS Deployment and Configuration Profile Generation

Installing an ITS 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 large-scale DRE systems, 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 both QT runtime library and the *WorkflowManager* component implementation library), (2) specifying the connectivity between ports of ITS components, and (3) mapping ITS components to the appropriate nodes in the target environment where the ITS will be deployed.

Problem → Ad hoc deployment and configuration profile creation of components for diverse system requirements. Large-scale DRE systems, such as an ITS, may re-

quire creation of assemblies containing 100's-1000's 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, e.g., handcrafting XML files describing component metadata (such as 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 CoSMIC tool-suite. At the heart of CoSMIC is the *Platform-Independent Component Modeling Language* (PICML) [13], which is a DSML that implements the OMG D&C specification and 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 is developed using the *Generic Modeling Environment* (GME) [5], which is an 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 connections among the ITS components, such as the facet/receptacle 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. Its model interpreter ensure the dynamic semantics of models built using PICML, which can range from performing analysis of models to synthesizing code for components and their metadata.

PICML contains multiple model interpreters, each performing a particular function. The most commonly used interpreter for our ITS 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.

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 5, 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 CIAO to drive the deployment of the complete ITS applications.

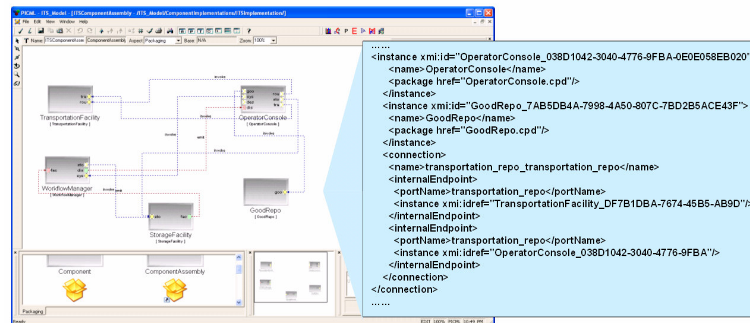

Figure 5. Partial PICML Assembly Model for ITS

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. 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, i.e., component interface descriptors and component implementation descriptors. To create a deployment profile for this case study 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 is much easier to create a PICML model for these connections visually than writing XML files manually. The PICML packaging interpret generates all six types of descriptor files described above, with a total number of 582 XML files averaging ~25 lines per file. By automatically generating the deployment via PICML's model interpreter, it enforces the correct-by-construction paradigm in component-based application development, which eliminates a common source of errors.

Our experience applying PICML to model the ITS deployment structure also shows that it raises the level of abstraction at which developers work and helps them 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 eliminates many sources of accidental complexities and improves overall system quality.

3.2.2 Automate ITS Component Deployment to Target Environment

To complete the deployment of ITS application, it is necessary to take the metadata describing the concerns from multiple actors and bring them together in an effective fashion into the target environment. Section 3.2.1 explains how the PICML MDD tool addresses key concerns in the ITS component configuration and assembly phase by automatically and correctly synthesizing various types of XML descriptors. These XML descriptors then form a *profile* that specifies system deployment requirements.

To deploy an ITS assembly, deployers must perform certain 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 the ITS components that are distributed throughout the target environment based on metadata in the deployment profile and executes the whole component assembly.

Problem → Ad hoc deployment mechanisms for variable ITS deployment requirements. In an ITS environment, each of the four deployment steps described above can have variations due to the differences in the given deployment profiles. For example, depending on the scale and amount of warehouse facilities, different ITS systems often have different number 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 support automatic ITS component deployment and configuration capability, we developed a run-time framework called the *Deployment And Configuration Engine* (DAnCE), whose structure is shown in the right part of the Figure 6. As shown in this figure, ITS system developers can model various D&C concerns for a warehouse via PICML, which automatically generates the corresponding D&C profile for the designated system. DAnCE then takes the generated profile, and automatically deploys the system into the CIAO component middleware platform.

As shown in Figure 6, 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. These interfaces include the *ExecutionManager*, *DomainApplicationManager*, *NodeManager* and *NodeApplicationManager* defined by the OMG D&C specification. The *ExecutionManager* and *DomainApplicationManager* run at the global domain level, whereas the *NodeManager* and *NodeApplicationManager* run on each node. The *ExecutionManager* and *NodeManager* manage the lifecycle of the ITS deployment process to help configure component servers on the nodes, install components into containers, and set up connections among components that may be distributed across multiple nodes.

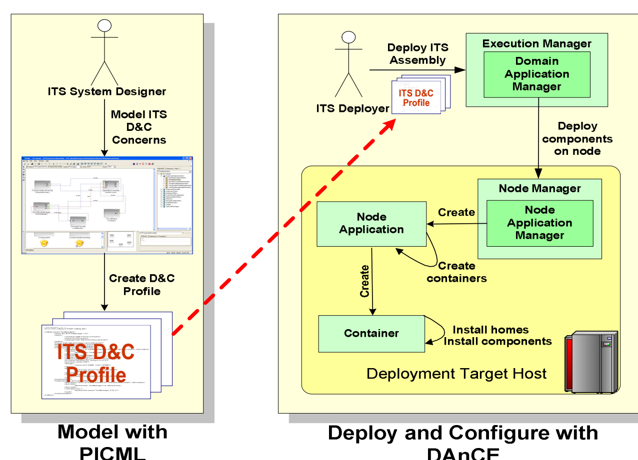

Figure 6. DAnCE Architecture and PICML Relationship

When ITS deployers instruct an *ExecutionManager* to deploy an ITS assembly, they must give the *ExecutionManager* the XML-based deployment profile generated by the PICML MDD tool. The *ExecutionManager* takes this profile as input and creates an in-memory representation of the metadata by parsing the XML files. The DAnCE *DomainApplicationManager* then populates a global in-memory deployment plan that describes a mapping of a configured ITS assembly into a target domain, which includes the information about nodes where components will be deployed, the mapping of component to nodes, the information about connections among component instances, the information about process collocation strategies and attribute configurations of components. Depending on the total number of nodes needed for a particular deployment, the *DomainApplicationManager* then splits the global plan into multiple local (node-level) deployment plans and passes them to each individual *NodeManager*, based on the specification in the PICML model of the ITS.

After each node-level deployment plan is sent to each corresponding *NodeManager*, the *NodeManager* will then parse the node-level deployment plan to find out the components that will be deployed on the node that it manages, and fetch the runtime libraries from the centralized component repository if these libraries are not in the local file system. This will in turn trigger the *NodeApplicationManager* residing in the node to spawn one or more *NodeApplications*, i.e., component servers, depending on the collo-

cation strategy specified in the ITS PICML model. After *NodeApplications* are spawned, the specified components will then be installed into the container of the *NodeApplications* and attributes specified in the PICML model will be configured by the container.

After components are installed in each individual node, DAnCE will create 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 and tearing down the application after it finishes executing.

Evaluating DAnCE for ITS. Based on the information captured by PICML, DAnCE maps the ITS software packages onto a running DRE system based on particular deployment profile. Without DAnCE tool support it is prohibitively hard to (re)install all 193 components on 26 nodes in our ITS case study manually, while taking into consideration of the heterogeneous software/hardware platforms, and variable component configuration, and process collocation strategies.

In contrast, by using DAnCE in conjunction with PICML, the whole 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 the *ExecutionManager* without manual intervention.

Our experiments with different deployment scenarios and development of the corresponding MDD tools show the complementary relationships between modeling tools (i.e., WMGL/CoSMIC) and underlying component infrastructure and D&C frameworks (i.e. CIAO/DAnCE). In particular, it is important to base the code generation process on the underlying middleware, thereby reducing the complexity of model interpreters since they only need to understand the middleware APIs.

4. Related Work

Our work extends earlier work on Model-Integrated Computing (MIC) [14] that focused on modeling and synthesizing embedded software. Examples of MIC technology used today include GME [5] and Ptolemy [15] (used primarily in the real-time and embedded domain) and Model Driven Architecture (MDA) based on UML and XML (which have been used primarily in the business domain). Our work combines GME metamodeling mechanisms and UML to model and synthesize component middleware used to configure and deploy DRE systems.

Cadena [17] is an MDD tool for building and modeling component-based DRE systems, with the goal of applying static analysis, model-checking, and lightweight formal methods to enhance these systems. Unlike our work, 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, such aspects could be captured through PICML MDD tool and then all the deployment and configuration work could be automated through DAnCE.

Lacour et. al. [16] use the Globus Toolkit to deploy CCM components on a computational grid. Unlike our work, this work provides neither a higher-level modeling tool for application developers to capture various concern aspects, such as deployment planning

and collocation strategy modeling, nor does it provide a way to automatically generate a deployment profile for the application. Also, unlike our approach, their tools and middleware platforms are targeted for enterprise distributed application instead of DRE systems, which have more stringent QoS requirements.

5. Concluding Remarks

Our prior work on MDD focused on the architectural design and implementation of CoSMIC and its DSMLs and our prior work on QoS-enabled component middleware focused on the design and optimization of CIAO. This paper focuses on our experience gained when integrating and applying these technologies to an inventory tracking system (ITS) case study in the warehouse management domain. The lessons we learned 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 Lightweight CCM and Deployment and Configuration (D&C) specifications require many configuration files due to their large number of configuration points.
- 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 only missing or 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 starting from the raw component middleware without MDD tool support, since the application could evolve from the existing set of PICML and WMGL models.
- Domain-specific modeling techniques can help to reduce the learning curve for end users. For example, warehouse modelers in our ITS project need little or no knowledge of how to write component software since they interact with the system through higher-level models that correspond to the "language" understood by domain engineers and visual modeling environments, such as WMGL.
- 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 relatively small amount of components we have in ITS. This observation motivates the need for further research in the automating the synthesis of large-scale DRE systems based on the different types of meta- and semantic information about assembly units, such as components or services.

CoSMIC's MDD tools are available at www.dre.vanderbilt.edu/cosmic and the CIAO QoS-enabled component middleware is available at www.dre.vanderbilt.edu/CIAO. GME is available at www.isis.vanderbilt.edu/Projects/gme.

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] A. Ledeczi "The Generic Modeling Environment", Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001.

[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, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, 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] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. Schmidt, "A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems," Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium, San Francisco, CA, Mar, 2005.

[14] J. Sztipanovits and G. Karsai, "Model-Integrated Computing," *IEEE Computer*, vol. 30, pp. 110–112, Apr. 1997.

[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] S. Lacour, C. Perez, T. Priol, "Deploying CORBA Components on a Computational Grid: General Principles and Early Experiments Using the Globus Toolkit", Proceedings of the 2nd International Working Conference on Component Deployment (CD 2004), Edinburgh, UK, May 2004.

[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.