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

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

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

Abstract

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

Keywords: Model-Driven Development, Domain-Specific Modeling Languages, Component Middleware

1. Introduction

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

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

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

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

2. Increased variability. Additions to the features of the system and/or availability of better implementations of the same type of systems can further increase functional and QoS variability. For example, new types of warehouse transportation facilities, such as forklifts or cranes, with sophisticated features and timeliness requirements may be introduced in a warehouse, hence requiring appropriate changes in the ITS software. It is hard to accommodate these changes within individual components without complicating the solution and affecting the overall QoS of the entire system. To maximize software reuse and productivity, therefore, increased scale and variability must be addressed by more effectively combining technologies and tools that support system configuration and integration.

Promising approach → Integrating model-driven development and QoS-enabled component middleware. A promising way to alleviate the challenges of DRE system scale and variability described above is to integrate model-driven development (MDD) [6, 7, 8, 9] techniques with QoS-enabled component middleware. MDD helps resolve key software development challenges by combining (1) *domain-specific modeling languages* (DSMLs), whose type systems formalize the application structure, behavior, and requirements within a particular domain, such as software defined radios, avionics mission computing, and warehouse inventory tracking. DSMLs are described using *metamodels* that define the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these concepts and (2) *model transformations and code generation* that help automate repetitive, tedious, and error-prone tasks in the software lifecycle to ensure the consistency of software implementations with analysis information associated with functional and QoS requirements captured by structural and behavioral models.

In prior work, we developed the *Component-Integrated ACE ORB* (CIAO) [4], which is a QoS-enabled component middleware platform that combines Lightweight CORBA Component Model (CCM) [3] features (such as standard mechanisms for specifying, implementing, packaging, assembling, and deploying component instances) with Real-time CORBA features [1] (such as thread pools, portable priorities, synchronizers, priority preservation policies, and explicit binding mechanisms). A key part of CIAO is the *Deployment And Configuration Engine* (DAnCE) [10], which implements the OMG Deployment and Configuration (D&C) specification [11] to help developers deploy and configure pre-built components and component assemblies. We also developed an MDD toolsuite called *Component Synthesis using Model Integrated Computing* (CoSMIC) (`www.dre.vanderbilt.edu/cosmic`), which is an integrated collection of DSMLs that support the specification, analysis, development, configuration, deployment, and evaluation of component-based DRE systems. To evaluate how well the integration of QoS-enabled component middleware and MDD tools helps resolve the scale and variability challenges described above, we developed an ITS case study that provides logistics support to manage the flow of items and assets in and across warehouses in a distributed and timely manner.

This paper presents our experience gained and lessons learned while integrating MDD and QoS-enabled component middleware to address two key variability concerns in designing the ITS: (1) warehouse configuration and management concerns and (2) component management, assembly, QoS configuration, and deployment concerns.

2. Overview of the ITS Case Study

2.1. ITS Component Architecture

Figure 1 illustrates the components that form the core implementation and integration units of our ITS case study. Some ITS components (such as the `OperatorConsole`) expose interfaces to end users, *i.e.*, ITS operators. Other components (such as `TransportationUnit`) represent hardware entities, such as cranes, forklifts, and belts. 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`).

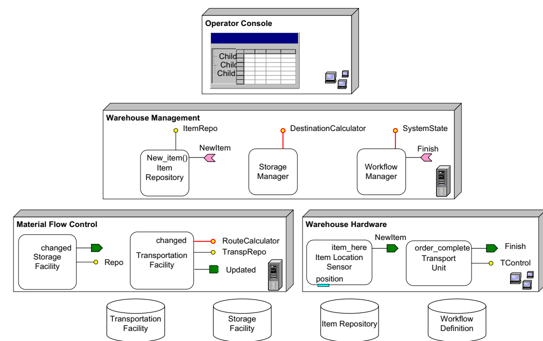

Figure 1. ITS Subsystems and Key Components

The capabilities shown in Figure 1 are used in the context of their associated ITS subsystems as follows: (1) the *Warehouse Management subsystem* consists of a set of high-level functionality and decision making components that calculate the destination location, (2) The *Material Flow Control subsystem* executes high-level decisions calculated by the

Warehouse Management subsystem to deliver items to the destination location, *e.g.*, it (re)calculates routes, schedules transportation facilities, and reserves storage facilities, (3) the *Warehouse Hardware subsystem* handles physical devices, such as sensors and transportation units (*e.g.*, belts, forklifts, cranes, and pallet jacks) that correspond to various component types, such as `ItemLocationSensor` and `TransportUnit`.

The functionality of the ITS subsystems shown in Figure 1 can be monitored and controlled by one or more `OperatorConsole` components, and all persistence concerns are handled via databases. The Material Flow Control subsystem requires high throughput for continuously refreshed data and soft real-time processing for regular tasks such as monitoring warehouse good delivery activities. For example, Material Flow Control subsystem need to continuously refresh and collect the geographical location data of good items as well as certain transportation units. Components in the Warehouse Hardware subsystem require hard real-time deadlines for certain processing tasks, such as hardware process control. For example, an automatic crane system is a hard real-time system because a delayed signal may cause hardware failure or damage. To make development more rapid and flexible, our ITS architecture is implemented using the CIAO/DAnCE QoS-enabled component middleware and the CoSMIC MDD tools described in Section 1 to separate the application business logic from system deployment and configuration concerns.

2.2. Scale and Variability in the ITS Case Study

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

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

3. Resolving ITS Challenges by Integrating MDD Tools and Component Middleware

This section describes how we applied the CoSMIC MDD tools and CIAO/DAnCE QoS-enabled component middleware to help simplify and automate the ITS development challenges described in Section 2. We describe the key problems faced when addressing these challenges, present our solutions, and evaluate these solutions in the context of the ITS case study.

3.1. Addressing ITS Warehouse Configuration Concerns

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

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

Solution: A DSML for warehouse configuration. To address the problems described above, we developed the *Warehouse Modeling and Generation Language* (WMGL). WMGL is a DSML in CoSMIC that represents warehouse structures and behaviors as visual models that allow developers to

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). WGML can also be used to depict and manipulate the available storage facilities, which include their physical position information and properties (*e.g.*, storage capacity and type of items they can store).

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

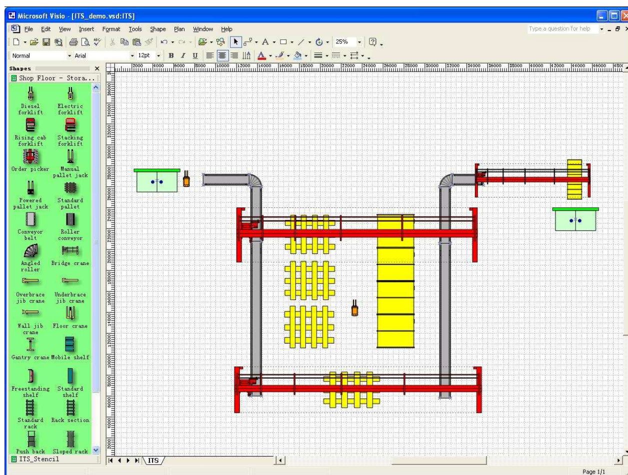

Figure 2. A Warehouse Configuration in WMGL

We built WMGL using Microsoft Visio since it supports a wide range of sophisticated graphics capabilities, an embeddable programming environment that enables developers to build custom tools, and integration with popular database management systems, such as Oracle and MySQL. Figure 2 illustrates a Visio screenshot of a partial ITS WMGL model, where warehouse model elements are available from the left-side master panel and the right-side panel contains a drawing that represents a warehouse configuration consisting of two moving angle belts, three cranes, four storage racks, two fork lifts and two gates. Modeling a warehouse in WMGL involves drawing the concrete warehouse physical structure and then adding customized properties (such as capacity, size, etc) to transportation and storage facilities model elements. Warehouse modelers can also specify the reachable range of particular transportation units (*e.g.*, forklifts and cranes) visually and define various properties (*e.g.*, capacity, heating or cooling) of storage locations.

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

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 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.

3.2. Addressing ITS Component Deployment and Configuration Concerns

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

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

operational requirements and (2) interactions between the components meet ITS QoS requirements.

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

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

Solution: MDD-based deployment and configuration of ITS components. In our ITS project, system deployment and configuration is performed via the *Platform-Independent Component Modeling Language* (PICML) [12], which is a DSML in the CoSMIC toolsuite that works together with the DAnCE middleware to implement the OMG D&C specification [11]. PICML provides capabilities to handle complex component engineering tasks, such as multi-aspect visualization and manipulation of components and the interactions of their subsystems, component deployment planning, and hierarchical modeling and generation of component assemblies. PICML itself uses the *Generic Modeling Environment* (GME) [13], which is a metaprogrammable development environment for building and processing visual DSMLs.

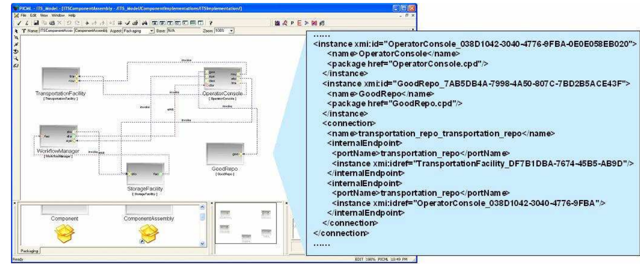

Figure 3. Partial PICML Assembly Model for ITS

PICML's metamodel and model interpreter enforce constraints defined by semantic rules associated with component assemblies. Its metamodel defines static semantic rules that determine valid connections between components. PICML's model interpreters ensure the dynamic semantics of models specified by users, *e.g.*, they can analyze models for various well-formedness properties and synthesize code for components and their XML descriptors that convey metadata needed by DAnCE.

In the context of ITS, a major cause of missed deadlines is *priority inversions*, where lower priority requests access a resource at the expense of higher priority requests. Priority inversions must be prevented or bounded since they can cause some critical paths in the ITS system to miss their deadlines. To reduce priority inversions, we use PICML to configure Real-time CORBA policies [14] of the ITS component instances, which include (1) processor resources via priority mechanisms, thread pools, and synchronizers, for real-time components with fixed priorities, (2) communication resources via protocol properties and explicit bindings to server objects using priority bands and private connections, and (3) memory resources via bounding the size of request buffers and thread pools.

As shown in Figure 3, the PICML-generated 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. This metadata is used by DAnCE to drive the deployment of the complete ITS applications, as explained in Section 3.2.2.

3.2.2. Automating ITS Deployment Process to Ensure QoS Requirements

Deployment is the sequence of activities that occurs between (1) the acquisition of software and its associated configuration and deployment metadata and (2) the actual execution of software in a target environment based on the acquired software and associated metadata. Likewise, configuration is the process of mapping known variations in the application requirements space to known variations in the middleware and application software so-

lution space. To complete the deployment and configuration of an ITS application, DAnCE uses the metadata generated by PICML (Section 3.2.1) to describe the concerns from multiple actors and combines them in the target environment to enforce overall system QoS requirements.

To deploy an ITS assembly, ITS developers must perform four tasks based on the deployment profile, including (1) *preparation*, which places ITS software packages into component software repositories [15], (2) *installation*, which downloads ITS components to component server processes that run in each node in the target environment, including embedded system nodes used to host TransportUnit components and PC nodes that host other types of ITS components, such as `WorkflowManager` and `OperatorConsole`, (3) *configuration*, which customizes QoS properties of components and containers on each node, and (4) *launching*, which connects the ports of ITS components that are distributed throughout the target environment and initiates system execution.

Problem: Ad hoc *deployment mechanisms for variable ITS deployment requirements.* In large-scale DRE systems, the deployment process must consider QoS requirements (such as low latency and bounded jitter) throughout the lifecycle. Components, containers, and component servers must therefore be deployed in accordance with real-time QoS properties by (1) specifying middleware configurations parameters (such as client request optimization options) and (2) setting the QoS policy options provided by the underlying middleware into semantically consistent configurations. For instance, whenever a `ConveyorBelt` component's hardware fails, the ITS should notify the `WorkflowManager` in real-time to minimize/avoid damage. Likewise, ITS `ConveyorBelt` and `Crane` components may need to be collocated with `WorkflowManager` in some assemblies to minimize latency.

The existing OMG D&C specification does not support real-time QoS policies. Moreover, QoS-enabled object-oriented middleware, such as Real-time CORBA and Real-time Java, do not adequately separate real-time policy configurations from application functionality, which yields tightly coupled deployment mechanisms, which makes deployment artifacts and effort hard to reuse, *e.g.*, there is no easy way to reconfigure a warehouse to accommodate the variability. It is therefore hard for ITS developers to configure, validate, modify, and evolve their systems consistently using QoS-enabled object-oriented middleware.

Solution: Extending the OMG D&C specification to support real-time QoS policies. DAnCE extends the OMG D&C specification to enable the configuration of real-time QoS policies through metadata generated by PICML. The architecture of DAnCE is shown in the right side of Figure 4. This figure also shows how ITS developers can model various warehouse D&C concerns 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 CIAO, thereby bridging the gap between higher-level MDD tools and lower-level component middleware runtime platforms.

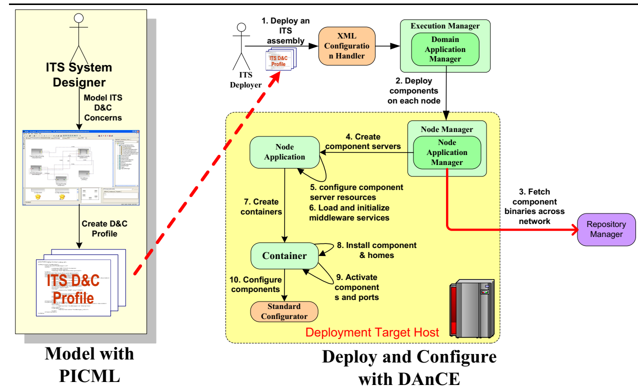

Figure 4. DAnCE Architecture and PICML Relationship

As shown in Figure 4, DAnCE provides runtime services that handle the instantiation, installation, configuration, monitoring, and termination of ITS components on the nodes of the target environment. Some services (such as `ExecutionManager` and `DomainApplicationManager`) run at the end-to-end domain level and process global deployment plans, whereas others (such as `NodeManager` and `NodeApplicationManager`) deploy and configure ITS components on each node. These services together manage the lifecycle of the ITS deployment process to configure component servers on the individual nodes, install components into containers, and establish connections among components.

To enforce QoS requirements, DAnCE extends the OMG D&C specification to define`NodeApplication` server resource configurations, which influence end-to-end QoS behavior. Figure 5 shows the different categories of server configurations that can be specified using the DAnCE *server resources XML schema*, which are related to system end-to-end QoS enforcement. Each server resources specification can set the following options: (1) *ORB command-line options*, which control TAO's connection management models, protocol selection, and optimized request processing, and (2) *ORB service configuration option*, which specify ORB resource factories that control server concurrency and demultiplexing models. Using this XML schema, ITS system deployers can specify their desired ORB configurations.

ITS components are hosted in containers created by

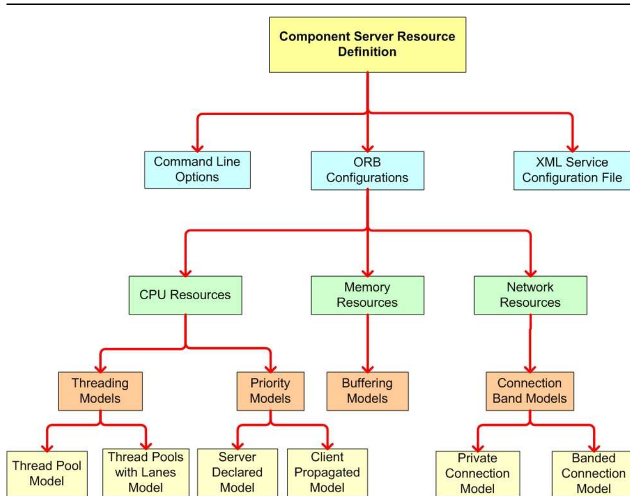

Figure 5. Specifying Real-time QoS Requirements

the `NodeApplication` process, which provides the runtime environment and resources for components to execute and communicate with other components in a component assembly. The ORB configurations defined by the *server resources XML schema* are used to configure `NodeApplication` processes that host components, thereby providing the necessary resources for the components to operate. For example, since some ITS components, such as `ItemLocationSensor` and `WorkflowManager`, handle real-time item delivery activities they can be configured to deliver more stringent QoS requirements than other ITS components, such as low end-to-end latency delay and bounded jitter.

4. Related Work

Our work on MDD extends earlier work on Model-Integrated Computing (MIC) (such as GME [13] and Ptolemy [16]) used primarily in real-time and embedded domains and Model Driven Architecture (MDA) [17] based on UML and XML used primarily in the enterprise business domain. Kennedy Carter's iUML Product Suite [18] supports the Executable UML process from textual requirements management through modeling to complete target code generation. The Rhapsody System Designer tool by I-Logix (www.ilogix.com) is based on OMG's MDA and UML 2.0 specifications and generates application code from UML models using multiple programming languages (such as C/C++, Java, or Ada) and multiple middleware platforms (such as CORBA and EJB). The tools, however, are based on OMG's UML specification, which is a *domain-independent* modeling language. In contrast, our work on CoSMIC focuses on *domain-specific* modeling languages (DSMLs) that better unify the problem space and solution space by capturing designer intent more effectively.

The *Embedded Systems Modeling Language* (ESML) [19] provides a visual metamodeling language that captures multiple views of embedded systems, allowing a diagrammatic specification of complex models. The modeling building blocks include software components, component interactions, hardware configurations, and scheduling policies. The user-created models can be fed to analysis tools (such as AIRES, VEST, and Cadena) to perform schedulability and event analysis. Unlike our work on CoSMIC and PICML, however, ESML is platform-specific since it is customized for the Boeing Bold Stroke PRiSm QoS-enabled component model. Moreover, ESML does not support nested assemblies and the allocation of components are tied to proprietary features of the Bold Stroke component model.

Cadena [20] is a MDD tool developed with Eclipse to build 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. We are collaborating with the Cadena team to create an integrated suite of MDD tools [21].

5. Concluding Remarks

This paper describes our experiences integrating MDD tools and QoS-enabled component middleware technologies and applying them to an inventory tracking system (ITS) case study in the warehouse management domain. Some lessons learned from our work thus far include:

• *MDD tools alleviate complexities associated with component middleware.* Although component middleware elevates the abstraction level of middleware to enhance software developer quality and productivity, it also introduces new complexities. For example, the OMG Lightweight CCM [3] and Deployment and Configuration (D&C) [11] specifications have a large number of configuration points. To alleviate these complexities we applied MDD tools, such as PICML and WMGL. In our ITS case study, 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 PICML/WMGL models. Likewise, 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

MDD tool remains manageable due to the limited number of well-defined configuration *hot spots* exposed by the underlying infrastructure.

• *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. WMGL enbodies certain domain rules which reduce the probability of architectural rules violation discussed in Section 2 and ensure the proper usage of component middleware. Conversely, our experience applying PICML to model the ITS deployment structure indicated that it raised the level of abstraction at which developers work, enabling them to concentrate on certain aspects (*e.g.*, deployment structure) in the multidimensional problem space associated with applying component middleware for DRE systems. This separation of concerns helped eliminate many sources of accidental complexities and improved overall system quality.

The CoSMIC MDD tools and CIAO/DAnCE QoS-enabled component middleware are available in open-source form from `www.dre.vanderbilt.edu`.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

[14] S. Paunov, J. Hill, D. C. Schmidt, J. Slaby, and S. Baker, "Domain-Specific Modeling Languages for Configuring and Evaluating Enterprise DRE System QoS," in *Proceedings of 13th Annual International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '06)*, (Potsdam, Germany), IEEE, Mar. 2006.

[15] S. Paunov and D. C. Schmidt, "RepoMan: A Component Repository Manager for Enterprise Distributed Real-time and Embedded Systems," in *Proceedings of the 44th ACM Southeast Conference*, (Melbourne, FL), Mar. 2006.

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

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

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

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

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

[21] G. Trombetti, A. Gokhale, D. C. Schmidt, J. Hatcliff, G. Singh, and J. Greenwald, "An Integrated Model-driven Development Environment for Composing and Validating Distributed Real-time and Embedded Systems," in *Model Driven Software Development- Volume II of Research and Practice in Software Engineering* (S. Beydeda, M. Book, and V. Gruhn, eds.), New York: Springer-Verlag, 2005.