# Concern-based Composition and Reuse of Distributed Systems

Andrey Nechypurenko[1], Tao Lu[2], Gan Deng[2], Emre Turkay[2]
Douglas C. Schmidt[2], Aniruddha Gokhale[2]

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

[2] Institute for Software Integrated Systems
2015 Terrace Place
Vanderbilt University, Nashville, TN, USA, 37203
{tao.lu, gen.deng, emre.turkay,
d.schmidt, a.gokhale}@vanderbilt.edu

**Abstract.** Successful reusable software for large-scale distributed systems often must operate in multiple contexts, e.g., due to (1) integration with other systems using different technologies and platforms, (2) constant fine tuning needed to satisfy changing customer needs, and (3) evolving market conditions resulting from new laws and regulations. This situation causes vexing challenges for developers of reusable software, who must manage the variation between these contexts without overcomplicating their solutions and exceeding project time and effort constraints. This paper provides three contributions to R&D efforts that address these challenges. First, it motivates the use of a concern-based approach to enhance the level of abstraction at which component-based distributed systems are developed and reused to (1) improve software quality and developer productivity, and (2) localize variability aspects to simplify substitution of reusable component implementations. Second, we present our experience dealing with different domain- and middleware-specific variability gained applying model-based component middleware software technologies to develop an Inventory Tracking System that manages the flow of goods in warehouses. Third, we present a concern-based research strategy aimed at effectively managing the variability caused by the presence of multiple middleware platforms and technologies. Our experience to date shows that using model-based software tools and component middleware as the core elements of software composition and reuse – in conjunction with concern-based commonality and variability analysis – helps reduce development complexity, improve system maintainability and reuse, and increase developer productivity.

**Keywords**: Commonality/Variability Analysis, Concern, Aspect, Model Driven Architecture, Component Middleware, CORBA Component Model (CCM).

## 1. Introduction

**Emerging trends and challenges.** Developing large-scale distributed systems is complex, time-consuming, and expensive. Ideally, distributed systems are developed as product-line architectures [SAIP] that can be specialized in accordance with customer needs, hardware/software platform characteristics, and different quality of service (QoS) requirements. To minimize the impact of modifications, which is needed for specializing product-line architectures, architects and developers need to decouple the stable parts of their system from the variable parts [Coplien99].

One source of variability in large-scale distributed systems is the heterogeneity of APIs supplied by different operating systems (e.g., threading and synchronization mechanisms in Windows and UNIX [C++NPv1]), GUI toolkits (e.g., MFC, Motif, GTK, and Qt), and middleware (e.g., CORBA, J2EE, and .NET). Moreover, each technology provides its own mechanisms to configure and fine tune the system to address customer's needs. Another source of variability is different business rules (such as changing tax codes or air quality regulations) mandated by governments in different countries or regions. To maximize the benefits of software reuse, therefore, these sources of variability must be addressed via composition, encapsulation, and extension mechanisms that support alternative configurations and implementations of reusable functionality.

Sources of variability in large-scale distributed systems have historically been managed by raising the level of abstraction used to develop, integrate, and validate software. For example, the heterogeneity (and accidental complexity) of assembly languages in the 1960s and 1970s motivated the creation and adoption of standard third-generation programming languages (such as Ada, C, C++, and Java), which raised the abstraction level and helped improve the efficiency and quality of software development. Likewise, the complexity of developing large-scale systems from scratch on heterogeneous OS APIs motivated the creation and adoption of frameworks and patterns that provide application servers (such as CORBA, J2EE and .NET), which factor out reusable structures and behaviors in mature domains into standard reusable middleware.

Despite the advantages of refactoring commonly occurring capabilities into high-level reusable tools and services, challenges remain due to the diversity of alternatives for a given technology. For example, there are many different higher-level programming languages, frameworks, and middleware platforms that solve essential the same types of problems, yet are non-portable and non-interoperable. Ironically, many of these tools and services were positioned initially as integration technologies designed to encapsulate the heterogeneity of lower-level tools and services. This irony is caused by both the broader domain of systems that the new abstraction layer tries to cover, and the way in which the tools and services are

implemented. Over time, however, the collection of integration technologies simply became another level of heterogeneity that needs to be encapsulated by the next generation of integration technologies.

**Solution approach**: an Integrated Concern Modeling and Manipulation Environment. To address the challenges stemming from the heterogeneity of middleware platforms and to elevate the abstraction level associated with developing large-scale distributed systems using third-generation programming languages, we have been developing an Integrated Concern Modeling and Manipulation Environment (ICMME) that defines and manipulates fundamental concerns (such as remoting, component lifecycle management, communication and processor resource usage, and persistency) that represent higher level system building blocks than components or classes in object-oriented approach. Our ICMME combines key techniques and tools from Model Driven Architecture (MDA) [MDA], Aspect-Oriented Software Development (AOSD) [AOSD], and component middleware paradigms to provide a higher-level environment for developing large-scale distributed systems.

Experience we gained from developing frameworks [C++NPv1,C++NPv2] and middleware platforms [TAO1, CIAO1] enabled us to identify and document core patterns [POSA2] for managing different types of middleware variability. To evaluate the extent to which ICMME technologies help to address variabilities at different levels of abstractions (i.e., from the variability of configuring optional settings of a particular middleware platform up to the variability of handling different middleware platforms), we have developed a prototypical Inventory Tracking System (ITS). This paper uses our ITS prototype to illustrate the reuse benefits of ICMME-based integration by focusing on a fundamental concern – remoting – and then (1) developing an MDA model of a component-based remoting infrastructure according to the patterns described in [Voelter], (2) visually mapping ITS components to the roles defined by these patterns, and (3) localizing the impact of variability, caused by a need to support different middleware, by creating a domain-specific code generator to produce code for a real-time CORBA Component Model (CCM) [CCM] implementation called The Component Integrated ACE ORB (CIAO). Creating generators for J2EE and .NET component middleware remains a future work.

**Paper organization.** The remainder of this paper is organized as following: Section 2 describes the structure and functionality of our ITS prototype; Section 3 discusses the lessons learned thus far from applying our ICMME approach to the ITS case study; Section 4 compares our ICMME approach with related work; and Section 5 presents concluding remarks.

## 2. Overview of the ITS Case Study

An Inventory Tracking System (ITS) is a warehouse management system that monitors and controls the flow of goods and assets. Users of an ITS include couriers, such as UPS, FedEx, DHL, as well as airport baggage handling systems. A key goal of an ITS is to provide convenient mechanisms that manage the movement and flow of inventory in a timely and reliable manner. For instance, an ITS should enable human operators to configure warehouse storage organization criteria, maintain the set of goods known throughout a highly distributed system (which may span organizational and even international boundaries), and track warehouse assets using GUI-based operator monitoring consoles. This section presents an overview of the behavior and architecture of our ITS prototype and describes how we have integrated MDA tools with component middleware to enhance productivity and quality.

### 2.1 ITS System Behavior

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

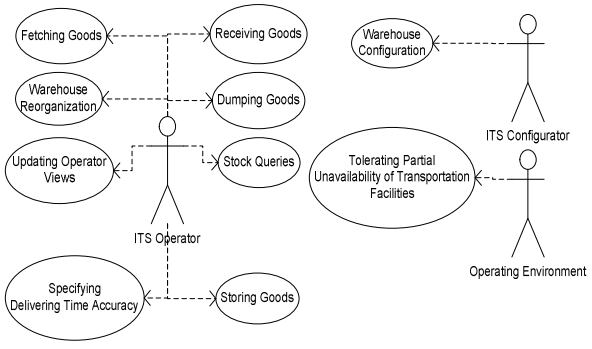


**Figure 1. Use Case Diagram for the ITS Prototype**

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

### 2.2 ITS Architecture

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

Figure 2 illustrates the key components that form the basic implementation and integration units of our ITS prototype. Some ITS components (such as the Operator Console component) expose interfaces to end users, i.e., ITS operators. Other components represent warehouse hardware entities (such as cranes, forklifts, and shelves) and expose interfaces to manage databases (such as Transportation Facility component and the Storage Facility component). Yet another set of components (such as the Workflow Manager and Storage Manager components) coordinate and control the event flow within the ITS system.

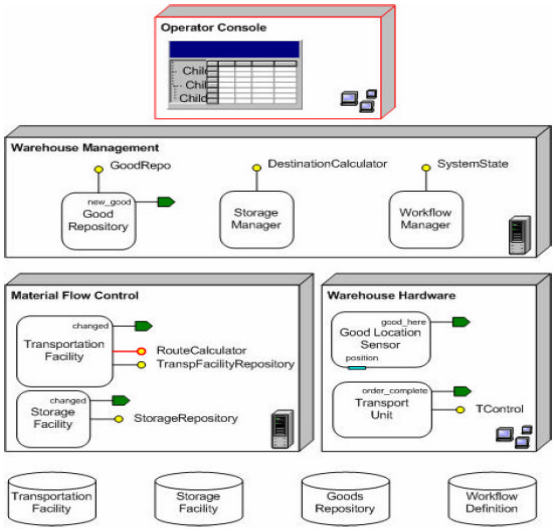


**Figure 2. Key CCM ITS Architecture Components**

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

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

The functionality of these three ITS subsystems is monitored and controlled via an Operator Console. All persistence aspects are handled via databases that can be managed either by the centralized DBMS or distributed DBMS over different DB servers. A typical interaction scenario between these three subsystems involves (1) a new good arriving at the warehouse entrance and being entered into the ITS either automatically or manually, (2) the WM subsystem calculating the final destination for storing the good by querying the Storage Facility for a list of available free locations and passing final destination to the MFC subsystem, (3) the MFC subsystem calculating the transportation route and assigns required transportations facilities, and (4) the MFC subsystem interacting with the WH subsystem to control the transportation process and if necessary adapt to changes, such as failures or the appearance of higher priority tasks.

### 2.3 Applying Component Middleware and MDA Tools to ITS

To evaluate how component middleware technologies can help improve productivity by enabling developers to work at a higher abstraction level than objects and functions, we selected the Component Integrated ACE ORB (CIAO) [CIAO1, CIAO2] as the run-time platform for our ITS prototype. CIAO is QoS-enabled CCM middleware built atop the The ACE ORB (TAO) [TAO1, TAO2]. TAO is a highly configurable, open-source Real-time CORBA Object Request Broker (ORB) that implements key patterns [POSA2] to meet the demanding QoS requirements of distributed real-time and embedded (DRE) systems.

CIAO extends TAO to provide the component-oriented paradigm to developers of DRE systems by abstracting critical systemic aspects (such as QoS requirements and real-time policies) as installable/configurable units supported by the CIAO component framework. Promoting these DRE aspects as first-class metadata disentangles (1) code for controlling these non-functional aspects from (2) code that implements the application logic, thereby making DRE system development more flexible and productive as a result. CIAO and TAO can be downloaded from deuce.doc.wustl.edu/Download.html.

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

1. Warehouse modeling, which simplifies the warehouse configuration aspect of the ITS system according to the equipment available in certain warehouses, including moving conveyor belts and various types of cranes. These modeling tools can synthesize the ITS database configuration and population.

2. Modeling and synthesizing the deployment and configuration (D&C) [D&C] aspects of the components that implement the ITS functionality. These modeling tools use MDA technology in conjunction with the CCM to develop, assemble, and deploy ITS software components.

These two aspects are relatively orthogonal to each other in terms of aspect separation, i.e., they depict the overall system from different perspectives, yet they are complementary to each other. For example, Figure 3 shows how the system modeler and warehouse modeler play different roles in the ITS development process.

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

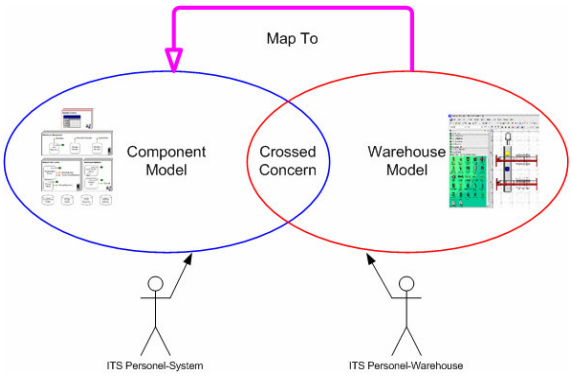


**Figure 3. ITS Modeling Aspects**

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

## 3. Towards an Integrated Concern Modeling and Manipulation Environment

Section 2.3 introduced a set of middleware components and modeling tools that helped increase the productivity and quality of our ITS development process. Based on our experience with the ITS prototype, we contend that to make software for distributed systems more reusable and resilient to future changes, we need to model and manipulate concerns separately. This section describes an Integrated Concern Modeling and Manipulation Environment (ICMME) that we are developing to achieve this vision.

### 3.1 Research Foundations

Based on our experience in developing large-scale distributed systems over the past two decades [POSA2, C++NPv1, C++NPv2], we have observed several problems that underlie the challenges outlined in Section 1. Below, we outline two of these problems and briefly describe how emerging technologies like MDA and AOSD [Ho02] could be exploited to address them.

**3.1.1. Low-level abstractions and tools** – Despite improvements in third-generation programming languages (such as Java or C++) and run-time platforms (such as component middleware), the level of abstraction at which business logic is integrated with the set of rules and behavior dictated by component models is still too low. For example, different component models provide different set of API and rules for component lifecycle management, example, e.g., there are multiple lifecycle management mechanisms available in CCM. As a result, this set of rules typically affects the implementation of business logic intrusively, i.e., the business logic developer implicitly assumes certain behavior from the component container and must adapt business logic accordingly. In addition, the level of abstraction and composition supported by third-generation languages does not intuitively reflect the concepts used by today's software developers [Mezini02], who are using higher level concerns (such as persistence, remoting, and synchronization) to express their system architectures.

A promising way to alleviate these problems with low-level abstractions and tools is to apply *Model Driven Architecture* (MDA) techniques [MDA] that express application functional and non-functional requirements at higher levels of abstraction beyond third-generation programming languages and conventional component middleware. At the core of the MDA is a two-level hierarchy of models:

- **Platform-independent models** (PIMs) that describe at a high-level how applications will be structured and integrated, without concern for the target middleware/OS platforms or programming languages on which they will be deployed. PIMs provide a formal definition of an application's functionality implemented on some form of a virtual architecture. For example, the PIM for the ITS could assume that there are two services available for each component: (1) a remote invocation service, i.e., an object request broker (ORB) and (2) an information storage and retrieval service, i.e. a database. At this stage it does not really matter whether these services are CORBA or SOAP and whether they use relational or object database, respectively.
- **Platform-specific models** (PSMs) that are *constrained* formal models that express platform-specific details. The PIM models are mapped into PSMs via *translators*. For example, the ITS uses the set of patterns and roles to describe the component collaboration infrastructure suggested by the OMG Component Collaboration Architecture [EDOC] that is specified in the PIM and could be mapped and refined to a specific type in the underlying platform, such as a QoS-enabled implementation of the CORBA Component Model (CCM) [CIAO1,CIAO2].

MDA tools use PIMs and PSMs to improve the understanding of software-intensive systems using higher-level models that (1) standardize the process of capturing business logic and quality of service (QoS)-related requirements and (2) ensure the consistency of software implementations with analysis information associated with functional and systemic QoS requirements captured by models. A key role in reducing software complexity via MDA tools is played by *meta-modeling* [GME], which defines a semantic *type system* that precisely reflects the subject of modeling and exposes important constraints associated with specific application domains.

3.1.2. Tangled concerns – Different concerns, such as component lifecycle management, resource usage, persistence, distribution, and safe/efficient cache and memory management, are often tangled within software source code, which impedes effective modularity [Kiczales]. If these concerns could also vary (either completely or partially) then the corre-

sponding incurred variability is also tangled with other concerns in source code and probably crosscut the whole system. Variability in commonly tangled concerns depends on many factors, such as deployment strategy and run-time conditions. For example, the communication latency between frequently communicating components depends on their distribution and deployment. Today, many applications are custom programmed manually to implement and compose these "cross-cutting" concerns, which is a tedious, error-prone, and non-scalable process. There are also limitations with third-generation programming languages that encourage the "tyranny of the dominant decomposition" [Tarr99], which involves the inability to apply different decomposition strategies simultaneously. Languages that support multiple inheritance address this problem to the limited extent, whereas languages without multiple inheritance make this task very hard.

A promising way to alleviate problems caused by tangled concerns is to apply Aspect-Oriented Software Development (AOSD) [AOSD] techniques. AOSD techniques go beyond object-oriented techniques to enable the design and implementation of separate cross-cutting concerns, which can then be woven together to compose complete applications and larger systems. In the absence of AOSD techniques, many concerns are tangled with the rest of the application code, thereby increasing software complexity. A key role in reducing complexity via AOSD techniques is modularization and separate handling of crosscutting concerns and generation of final application by means of aspect weaving tools [Gray1, Gray2].

### 3.2 Types of Changes caused by Variability

To understand what types of concern manipulation functionality should be supported by an ICMME, during the design and implementation stages of the ITS project we systematically captured the terminology we used to express what types of changes were made, as well as their consequences (i.e., affected interfaces, classes, and components). Moreover, based on our daily project experiences and by observing how typical change requests occurred, we observed that change requests were often formulated in terms of *features* of the system, such as adding, removing, or updating a certain capabilities. Only in trivial cases was just one particular class affected. Generalizing these observations and combining them with other experiences we have had developing ACE, TAO, and CIAO, it appears that most practical software systems have groups of logically connected classes and components that together implement the functionality of a particular concern. This implementation could be either localized at a certain place or crosscut several software artifacts. As a result, we conclude that changes could be categorized into the following types:

- *Local changes,* which are typically caused by errors and do not lead to changes in relationships between the core system classes/components and also do not change roles played by each particular class. A common example of local changes is refactoring measures [Refactoring], which are caused either by errors or by the need to organize the code better. Typically, such measures do not lead to role changes, which is why they are treated as "local."
- *Structural changes,* which sometimes occur due to (1) the need to add, remove or update some functionality, which is implemented (or supposed to be implemented) by several components/classes, each playing a certain role in collaboration to achieve the goal of supporting required functionality and (2) serious bugs which in turn lead to the redesign of the implementation structure of some functionality. For an example of structural changes, consider a new requirement to support dynamic (re)configuration of components that were statically compiled previously. This requirement can be addressed by the Component Configurator [POSA2] pattern, which allows an application to link and unlink its component implementations at run-time without having to modify, recompile, or relink the application statically. This change will require at least one class must derive from the base *Component* class and at least one class will be changed or introduced to play the *Component Repository* role described in the Component Configurator pattern. In such a situation, therefore, a set of classes must be changed together to handle the new requirement.

In this paper we concentrate on the structural changes because they usually affect several places within a software (sub)system, which indicates the existence of higher level relationships within the system. These relationships are usually manipulated as a whole, i.e., added/removed completely. As demonstrated in the patterns literature [POSA1, POSA2, GoF], it is possible to identify such stable relationships in the form of pattern languages.

There are also aspects and relationships that cannot be modularized using OO techniques [AOP]. For example, remoting, resource management, and transaction handling are concerns that are seldom modularized with OO techniques. If developers are committed to implementing a certain pattern, is they rarely implement them partially, e.g., the Observer [GoF] pattern combines both the observer and the subject. Even if some roles in a pattern are absent, their presence is assumed implicitly during implementation and will likely be provided later by some other developer or tool. These observations confirm the approach advocated by AOSD community about software systems as a set of (sometimes cross-cutting) concerns, which can be encapsulated using OO techniques (e.g., patterns) in some cases and in other cases new approaches are required (e.g., MDA and AOSD).

### 3.3 Raising the Abstraction Level

Based on the observation about type of changes and their typical impact on implementation presented in Section 3.2, we suggest using concerns as building blocks for an ICMME. A concern is a specific requirement or consideration that must be addressed to satisfy the overall system goal [RLaddad]. Each concern defines roles that could be played by that part of

the system that implement this concern. A key issue that must be resolved to use concerns effectively involves the relationship between concerns and the underlying business logic.

It is possible to think about concerns as *interfaces* in OO sense. In this case, the process of assigning components to certain roles defined by concerns could be treated as *implementation* of the concern. This approach leads to the interesting analogy between interface implementation (by means of inheritance, delegation, or any other technique) in OO sense and the same relationships between concerns and "base" code implementing a certain concern. For example, if we consider the ability to demultiplex callback events efficiently using the Asynchronous Completion Token (ACT) [POSA2] as the concern – and there is a set of classes implementing ACT – then we can (roughly) say that this set of classes "is an" efficient callback demultiplexer and they can be used wherever ACT functionality is expected without visible difference to the ACT users who rely on ACT functionality. According to the Liskov Substitution Principle (LSP) [LSP], this relationship between abstract description of efficient demultiplexing concern (encapsulated using ACT design pattern) and set of classes implementing ACT feature is inheritance.

By defining such fundamental relationships between concerns presented in the form of design patterns or any other role-based definition of some functionality and implementation of this functionality as a role mapping to available business logic, we can provide a powerful mechanism to encapsulate the variability at a higher abstraction level than that provided by conventional third-generation programming languages – in particular, we can encapsulate the impact of middleware platform variability on the rest of the system. The primary advantage of this approach is the ability to systematically introduce changes to the system using roles, defined by concerns. For example, if developers want to add a Visitor pattern [GoF] implementation to the code, a wizard support by the modeling tools could guide the user through the role mapping process to make sure that all roles defined by the Visitor pattern are mapped by the developers to the appropriate implementation classes.

Figure 4 provides the high level view on the complete ICMME modeling process. This figure shows how domain-specific models are used as input for modeling application-specific business logic and either selecting existing or creating new reusable concern models. After completing the role mapping process, the platform-specific model will be generated, followed by the assembly of the complete executable application.

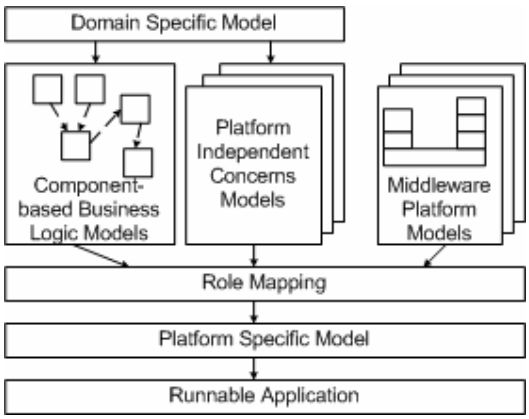


**Figure 4. ICMME Concern-based Modeling Process**

The OMG EDOC standard [EDOC-Patterns] addresses the need for role-based description of higher level functionality by standardizing a set of modeling and meta-modeling capabilities to ensure consistency and interoperability between different modeling tools. In addition, OMG's MDA approach standardizes mappings from platform-independent system models (PIMs) to popular platform-specific middleware (PSMs), such as CORBA, EJB and WebServices.

### 3.4 Handling Middleware Platform Variability Via Concern-based ITS Design

To provide a concrete illustration of the idea of concern-based variability localization within our ICMME, we now describe how we are supporting (1) platform-independent definitions of the remoting concern of ITS, which defines the mechanisms for passing messages between components instantiated in different processes and possibly running on different hosts and (2) platform-specific mappings of the remoting concern to various component middleware platforms, such as CCM or EJB. After completing the high-level domain-specific modeling steps described in Sections 2.2 and 2.3, we next specify the set of components corresponding to the domain-specific model elements, as well as the way these components communicate with each other. This specification process can be guided interactively by a model-based tool, such as the wizards used to configure various types of tools on Windows platforms.

For example, to specify and manipulate the remoting concern the modeling tool can guide the developer through the three steps shown in Figure 5 and described below:

1. As shown in in Figure 5, step 1 involves choosing the fundamental communication paradigm, such as Asynchronous Message Model or RPC Communication Model using the Broker pattern [POSA1]. Selecting the communication paradigm provides a more detailed specification of the roles needed to support a particular communication type.
2. The refinement process is shown as step 1 in Figure 5, which uses an interactive modeling tool to refine the model based on the set of available patterns for client and server implementations of Broker-based distributed systems [Voelter, Kircher]. In this step, a more fine-grained model of the roles played by components in the broker-based distributed system is presented. This refinement process could be performed by platform-specific roles to support various component middleware implementation platforms, such as CORBA CCM or Sun's EJB.
3. Step 3 of Figure 5 show how the interactive modeling tool can be used to allow developers to deploy and configure each component according to the roles defined by concerns. In this step, developers can map the high-level architecture blocks presented in Figure 1 to the corresponding roles according to the selected remoting paradigm, which is formalized in form of patterns according to the EDOC specification [EDOC-Patterns]. *Mapping* is the process of defining which part of a component plays the role(s) expected by certain elements of a concern. One way to perform this mapping is to apply the on-demand remodularization technique described in [Mezini03].

The three steps presented above in conjunction with the ICMME provide the following improvements in handling variability aspects:

- Using patterns as an abstract base for a family of different implementations for certain aspects of a distribute system provides a variability encapsulation mechanism similar to inheritance in object-oriented programming language, but at a higher level of abstraction. As a result, reusability can be achieved at large scale and larger parts of distributed applications are shielded from changes caused by variations.
- Pattern-based modeling of different aspects formally describes what is expected and provided by certain components and subsystems, thereby forming a solid foundation for formally defining controllable and verifiable substitution of implementation parts. Having such formalized descriptions helps reduce accidental complexities caused by inconsistent combination of semantically inconsistent variable parts.
- Support for model refinements using iterative model-based tools helps to simplify the process of finding variation points for complex cases where it is hard for system architects and developers to foresee all variability aspects in advance.
- Code generation using multiple fine-granularity model interpreters explicitly tuned for certain modeling functionality addresses the problem of overly complex code generators, which could otherwise obviate the benefits of the higher-level modeling techniques described in this paper.

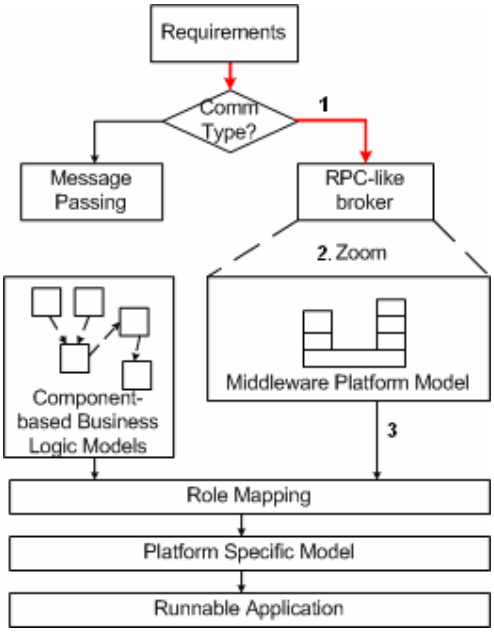


**Figure 5. Remoting Aspect Encapsulation using Mapping Technique**

## 4. Related Work

This section reviews work related to our integrated concern and model manipulation environment (ICMME) and describes how modeling, domain analysis, and generative programming techniques are being used to model and provision component-based distributed systems to handle variability more effectively than conventional software development techniques.

Our work on ICMME extends earlier work on Model-Integrated Computing (MIC) [Janos:97, HarelGery:96, Lin:99, Gray:01] that focused on modeling and synthesizing embedded software. Examples of MIC technology used today include GME [GME:01] and Ptolemy [Lee:94] (used primarily in the real-time and embedded domain) and Model Driven Architecture (MDA) [MDA] based on UML [UML:01] and XML [XML:00] (which have been used primarily in the business domain). Our work on ICMME combines the GME tool and UML modeling language to model and synthesize component middleware used to configure and deploy distributed applications.

Generative programming (GP) [Czarnecki] is a type of program transformation concerned with designing and implementing software modules that can be combined to generate specialized and highly optimized systems fulfilling specific application requirements. The goals of GP are to (1) decrease the conceptual gap between program code and domain concepts (known as achieving high intentionality), (2) achieve high reusability and adaptability, (3) simplify managing many variants of a component, and (4) increase efficiency (both in space and execution time). Our ICMME approach uses GP to map models to the C++ code during the code-generation phase. Our code generator for CIAO CCM container produces code, which utilizes many techniques associated with GP. Using pure GP, however, can be labor intensive, tedious, and error-prone to compose consistent set of components together. To avoid these problems, our ICMME code generation approach complements GP technique by assuring that only consistent set of components will be composed by introducing high-level model constraints that will be processed accordingly by model interpreter during code generation and model validation phases.

Aspect-oriented software development (AOSD) is a GP technology designed to more explicitly separate concerns in software development. AOSD techniques make it possible to modularize crosscutting aspects of complex distributed systems. An aspect is a piece of code or any higher level construct, such as implementation artifacts captured in a MDA PSM, that describes a recurring property of a program that crosscuts the software application, i.e., aspects capture crosscutting concerns.

Scope, Commonality, and Variability (SCV) analysis [Coplien99] is related work on domain engineering that focuses on identifying common and variable properties of an application domain. SCV uses this information to guide decisions about where and how to address possible variability and where the more "static" implementation strategies could be used. Our ICMME approach complements CSV at the phase where the step from abstract definition of commonality and variability aspects should be transformed to the model, which is concrete enough for the code generation. In the nomenclature of SVC, PIMs represent the common aspects of distributed systems, whereas PSMs implement the variability aspects. SCV defines the basic principles and procedure to capture commonality and variability. We enhance this approach by formalizing the models up to the level where captured commonality and variability could be processed by model interpreter to produce working C++ or Java source code.

## 5. Concluding Remarks

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

Our experience gained from applying ICCME to our ITS prototype can be summarized as follows:

- Even for mid-size distributed systems (e.g., consisting of around 20 to 50 architectural components), the complexity reduction stemming from model-driven code generation can be obviated by an increase in model interpreter complexity caused by overly general models and interpreters. To address this problem, the code generation process can be split into several intermediate steps – such as platform-independent models (PIMs) to platform-specific models (PSMs) to source code transformation – to improve reusability of the model interpreter itself. This structure helps to simplify model interpreters since each interpreter containing less functionality. It is therefore possible to substitute only certain modules of the interpreter to serve different application needs, thereby achieving better reusability at the model interpreter level.

- Complexities related to the existence of variable parts within the software systems must be addressed in systematical way via formalized descriptions of integration points and mechanisms for variable parts of the ITS architecture. Role-based abstract definition of such points can be used for these purposes. For cases where best practices are documented

in the form of patterns, it is beneficial to use patterns as role-based platform-independent formalization mechanism within models. Patterns can play the same role in distributed system architectures that abstract classes play in object-oriented design, where they provide a common interface for a family of related implementations. Having such an abstract, pattern-based "interface" – with many possible mappings to implementation artifacts – helps to localize the variable aspects and shields other system components from changes resulting from providing new implementations for certain variable aspects.

In future work, we plan to enhance our ICMME by supporting open standards, such as the OMG Meta-Object Facility (MOF) [MOF] and Enterprise Distributed Object Computing (EDOC) [EDOC] specifications to support a wide range of usage patterns and distributed applications. These efforts will create and validate a broader range of modeling tools that cover key modeling- and middleware-related aspects.

## References


[AOP] G. Kiczales, "Aspect-Oriented Programming", Proceedings of the 11th European Conference on Object-Oriented Programming, Jun 1997.

[AOSD] http://www.aosd.net

[C++NPv1] Douglas C. Schmidt and Stephen D. Huston, C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns, Addison-Wesley, Boston, 2002.

[C++NPv2] Douglas C. Schmidt and Stephen D. Huston, C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks, Addison-Wesley, Reading, Massachusetts, 2002.

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

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

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

[Clements] Paul Clements and Linda Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.

[Coplien99] J. Coplien, D. Hoffman, D. Weiss, "Commonality and variability in software engineering", IEEE Software, November/December 1999, pp. 37-45.

[Czarnecki] Krzysztof Czarnecki, Ulrich Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley Pub Co. ISBN: 0201309777

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

[EDOC] OMG adopted specification: „UML Profile for Enterprise Distributed Object Computing Specification " Document number ptc/02-02-05.pdf

[EDOC-Patterns] Object Management Group. UML Profile for Patterns, v1.0. formal/04-02-04.

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

[GME01] Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., Volgyesi P.: The Generic Modeling Environment, Workshop on Intelligent Signal Processing, accepted, Budapest, Hungary, May 17, 2001.

[GoF] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.

[Gray:01] Jeffery Gray, Ted Bapty, and Sandeep Neema, "Handling Crosscutting Constraints in Domain-Specific Modeling," Communications of the ACM, pp. 87–93, Oct. 2001.

[Gray1] Jeff Gray, Janos Sztipanovits, Ted Bapty Sandeep Neema, Aniruddha Gokhale, and Douglas C. Schmidt, "Two-level Aspect Weaving to Support Evolution of Model-Based Software," Aspect-Oriented Software Development, Edited by Robert Filman, Tzilla Elrad, Mehmet Aksit, and Siobhan Clarke, Addison-Wesley, 2003.

[Gray2] Sandeep Neema, Ted Bapty, Jeff Gray, and Aniruddha Gokhale, "Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems," in Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, PA, Oct. 2002.

[HarelGery:96] David Harel and Eran Gery, "Executable Object Modeling with Statecharts", Proceedings of the 18th International Conference on Software Engineering, pp. 246-257, IEEE Computer Society Press 1996.

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

[Ho02] Wai-Meng Ho, Jean-Marc Jezequel, Francois Pennaneac' h, and Noel Plouzeau, "A Toolkit for Weaving Aspect-Oriented UML Designs," First International Conferenceon Aspect-Oriented Software Development, Enschede, The Netherlands, April 2002

[Janos:97] J. Sztipanovits and G. Karsai, "Model-Integrated Computing," IEEE Computer, vol. 30, pp. 110–112, Apr. 1997.

[Kiczales] G. Kiczales, AOP The Fun has Just Begun, New Visions for Software Design & Productivity Workshop, Vanderbilt University, Nashville, TN, 2001;

[Kircher] M. Kircher and P. Jain, Pattern-Oriented Software Architecture - Patterns for Resource Management, Volume 3, John Wiley & Sons, will be published in May 2004.

[Lee:94] 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.

[Lin:99] M. Lin, Synthesis of Control Software in a Layered Architecture from Hybrid Automata, in: HSCC, 1999, pp. 152–164.

[LSP] Barbara Liskov, "Data Abstraction and Hierarchy," SIGPLAN Notices, 23, 5 (May, 1988).

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

[MOF] OMG: "Meta-Object Facility, version 1.4", Jan 11, 2002.

[Mezini02] Mira Mezini and Klaus Ostermann, "Integrating Independent Components with On-demand Remodularization," in To appear in the Proceedings of the 17th ACM Conference on Object-Oriented Pro-gramming, Systems, Languages, and Applications (OOPSLA'02), Seattle, Washington, USA, November 2002, ACM.

[Mezini03] M. Mezini and K.Ostermann. Conquering Aspects with Caesar. In (M. Aksit ed.) Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), March 17-21, 2003, Boston, USA. ACM Press, pp. 90-100

[POSA2] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2. New York: Wiley & Sons, 2000.

[Refactoring] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts. Refactoring: Improving the Design of Existing Code. Addison-Wesley Pub Co. ISBN: 0201485672

[RLaddad] Ramnivas Laddad, "AspectJ in action. Practical Aspect-Oriented Programming". Manning Publications Co. ISBN 1-930110-93-6

[SAIP] Len Bass, Paul Clements, Rick Kazman. Software Architecture in Practice, Second Edition. Addison-Wesley Pub Co. ISBN: 0321154959

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

[TAO2] Douglas C. Schmidt, David L. Levine, and Chris Cleeland, Architectures and Patterns for High-performance, Real-time CORBA Object Request Brokers, Advances in Computers, Academic Press, Ed., Marvin Zelkowitz, 1999.

[Tarr99] Peri Tarr, Harold Ossher, William Harrison, and Stanley Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," International Conference on Software Engineering (ICSE), Los Angeles, California, May 1999, pp. 107-119.

[UML:01] Object Management Group, Unified Modeling Language (UML) v1.5, OMG Document formal/2003-03-01 Edition (Mar. 2003).

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

[XML:00] W. A. Domain, Extensible Markup Language (XML), http://www.w3c.org/XML