TOWARD ENHANCING REUSABILITY OF COMPONENT MIDDLEWARE DSMLS USING GENERALIZATION AND STEP-WISE REFINEMENT

By

Ritesh Neema

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

May, 2010

Nashville, Tennessee

Approved:

Professor Aniruddha Gokhale

Professor Jules White

To my family.

ACKNOWLEDGMENTS

I would like to take this opportunity to express my sincere gratitude towards my academic and research advisor Dr. Aniruddha Gokhale for helping me through this academic pursuit. I am grateful to him for providing me with the necessary guidance and supervision, for constantly motivating me to finish this research, and for his humbleness and kindness to allow me to study at Vanderbilt University and conduct this research.

I also extent my sincere thanks to Dr. Jules White, my second reader, for his thorough evaluation of this thesis and providing his detailed and invaluable feedback.

I am indebted to many in ISIS, including Sumant Tambe, Anantha Narayanan, Csaba Toth, and Peter Volgyesi, for technical assistance and help.

I am also thankful to Vanderbilt University for giving me the opportunity to study and pursue my Masters degree.

On the personal note I own my loving thanks to my brother Himanshu Neema and sister-in-law Reena Neema for understanding, supporting, motivating, and cheering me and for always being there for me. Finally, but not the least, I offer my sincere thanks to my family including, Suresh Chandra Neema, Geeta Neema, Himanshu Neema, Reena Neema, Kavish Neema, Supriya Neema, Vaibhav Doshi, Purvashi Doshi, Raina Neema, Sanya Neema, and Hiya Doshi, who's love, support, best wishes and prayers were always with me.

Ritesh Neema

Vanderbilt University

May 2010

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVATIONS

AOP	Aspect-Oriented Programming
API	Application Programming Interface
CCM	CORBA Component Model
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
COTS	Commercial-Off-The-Self
CVA	Commonality/Variability Analysis
DLL	Dynamic Link Library
DRE	Distributed Real-time and Embedded systems
DSML	Domain Specific Modeling Language
EJB	Enterprise JavaBeans
ESML	Embedded Systems Modeling Language
GCML	Generic Component Modeling Language
GME	Generic Modeling Environment
GPS	Global Positioning System
GReAT	Graph Rewriting And Transformation
GUI	Graphical User Interface
J2EE	Java 2 Platform, Enterprise Edition
J2EEML	Java 2 Platform, Enterprise Edition Modeling Language
J2SE	Java 2 Platform, Standard Edition
MBSE	Model-Based Software Engineering
MDE	Model-Driven Engineering
MMS	Magnetospheric Multi-Scale
OS	Operating System
PICML	Platform Independent Component Modeling Language
QoS	Quality-of-Service
SCE	Shipboard Computing Environment

CHAPTER I

INTRODUCTION

Standardized component middleware technologies, such as *CORBA Component Model* (CCM) [1], *Enterprise JavaBeans* (EJB) [2], and *Component Object Model* (COM) are used to build large-scale *Distributed Real-time and Embedded* (DRE) systems. A key requirement of these middleware is that they remain highly flexible and support a large number of features since they have to be applicable to a wide range of domains and applications. To enhance flexibility of component middleware technologies, there exist *Domain Specific Modeling Languages* (DSMLs) [4], such as *Platform Independent Component Modeling Language* (PICML) [5] for CCM and *Java 2 Platform Enterprise Edition Modeling Language* (J2EEML) [6] for EJB, that are used to apply *Model-driven Engineering* (MDE) [3] approaches to DRE systems for different platforms.

I.1 Motivation

The emergence of DSMLs for *commercial-off-the-self* (COTS) component middleware technologies significantly enhances the application development process by addressing several challenges including level of abstraction, reusability, and automation. Despite these improvements, the full potential of DSMLs remains to be realized.

Although these component middleware DSMLs reduces the complexity of the middleware technologies by increasing the level of abstraction, these modeling languages are themselves complex enough to overwhelm the developers. In the early stages of computing the languages included only a few hundred features. However, with the growth of platform complexity that has evolved faster than the ability of the languages to mask it, these languages have also become complex with hundreds and thousands of features. Even for a simple application the developers have to work with a very complex language with thousands of features. This deficiency of the middleware DSMLs can divert the application developers to focus on the important features of the applications such as components and cause them to easily miss or make mistake with the important features. This will considerably increase the development efforts of developing and debugging application models.

Furthermore, we have seen that every second a new technology is taking birth. The world is moving fast with constant innovation of new technologies. For example, in middleware technologies, when the term middleware first appeared, there were only few middleware technologies known. The term was associated mainly with relational databases for many practitioners in the business world. This is no longer the case now. Concepts similar to today's middleware previously went under the names of network operating systems, distributed operating systems, and distributed computing environment. The middleware technologies are upgraded to more advanced level where they are highly flexible and support a large number of features. In addition, as the practitioners keep moving to new middleware technologies, several existing middleware technologies fall behind and no longer remain useful.

I.2 Problem Statement

As a consequence of evolution in middleware technologies, the middleware DSMLs become complex and causes the development process of the application models to consume excessive time and effort. Even a simple application requires considerable amount of time and effort thereby making the development process arduous. Also, the application models that are developed for the outdated middleware technologies must also be upgraded to preserve intellectual property and investments. Furthermore, to move the application model from one middleware technology to another can also be required because of client requirements or other business reasons. While developing, upgrading, and moving these application models, software developers are increasingly faced with the challenges of complexity and migration. One of the possible ways of addressing the issue of migration is to use the traditional approach of creating the application model in one of the COTS middleware technologies from scratch by referring to the existing application model. Although this traditional approach reduces the development efforts to some extent by allowing software developers to leverage an existing application model for developing application models in multiple middleware technologies, it is still largely low-level, tedious, complex, error-prone, and technology specific. Moreover, it also assumes that the earlier models are well-documented and fully capture all design decisions.

A solution to this problem is to provide a mechanism that raises the level of abstraction and enhances the reusability and automation capabilities of the DSMLs to reduce the development efforts significantly. Visualization has always been an effective way to communicate both the abstract and concrete ideas. Visual tools such as DSMLs for MDE and *Graphical User Interfaces* (GUI) for third generation programming languages hold the same promise for middleware application development. A visual tool that raises level of abstraction and enhances the reusability and automation during middleware application development process to make the applications reusable for migration and to simplify the development process is urgently needed. Researches in techniques such as Step-wise Refinement [7] have shown promise in simplifying the application development processes by increasing the level of abstraction and by applying *generalization/specialization* techniques in a step-wise manner. Similarly, GUI and MDE have also shown promise in increasing the level of abstraction while enhancing the reusability, efficiency and ease of use for the underlying logical design. These GUI and MDE tools combined with the step-wise refinement technique largely simplify the development process of applications by reducing complexity and by eliminating the overhead of repetitive and error-prone manual process, thus enhancing the reusability and automation of the application models.

I.3 Research Approach

In this research we synthesize the capabilities of *Commonality/Variability Analysis* (CVA) [8], *Step-Wise Refinement* [7], and *Model Integrated Computing* (MIC). In this context, we develop a Model-driven Feature-Refinement Programming tool chain. This tool chain comprises of a DSML called *Generic Component Modeling Language* (GCML), platform specific GUIs, and model interpreters that apply the combination of MDD and Feature Refinement technologies to component middleware technologies. Using this model-driven feature-refinement technique to generate application model can result in the automation and simplification of migration and development of application model by enhancing reusability, increasing level of abstraction, and by removing the error-prone manual steps involved in migration and specialization of middleware.

We demonstrated our design by considering two main aspects of the application development process: (a) develop an application model for a given middleware platform from scratch using one of the existing DSMLs is shown, and (b) transform an existing application model from one middleware platform to another. In the first aspect, we describe the step-wise feature refinement technique –which is a powerful paradigm for developing a complex program from a simple program by adding features incrementally. The steps in this technique include: (a) development of a generalized DSML called GCML, based on CVA technology, which enables the developers to define the component modeling features at a very high level of abstraction and reuse it to generate a platform specific model for any of the supported middleware technologies, (b) development of the GUIs for providing middleware technology specific features to refine the abstract model developed using GCML, and (c) development of the platform specific application model using the middleware technology specific DSMLs. In the second aspect, we describe an automation technique for migration of an existing application model developed in one of the middleware technology DSMLs to a different middleware platform or to a newer version of the same middleware platform. In this automation technique, the existing application model is first generalized into a GCML abstract model, which can be further transformed into an application model specific to any of the middleware technology DSMLs using the steps described in the first aspect.

I.4 Thesis Organization

The rest of the thesis is organized as follows: In Chapter II, we introduce model-driven engineering and component middleware. We also describe PICML and J2EEML – DSMLs developed using model-driven engineering for CCM and EJB respectively. Chapter III illustrates the problem in using model-driven engineering across multiple middleware technologies and lists the issues involved with using the existing approaches. In Chapter IV, we describe the related research work used for model transformation and to enhance the reusability in model-driven engineering. Chapter V describes our solution approach and presents the modeling details of GCML along with platform-specific GUIs and Interpreters. In Chapter VI, we describe a case study using our proposed approach to address various challenges in developing a middleware application model using PICML paradigm and then migrating it into a J2EEML application model. We present this case study for an application model called Basic Single Processor (BasicSP) developed using PICML. In Chapter VII, we present our experimental results and analyze the effectiveness of our approach with respect to abstraction, reusability, automation, flexibility, and efficiency. Finally, in Chapter VIII, the thesis concludes and identifies areas for future work.

CHAPTER II

RELATED WORK

DSMLs significantly enhance the component middleware application development process by addressing several challenges including simplification, abstraction, reusability, and automation. Despite this improvement, the reusability and complexity of the component middleware DSMLs remain low with respect to the concepts and concerns of the application domains. In this chapter, we describe current techniques for reusability enhancement of DSMLs and for simplification and automation of the application model development process. We also emphasize on how our work differs from existing techniques.

II.1 Research on Automation of Application Model Development

There exists a wide range of techniques that focus on automation of the application model development process and on increasing the reusability of the DSMLs. For instance, model transformation technique - that takes a model conforming to a given metamodel as input, and converts it into another model conforming to a different metamodel. Model transformation is a highly active area of research focusing on automation and reusability of models and modeling systems. To list a few, the work presented by Y. Lin et. al. [9] describes a high-level aspectual model transformation language that is designed to specify tasks of model construction and evolution, and uses a model transformation engine to execute transformation specifications in an automated manner. Also, the work presented by Amogh Kavimandan [10] focuses on reusable model transformation techniques for automating middleware QoS configuration in DRE systems. Furthermore, the *Graph Rewriting And Transformation* (GReAT) [11] tool – developed using GME at the Institute for Software Integrated Systems (ISIS) – can be used to define transformation rules using its visual language in terms of source and target languages (i.e., metamodels), and to execute these transformation rules to automatically generate target models using the GReAT execution engine (GR-Engine).

II.2 Research on Enhancing Reusability in Model-driven Engineering

One of the approaches used to enhance the reusability in model-driven engineering is *Aspect-Oriented Programming* (AOP) [12]. AOP is primarily used for separation of concerns that cut across multiple application domains and reduction of development efforts needed to support the evolution of large-scale system models. For example, the work presented by C. Zhang et. al. describes a Modelware methodology [15] that combines the capabilities of Model-Driven Architecture (MDA) approach and AOP to separate the intrinsic and extrinsic functionalities of middleware. It reduces the development efforts needed to support the evolution of middleware functionalities by lowering the concern density per component and enhancing the reusability of components of middleware architectures. Also, POSAML [14] is yet another technique that uses the MDE and AOP approaches for middleware specialization. It allows modifying an existing functionality without refactoring any code, addresses concerns with minimum coupling, and makes it easy to add new functionality by creating reusable aspects. With these capabilities, POSAML significantly enhances the reusability of component middleware DSMLs and considerably automates middleware specialization.

Our work differs from existing approaches in the following way. Model transformation is an application-specific technique to reuse models and automate their migration across various platforms. On the other hand, the AOP approach is a domain-specific approach that automates middleware specialization only for newer versions of the same domain (with added functionalities). It does not support reusability and automation across multiple middleware platforms. However, our work enables reusability of component middleware DSMLs and simplification and automation of the development process for developing a new application as well as for migration of application models to newer versions of the same platform and across multiple middleware platforms.

CHAPTER III

BACKGROUND: MODEL-DRIVEN ENGINEERING FOR COMPONENT MIDDLEWARE TECHNOLOGIES

"Model-driven engineering technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively."

- Douglas C. Schmidt, MDE, February 2006

In this chapter we provide an overview of component middleware and model-driven engineering which are integral to this research. We describe CCM and EJB component middleware which are chosen as the base of illustration of the conceptual idea behind our research. We also illustrate how domain-specific modeling languages for component middleware technologies alleviate the complexity and express domain concepts effectively using the examples of PICML and J2EEML DSMLs for CCM and EJB respectively.

III.1 Overview of Component Middleware

***Middleware**:* Middleware is the reusable software that lies between the applications and the underlying operating systems, network protocol stacks, and hardware. The primary function of middleware is to connect application programs with the hardware and software components and to mediate interactions between the parts of an application, or between applications. One of the major achievements in introducing middleware in the software development process is that it alleviates complexities associated with developing software applications to a great extent. Middleware is a high-level building block that shields application-specific functionality from complex lower-level details. This decoupling of application from lower-level details allows developers to focus on application-specific functionalities, rather than spending excessive amount of time with lower-level infrastructure challenges. Recently, middleware has emerged as highly effective in building *enterprise applications*. These enterprise applications are complex, scalable, distributed, and component-based, and require mission-critical application software that can perform business functions such as accounting, production scheduling, and customer information management and maintenance. They are frequently hosted on servers and PCs and simultaneously provide services to a large number of enterprises, typically over a computer network and are developed using COTS component middleware.

Component middleware is a special class of middleware that manages the life-cycle of components, handles interactions between them, and enables reusable component-based services to be composed, configured and installed to build enterprise applications and DRE systems more rapidly and robustly. Component middleware overcomes the limitations of object-oriented middleware, such as excessive manually performed tasks, difficult to understand application structure, difficult to modify or extend an existing application, and many more. It addresses these limitations by shifting the main focus of programming from objects to components provided with well-defined interfaces to interact with them. These components are then assembled to build and execute applications on the servers. In particular, the motivations for component middleware for enterprise application developers are as follows:

- Building applications by composing existing components.
- Illustrating interactions between components with formalism.
- Notion of connector: Defining software architecture by connecting components with one another.
- Describing deployment of components with formalism
- Separation of functional and non-function aspects to allow reusability and thereby enabling developers to focus on application concerns (functional) rather than low-level integration problems (non-functional).

Examples of COTS component middleware include the *Common Object Request Broker Architecture* (CORBA), Component Model (CCM), Enterprise JavaBeans (EJB), and Common Object Model (COM) – each of which varies in the APIs, protocols, and component models that it uses. In the next section, we describe two of these component middleware technologies in more detail, viz. CCM and EJB.

CORBA Component Model (CCM)

CCM is a server-side component model for building and deploying CORBA applications. It uses accepted design patterns and facilitates their usage by enabling a large amount of code to be generated. This also allows system services to be implemented by the container provider rather than the application developer. The CCM extends the CORBA object model by defining features and services in a standard environment that enables application developers to implement, manage, configure and deploy components that integrate with commonly used CORBA services. These server-side services include transactions, security, persistence, and events.

The CCM specification introduces the concept of components and a comprehensive set of interfaces and techniques for specifying implementation, packaging, and deployment of components. Components encapsulate business logic and interact with other components via ports. Figure 1 show the key elements of the CCM model which includes:

- ***Container:*** This provides a run-time execution environment, encapsulates component implementations, and provides system services such as lifecycle, transactions, persistence, and security. These services act as the interface between a component and the outside world and allow access to any component through container-generated methods which in turn invoke the component's methods.
- ***Component Assembly:*** This is a higher-level abstraction that is used to describe component compositions, including component locations and interconnections between components.
- ***Components:*** These are the implementation entities that export a set of interfaces to clients. Components can also express their intent to collaborate with other components by defining ports that specify how components interact.
- ***Component Home:*** This provides operations to manage components in an application. It consists of two main operations: Factory operations, which are used to create an instance of the specific component type, and Lookup operations, which are used to retrieve components from a database or repository.

- ***Component Ports*****:** These allow components to interact with the outside world as well as other components. These ports have an extension interface pattern that provides multiple interfaces for the clients and other components to interact with the components. CCM comprises of four kinds of ports: (i) ***Facets,*** which provide access to specific component methods through different interfaces with unique names, thus provide multiple views to its clients, (ii) ***Receptacles,*** which are interfaces that allow components to interact with each other by connecting them with the interacting components' objects and invoking methods upon these objects, (iii) ***Event sources/sinks,*** which are interfaces that allow components to establish a publisher/subscriber pattern between them. A component is called a publisher if it publishes or emits an event by declaring an event source, whereas a component is called a subscriber if it shows interest in consuming those events by declaring event sinks, and (iv) ***Attributes:*** These are named configurable properties that can be accessed and modified by the corresponding operations to perform an action or to raise exceptions based on the value of the attributes.

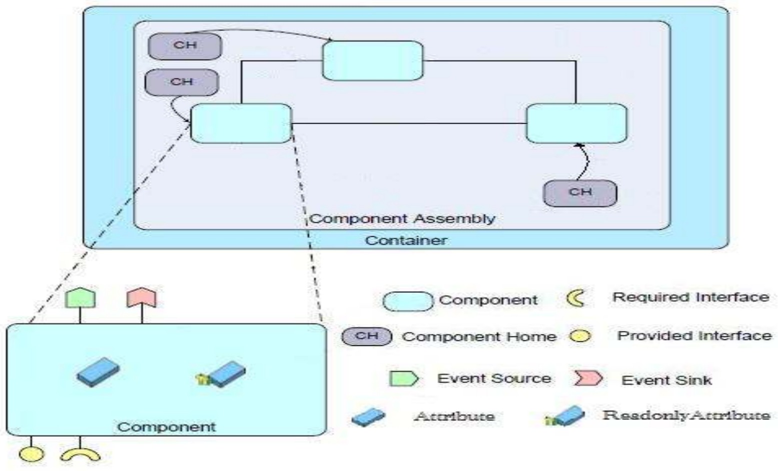


Figure 1: Key Elements in the CORBA Component Model

Enterprise JavaBeans (EJB)

Java 2 Platform Enterprise Edition (J2EE) is a Java-platform centric environment that allows developers to develop, build and deploy enterprise applications. J2EE reduces the complexity of enterprise applications by building them as assembly of well-defined and easy to use components by supporting with component services and performing several functions automatically. J2EE is the advanced version of Java 2 Platform Standard Edition (J2SE) - takes advantages of many features of J2SE and provides full support for Enterprise JavaBeans (EJB) (i.e., *business logic layer),* Java Servlets API, and JavaServer Pages (i.e., *presentation layer)*.

The Enterprise JavaBeans (EJB) is an architecture that enables a simplified approach for the development and deployment of component-based robust business applications. This EJB is essentially a managed component that resides in the J2EE container, which manages the life-cycle of the components. EJB technology allows component developers to focus on business logic by concealing application complexity in a multitier application development.

EJB technology enables developers to model full range of objects that are useful in the enterprise applications. As shown in Figure 2, the key elements of the EJB architecture include:

- ***EJB Server:*** This is a process or application that provides a run-time environment for the execution of server applications that uses enterprise beans. It contains the EJB container and provides the services required by the enterprise beans.

- ***EJB Container***: This provides life-cycle management and other services for the EJB components. An EJB container intercedes between clients and components and manages the invocation of component methods by clients or other containers running on different servers or machines.
- ***EJB Component or EJB Bean:*** This is a server component consisting methods that typically provide business logic in distributed applications. These methods are invoked by the ***EJB Client*** and result in a database update. The types of EJB components that can be implemented are as follows: (i) ***Session Beans***, which are the non-persistent enterprise beans that represent client session's behaviors. Session beans are of two types: stateless and statefull. Stateless session beans are client specific and maintain single client's session information related to multiple method calls and transactions. Statefull session beans are not client-specific and are used by their container to handle multiple clients' requests, (ii) ***Entity Beans,*** which are the persistent enterprise beans that represent the collections of data and encapsulate operations on the data they represent. For example, rows of tables in a relational database, and (iii) ***Message-driven Beans,*** which are the enterprise beans that receive and process messages asynchronously. A message-driven bean typically works as a JMS message listener, which receives JMS messages instead of events. These messages may be originated by either an application client or another enterprise bean.

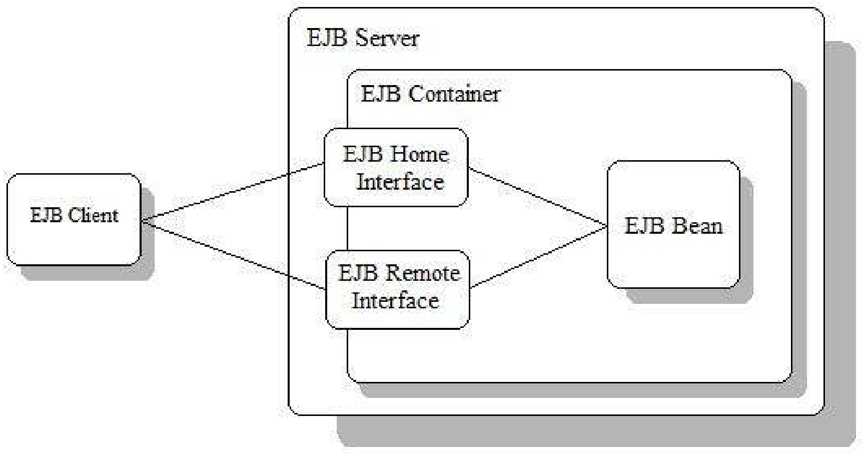


Figure 2: EJB Architecture

III.2 Overview of Model-Driven Engineering

Model-Driven Engineering (MDE) is a technique used for software development that primarily focuses on models instead of programs as first-class entities for development. MDE emphasizes on raising the level of *abstraction* and the need to have useful models that can be manipulated automatically by programs, thus increasing *automation* in software development. To make these models useful and increase the level of abstraction, it is necessary to define these models completely and formally at different levels of abstraction for developing systems. These definitions are created using metamodels. Based on these metamodels the executable model transformations are implemented that increase automation in software development by automatically composing, refining, and reversing or refactoring models. The key elements of MDE approach includes:

- ***DSML:*** This enables developers to model meaningful applications within the application domain it abstracts.

- ***Metamodeling:*** This involves the analysis, construction, and development of the key characteristics, rules, constraints, and models related to DSMLs and for the purpose of modeling a predefined class of problems within a particular domain. It is the process of designing languages through meta and meta-meta notations.
- ***Model Transformation:*** This enables developers to automate and ensure the consistency of software implementations via analysis information and requirements captured in the models of domain-specific structure and behavior.

During the life span of computing, consistent efforts have been made at developing higher-level platform and language abstractions. MDE technologies yield such higher-level abstractions in software development by focusing on architecture and corresponding automations. For example, DSMLs, developed using metamodeling, specify the domain's semantics and syntax more accurately. This increased abstraction and automation promotes a simpler software development process (i.e., using models) with a greater focus on problem space and thus ensures that the user needs are satisfied by the software system. Moreover, MDE tools allow developers to perform model checking by enforcing constraints and identify and avoid many errors early in the software development process. Furthermore, when developers apply MDE tools to model large-scale systems containing thousands of elements, they can quickly examine several design alternatives, and identify and evaluate various compatible configurations.

III.3 DSMLs for Component Middleware Technologies

The essential idea of MDE is to shift the attention from program code to models. This way models become the primary development artifacts that are created with the particular DSMLs. A DSML formalizes the application structure, behavior, and requirements within particular domains, such as avionics mission computing, online financial services, or even the domain of middleware platforms. DSMLs are described using metamodels, which define the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these domain concepts. Developers use DSMLs to build applications using elements of the type systems captured by metamodels and express design intent declaratively rather than imperatively. DSMLs facilitate the model-based design, development, and analysis of *vertical application domains*, such as industrial process control and telecommunications. They are also applicable to *horizontal application domains*, such as component middleware for DRE systems - which provide the infrastructure for many vertical application domains. Regardless of whether the DSMLs target vertical or horizontal domains, model interpreters can be used to generate various artifacts (such as code and metadata descriptors), which can be integrated with component frameworks to form executable applications and/or simulations. For example, DSMLs for horizontal platform include PICML, which facilitates the development of QoS-enabled component-based DRE systems, and J2EEML, which facilitates the development of EJB applications.

III.3.1 PICML

PICML is a DSML, defined as a metamodel using *Generic Modeling Environment* (GME) [13], to support development of DRE systems. PICML is defined for describing components, types of allowed interconnections between components, and types of component metadata for deployment. Using GME tools, the PICML metamodel can be compiled into a *modeling paradigm*, which defines a domain-specific modeling environment. From this metamodel, the metamodel interpreters generates ~20,000 lines of C++ code representing the modeling language elements as equivalent C++ types. The generated code allows manipulation of modeling elements, *i.e.*, instances of language types using C++, and forms the basis for wiring *model interpreters*, that traverse the model hierarchy to perform various kinds of generative actions, such as generating XML-based deployment descriptors. These descriptors are based on the OMG Deployment and Configuration Specification [16] and include: *component interface descriptor*, which describes a single component's interfaces, ports, and attributes; *implementation artifact descriptor*, which describes a single component's implementation artifacts; *component implementation descriptor*, which describes a specific implementation of a component interface and also contains component interconnection information; *component package descriptor*, which describes a single component's multiple alternative implementations; *package configuration descriptor*, which describes a component package configured for a particular requirement; *component deployment plan*, which describes the plan that guides the runtime deployment; and *component domain descriptor*, which describes the deployment target – the nodes and networks – on which the components are to be deployed.

III.3.2 J2EEML

J2EEML is a DSML that formally captures the design of EJB systems, their QoS requirements, and the autonomic adaptation strategies of their EJBs. J2EEML constraint checkers help ensure that autonomic applications are constructed correctly and its models capture autonomic properties and reduce the design and implementation complexity of autonomic systems. The key aspect of J2EEML is the formal mapping from QoS requirements to application components. The formal mapping allows developers to address several design challenges. For example, developers can clearly understand which components to monitor in the application since they can visualize the relationships between components and QoS goals. This understanding facilitates intelligent decisions about what to monitor and where monitoring logic should reside. Developers can also design hierarchical QoS goals to divide and conquer complex QoS analyses, which provide the ability to understand what type of analysis engine to choose and the ability to understand how to decompose the analysis engine into layers. Developers can also associate adaptation plans with each QoS goal to design the planning aspects of the autonomic application and aid in choosing a single-layer or multi-layered planning architecture and in specifying the actions that the autonomic layer is responsible for choosing from in the event of a QoS failure.

CHAPTER IV

AUTOMATED AND SIMPLIFIED MODEL MIGRATION AND DSML REUSE

Although DSMLs address many challenges including, complexity, level of abstraction, reusability, flexibility and many more in developing DRE systems relative to component middleware technologies, unresolved challenges remain. In this chapter we describe the motivational application scenario in the context of the simplification and reusability of component middleware DSMLs using the examples of PICML and J2EEML that also help in model migration. We also describe the open issues in the scenario which are remaining to be resolved.

IV.1 Motivational Application Scenario

The motivation for designing a tool for reusing and simplifying the component middleware DSMLs during the application development process comes from the non-intuitive and non-reusable nature of traditional approaches. Furthermore, these traditional approaches could be error-prone, complex and tedious, as they are usually attempted manually, with respect to the large application domains and could cause large performance overheads. There are many scenarios possible in which the simplification and reusability of component middleware DSMLs can play a major role in the application development process. Out of these many scenarios, to describe the problems with the present DSMLs for component middleware technologies, we choose the main scenarios as follows:

IV.1.1 Developing Application Models using Current DSMLs

Over the past few decades, software developers and researchers have been creating abstraction and reusability that help them to simplify the program development process and shield them from the complexities of this environment. For example, early programming languages and operating systems (OS), such as assembly language, Unix OS abstract the complexities involved in programming in machine code directly to hardware. Despite this maturation of third-generation languages, several challenges remain. Of these problems, the primary problem is the growth of platform complexities, which has evolved faster than the ability of general-purpose languages to overcome it. For example, popular middleware platforms, such as CORBA and J2EE have hundreds of classes and methods with many dependencies and side effects that require huge amount of effort to program and run properly. Furthermore, most application and platform code is still written and maintained manually using third-generation languages, which bring upon excessive time and effort and complexity. For example, it is hard to write Java or C# code correctly and efficiently for large-scale distributed systems with thousands of interconnected software components.

To address this platform complexity and the lack in third-generation languages to alleviate the complexity and efficiently develop the large-scale distributed systems, software developers and researchers rely on MDE technologies. The main component of MDE technologies is DSMLs, which are described using metamodels, which define the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these domain concepts. These DSMLs for the component middleware technologies, considerably enhances the application development process by addressing several challenges including level of abstraction, reusability, and automation. Although the imporovements are huge, these component middleware DSMLs are still complex enough that the developers have to put a considerable amount of time and efforts even for a simple application. For example, Figure … show the DSML of CCM called PICML.

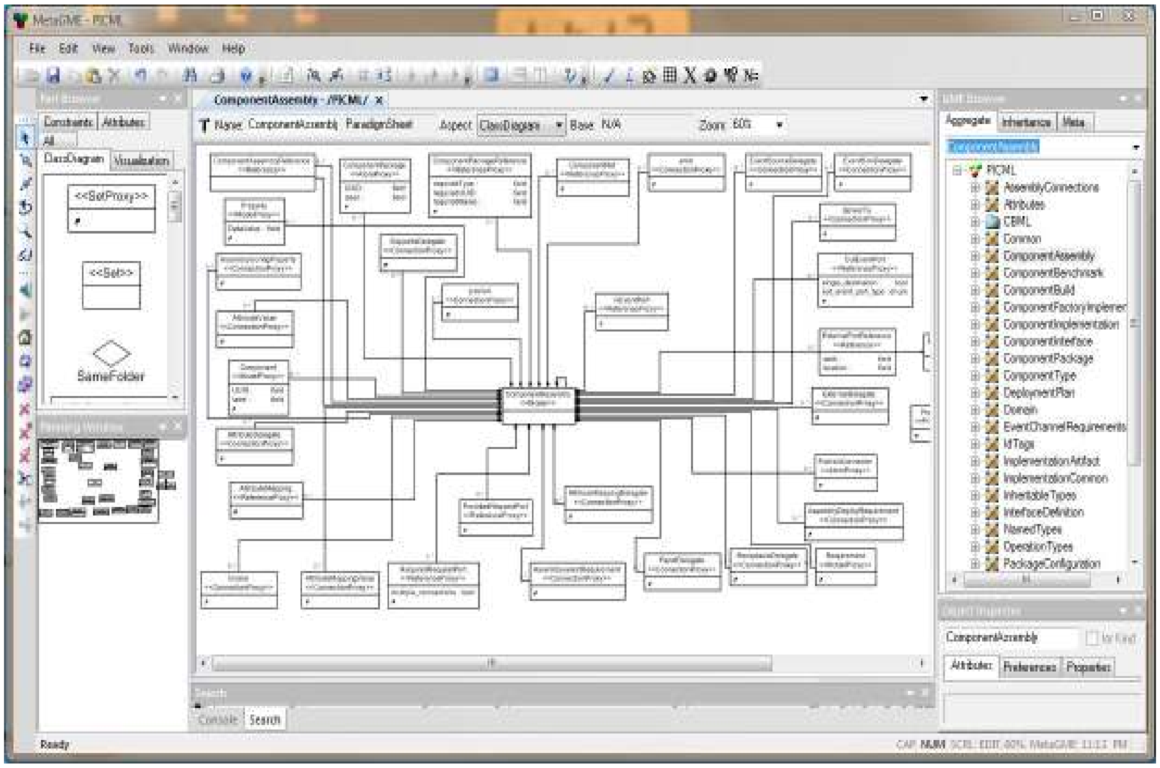


Figure 3: PICML Modeling Language

PICML is a modeling language for building CCM application models that can be used to generate application code using model transformation and code generation techniques. PICML is a vast modeling language with thousands of features available for building application models, which makes it so complex that even for simple application model developers has to work around through huge number of features. This lack of simplicity may deflect the primary focus of developers from the important features and may result into developing an inefficient and complex application model. Thus it is an important motivation for research to simplify the application model development process while maintaining the flexibility, reusability and efficiency of the component middleware DSMLs.

IV.1.2 Upgrading Component Middleware Technology

Today the world is moving forward at a very fast pace. Every second a new technology is invented. These evolutions in technologies also result in the increased complexity of the platforms and languages. From the early days of computing, abstraction and reusability become the most important aspects of language and platform technologies to reduce complexity. Although these early languages and platforms raised the level of abstraction, they still lack in the reusability of programs and platforms. As a result, they had relatively little impact on commercial software development, focusing primarily on a few domains.

Advances in languages and platforms during the past two decades have raised the level of abstractions and increases the reusability available to developers, thereby alleviating the complexity and reducing programming efforts. For example, languages like C++, Java, or C# instead of FORTRAN or C. Similarly, today's reusable libraries and application framework platforms minimize the need to reinvent common and domain-specific middleware services. Due to these advances of third-generation languages and reusable platforms, software developers are now better equipped to alleviate complexities and efforts associated with application development using earlier technologies. Despite these improvements third-generation languages still lack in the ability to alleviate the complexity of creating component middleware applications. To overcome this deficiency of third-generation languages the methodology of building applications evolved into MDE technologies, which uses DSMLs to create application model to generate the component middleware applications. However, as we have seen from above that technologies often evolve rapidly, these component middleware technologies are also evolving at a fast rate. Researchers keep creating new versions of these technologies. These also require the simultaneous modification of the middleware DSMLs to accommodate the rapid growth of middleware technologies. Since these platforms DSMLs are evolving rapidly, developers expend considerable amount of effort and time by manually porting application models to newer versions of the same platform DSML. Moreover, the application models themselves may also be needed to be upgraded depending upon the client requirements. This upgrade in the application models due to the upgrade in middleware DSMLs or due to client requirements may involve a huge amount changes, For instance, in a given application model a component which occurs hundreds of times in the application at different places or which has hundreds of dependency with other components or even other objects is required to change.

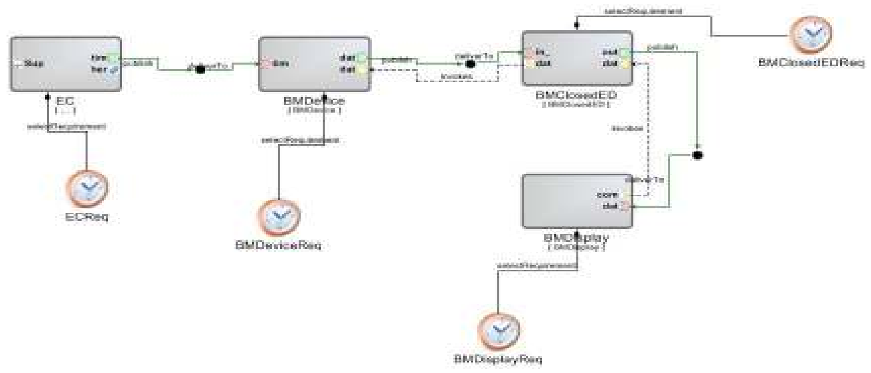


Figure 4: Basic Single Processor Application Model

Figure 3 shown above is the application model of Component Assembly of Basic Signal Processor (BasicSP), which was developed using the paradigm (or DSML) PICML. This component assembly contains 4 components: *EC*, *BMDevice*, *BMClosedED*, and *BMDisplay*. This is a very small part of a small application. These components are the references to the actual components which are defined in the interface definitions folder of the application model. Now if situations occur in which all of this components have to be modified or even change than you have to manually modify them at all the places where they are declared and where they are used, which could be tedious, error-prone and complex if it is a large-scale application model with thousands of components. Similarly, if the paradigm itself is modified and updated, then the whole application model needs to be changed to become compatible with the new updated paradigm. In many situations even the developers will choose to develop the application model right from scratch instead of modifying the existing one. As a result, the process of upgrading the application models will require developer's excessive time and efforts in the modification of complex large-scale application model.

IV.1.3 Migration of Component Middleware DSMLs

It is always a topic of competition for the developers to choose the language for developing the application. Right from the early days of computing, the process of choosing the application development language becomes the most vital part of the analysis process of the software development life cycle. Every time there is a new project to be developed, there is a long evaluation period where one decides what technology to use. There are so many pros and cons of all these languages that it becomes the important aspect to decide what language to use. For example, developing a web application can be done using .NET or using Java, or using PHP. But to decide the right language to develop the particular web application needs a well performed evaluation. Table 1 below is a comparison chart between .NET, Java, and PHP, showing the key differences between these languages.

Table 1: Comparison chart between .NET, Java, and PHP

Feature	**.NET**	**Java**	**PHP**
Compiled Code – Increases website speed (precompiled is the fastest)	Yes – both precompiled and dynamically compiled when a page is requested	Yes – both precompiled and dynamically compiled when a page is requested	No – a 3rd party accelerator can be used to increase performance but it is not installed on most shared hosting servers
Scripted Language – results in poor website performance	No	No	Yes – a 3rd party accelerator can be used to increase performance but it is not installed on most shared hosting servers
Object Oriented – Increases the ability for code reuse and provides enhanced features as well as reduced development time; since code is more reusable, results in fewer bugs that can be discovered by any client and fixed for everyone; encourages developers to write more maintainable code	Yes	Yes	No
Supported Development Languages – easier to find developers	C++, C#, Visual Basic.NET, Jscript.NET, Python, Perl, Java(J#), COBOL, Eiffel, Delphi – 25	Java	PHP
Browser Specific HTML Rendering – different HTML is automatically sent to IE than to Netscape, reducing incompatibility issues	Yes	No	No
Open Source	No	Yes	Yes

Advances in these languages lead the developers to build the application with increased performance and more advanced features. For example, the precompiled and dynamically compiled code increases considerably the performance of the web applications developed using .NET or Java. Due to this advances of third-generation languages software developers make a best choice of the language most of time. Unfortunately, this is not the case all the time. For example, at a later stage of application development, situation may arise that it would have been more beneficial if the application was developed using other language. In this situation the developer might compromise the benefits of using other language or might think of re-developing the application using the other language. This required the application to be converted into another language, which is often cumbersome and error prone.

Software developers and researchers over the past few decades put a lot of effort to increase the reusability available for programming, thereby reducing the programming efforts. For example, API's and libraries to a great extent reduce the effort associated with application development by minimizing the need to re-create the common services. Despite these efforts, the problem remains due to the growth of platform complexities, which has evolved faster than the ability of general purpose languages to overcome it. For example, component middleware application platforms, such as EJB and CCM have thousands of methods and classes with many dependencies and side effects that require huge amount of effort to program. This increasing complexity has driven the need for reusability in the design and development of the software systems. This has led to the adoption of the traditional engineering practice of *modeling* into software engineering. Model-Based Software Engineering (MBSE) is a software development methodology that places emphasis on the formal understanding of the features and structure of a product family, by creating and using reusable models, thus reducing the application development efforts. This methodology has a set of abstractions, termed as DSML, which can be used to describe an entire class of systems. DSML become one of the major tool for developing complex component middleware applications as they significantly reduces the programming efforts by allowing developers to use the reusable metamodels to build applications using elements of the type systems captured by metamodels and express design intent declaratively rather than imperatively. However, as we know the component middleware technologies are evolving at a considerable fast rate, the newer platforms appearing regularly. Although the reusability feature of DSMLs for the component middleware technologies considerably enhances the application development process and reduces the development efforts, they are not reusable across these multiple platforms. For example, an application model in one platform DSML may need to be converted into another platform DSML for reasons including, client requirements, incorporating capabilities specific to a particular platform DSML, etc. This issue of *model transformation* usually done by creating the application either from scratch or writing application specific model transformation rules, which require developers to expend considerable effort.

IV.2 Open Issues in the Reusability of Component Middleware DSMLs

Although the emergence of DSMLs for the component middleware technologies greatly reduces the application development efforts by addressing the challenges involving simplification, abstraction and reusability, software developers still spend an excessive amount of time and effort in developing application models using complex languages and converting existing application models to make it compatible in the following situations: a) during the upgrade of the component middleware DSMLs to create newer version, and b) during the transformation of application model from one component middleware DSML to another. For instance, J2EEML and PICML, which are the DSMLs for CCM and EJB comprises of hundreds and thousands of features. These require developers to spend considerable amount time and effort to create application models. Also, different version of J2EE platform are available and to make the J2EEML (DSML for J2EE platform) compatible to the latest version of J2EE, it also needed to be upgraded. Similarly, many of the software industries are now considering Web Sphere instead of CCM for middleware applications, which require them to convert the existing application from CCM to Web Sphere to use them with the new applications. As this component middleware DSMLs are not reusable in the above two situations, the issue of conversion usually done by creating the application model either from scratch or by writing application specific model transformation rules. In both the cases the approach will be tedious, complex, error-prone, and technology specific for hundred or thousand of large-scale applications.

CHAPTER V

GENERALIZATION AND STEP-WISE REFINEMENT

In this chapter we describe our research approach and the mechanisms developed to simplify and automate the development and migration of component middleware application models. This include the Commonality/Variability Analysis approach in which we analysis the J2EEML and PICML to determine the features that are common and variable in them. We also describe the step-wise refinement approach, discussed by Dr. Don Batory, to simplify the application development process. Based on this approaches we describe the modeling details of GCML, the graphical user interface and the model interpreters used in our research approach to enhance the reusability, simplification, and automation of component middleware DSMLs while developing and migrating to newer version of same platform or across multiple middleware platforms.

V.1 Commonality/Variability Analysis

Increasingly, software engineers spent their time creating *software families* consisting of similar systems with many variations. They search or the right decomposition of their software into modules or classes, but have limited guidance in finding those decompositions, especially in the face of constraints on performance, reliability, and ease of use. Commonality and variability analysis (CVA) gives software engineers a systematic way of thinking about and identifying the product family they are creating. Among other things, it helps developers

- Create a design that contributes to reuse and ease of change by increasing the level of abstraction
- Predict how a design might fail or succeed as it evolves, and
- Identify opportunities for automating the creation of family members

When commonalities are invariant and variabilities precisely defined, developers create opportunities for high-payoff automation to simplify and improve the application development process. Based of this CVA approach we capture the key characteristics of the two platform specific DSMLs (i.e., J2EEML and PICML). As shown in the Table 2 below, between these two DSMLs we determine the commonalities, which describe the attributes that are common in them and variabilities, which describe the attributes that are unique in them.

Table 2: Commonality/Variability Analysis of PICML and J2EEML

Features	**PICML**	**J2EEML**
Component	✓	✓
Component Assembly	✓	✓
Component Interaction	✓	✓
Factory Operations	✓	✓
Lookup Operations	✓	✓
Ports	✓	
Attributes	✓	
File/Package	✓	
Different types of component interactions	✓	
Folders	✓	
Post Create Factory Operation		✓
Attributes of the Factory and Lookup Operation		✓
Different types of components		✓

V.2 Step-wise Refinement

Step-wise refinement is a powerful paradigm for developing a complex program from a simple program by adding features incrementally where a feature is a product characteristic that is used in distinguishing programs within a family of related programs. The concept of "step-wise refinement" is to take an object and move it from a general perspective to a precise level of details. But to do so, it has been realized that it cannot simply go from the general to the specific in one felled swoop, but instead, in increments (steps). Step-wise refinement is the top-down presentation of a software system's functionality as a sequence of layers of increasing detail, beginning with the very abstract and ending with the very concrete, and with each layer an incremental refinement of the previous one. The refinement framework is in practice provides a framework for encouraging correct, accountable and even efficient application. As refinements reify levels of abstraction, feature refinements are often called layers – a name that is visually reinforced by their vertical stratification. The advantage of Step-wise refinement is that it allows for incremental development but on a much finer level of granularity. It also uses unit tests as an integral feature of the development process. The software is also rapidly build as step-wise refinement lends itself naturally to producing working prototypes of the software as it develops, and it is often possible to build prototypes in remarkably short periods of time. Step-wise refinement is highly scalable, as large systems can developed in a structures and predictable fashion from it.

In our research, we describe the step wise refinement approach by using the middleware application DSMLs, J2EEML and PICML. We used step wise refinement by considering the two aspects of the middleware application development process. Firstly, the process of developing the application model from scratch and secondly, transforming an existing application model from one middleware application DSML to another. In the first aspect, we use 3 steps in the step-wise refinement: 1) Using the CVA mechanism we determine the attributes that are common in both J2EEML and PICML. Based on this analysis we build an abstract DSML at a very high level of abstraction called GCML. GCML is a generalized DSML that allows the developer to define the component modeling features at a very high level of abstraction and that are common to both J2EEML and PICML. As these features are common to both J2EEML and PICML, they can be reuse to generate the platform specific model for both of them. 2) In the second incremental step we refine the abstract model created using GCML by adding more features and attributes that are associated with the features that are selected in the abstract model. To add these features we use the GUI which is a *human-computer interface* for third-generation programming languages that uses *windows, icons* and *menus* and which can be manipulated by a mouse. Advantages of GUI includes, intuitiveness by making it easier to learn and to use and providing users with immediate, visual feedback about the effect of each action, increases the level of abstraction and enhance the efficiency and ease of use for the underlying logical design. 3) Once the platform specific features are selected in the GUI, the next step involve the automatic generation of the platform specific application model. This generated model is again refined, using the platform specific paradigm of J2EEML or PICML, by adding application specific features. This will make ready the application model to generate the executable application that can be deployed.

In the second aspect of application development process in which an existing application model is transform from one middleware application DSML into another, we use 4 steps in the step-wise refinement: 1) In the first step, we generate the abstract model, which is compatible with GCML, from an existing platform specific application model. In our case it could be a J2EEML application model or PICML application model. This reverse generation of the abstract model from an existing application model is done automatically using the model transformation capability of the model-driven engineering. Once the abstract model is generated the remaining 3 steps are same as discussed above in the first aspect of application development process to generate the application model of either J2EEML or PICML.

V.3 Generic Component Modeling Language

Generic Component Modeling Language (GCML) is a DSML defined at a very high level of abstraction and enables the developers to define the component modeling features that are common to both J2EEML and PICML and reuse that abstract model to generate the platform specific model which is compatible to either J2EEML or PICML. Figure 4 shows the metamodel of the GCML that describe the common component features of both PICML and J2EEML.

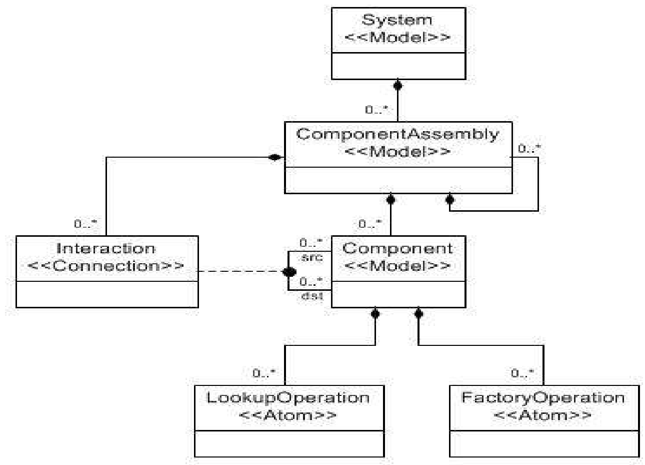


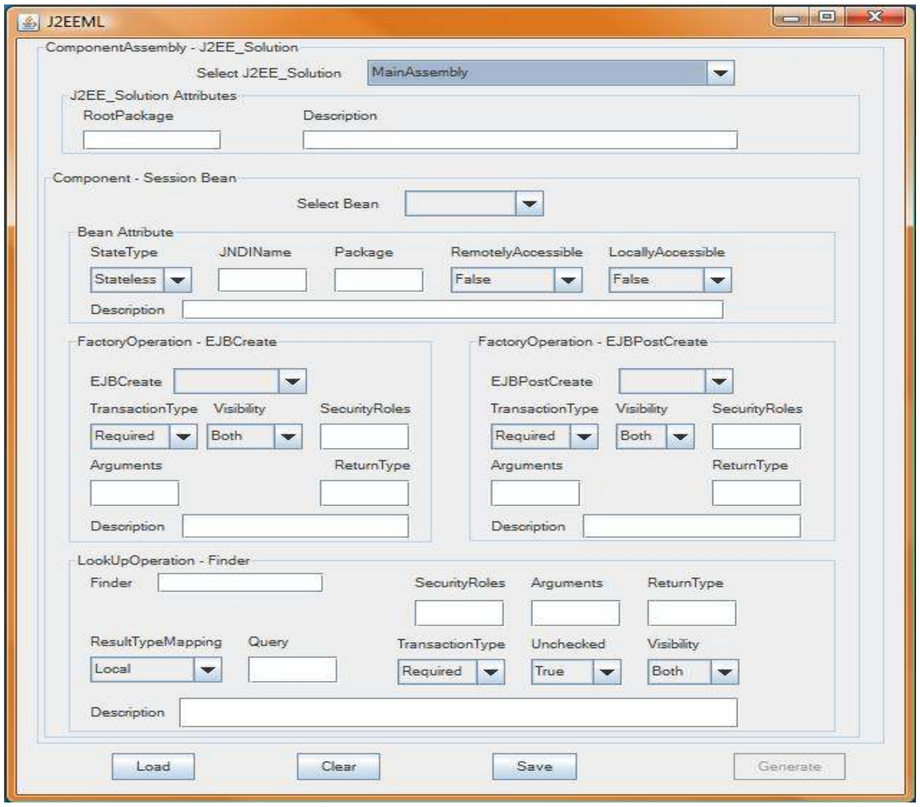
Figure 5: Generic Component Modeling Language (GCML)

The key characteristics of the platform specific DSMLs that we captured in GCML include: *System*, which is the root model that we build using the GCML paradigm. This model is the base model which will be transformed into the platform specific application model. *Component Assembly*, which is an abstraction for composing components into larger reusable entities. A component assembly typically includes a number of components connected together in an application-specific fashion. Unlike the other entities, there is no runtime entity corresponding to a component assembly. This component assembly has a cardinality of 0..*, which means that a system model can have n number of component assemblies. The J2EEML feature corresponding to the component assembly of GCML is named as *J2EE_Solution*. *Components* are the centerpieces of the component applications. These Components separate application logic from the underlying middleware infrastructure. A Component's main function is to tie together and organize the features of the objects and other types it uses. The cardinality of component is also 0..*. The J2EEML feature corresponding to the component of GCML is named as Bean. In J2EEML this bean is further refined into two types of beans as: *Session Bean* and *Entity Bean*. This component of GCML supports only the session bean of the J2EEML. *FactoryOperation* is a type of operation that creates something and returns it. In this context, this FactoryOperation in GCML creates the instances of the components and the implicit return type is the type of the component in which they are defined. A component may contain any number of FactoryOperations. *LookupOperation* is optionally found in the component. The corresponding feature of FactoryOperation of GCML in J2EEML is a combination of *EJBCreate* and *EJBPostCreate* features. A LookupOperation is intended to function in an application by looking up the component (in which it is defined) in a database or repository. In J2EEML the corresponding feature of LookupOperation of GCML is named as *finder*. And *Interaction*, which are the connection between the components. These connections are made to indicate component-to-component interactions. In J2EEML this interactions are indicate bean-to-bean interactions. Beans can have any number of interactions between them. In PICML this interactions are more specific. It can be publish/deliverTo interaction or an invoke interaction between components. The cardinality of these interactions also varies in J2EEML and PICML.

V.4. Graphical User Interface

GUI is a computer environment that simplifies the user's interaction with the computer by representing programs, commands, files, and other options as visual elements, such as icons, pull-down menus, buttons, windows, and dialog boxes. Advantages of GUI include: it provides a standard method for performing a given task each time the user requests that option, rather than crating a set of commands unique to each potential request. GUI also allow users to take full advantage of the powerful *multitasking* (the ability for multiple programs and /or multiple instances of single program to run simultaneously) capabilities of *operating systems* which result in increase in the flexibility of GUI use and consequent rise in user's productivity. The major advantage of GUI is in increasing the level of abstraction while enhancing the efficiency and ease of use for the underlying logical design.

We have developed the GUIs as our second step in the step-wise refinement technique for both J2EEML and PICML to add the platform specific features associated with the features that are selected in the abstract model of GCML. The features included in the GUI of J2EEML and PICML are based on the variability analysis of the CVA approach as shown in Table 2. The GUI is designed in such a manner that it allows the user to select the features in a hierarchical manner. For instance, in the beginning the user will be able to select the top most assembly from the drop-down list of the assemblies. Once the user made his selection, he can select the attributes of that assembly, features which are present in that assembly and attributes of those features, and features of the features of assembly and their attributes. This hierarchical fashion is used to simplify this step of refinement in the application development process and reduces the development efforts. Figure 5 shows the GUI for J2EEML and Figure 6 shows the GUI for PICML.



Select J2EE_Solution: MainAssembly

StateType: Stateless
RemotelyAccessible: False
LocallyAccessible: False

TransactionType: Required
Visibility: Both
SecurityRoles

TransactionType: Required
Visibility: Both
SecurityRoles

ResultTypeMapping: Local
TransactionType: Required
Unchecked: True
Visibility: Both

Figure 6: GUI for J2EEML-Specific Features

In the GUI of J2EEML we capture the features as: all the assemblies, which represent the *J2EE_Solution* in J2EEML, in order of their hierarchy, are captured in the combo-box with label *"Select_J2EE_Solution"*. The two text-boxes "*RootPackage"* and *"Description"* under the J2EE_Solution combo box represent the attributes for the J2EE_Solutions. This J2EE_Solution and its attributes are included in the *ComponentAssembly – J2EE_Solution* panel. For each assembly, in the *Component – SessionBean* panel the user is allowed to select a *Bean* captured corresponding to the component and the attributes of the bean represented under the *Select_Bean* combo box in the *Bean Attribute* panel. Next in the hierarchy are the three features, *EJBCreate, EJBPostCreate,* and *Finder,* that a user can select for each bean. There can any number of these three features a bean can contain. The user can provide values for the attributes of each of these three features represented in their corresponding panels. Finally, we have 4 buttons in the bottom and their function, as the name suggest, is defined as: *Clear*, which allow the user to clear all the entries he made in the GUI. *Save,* which allows user to save all the entries in a text file, which we called as configuration file, to persist the data. This will reduce the development effort if user wants to regenerate a model with minimum change in some of the features in the GUI. *Load* allows the user to load the configuration file and fill all the entries automatically in the GUI. *Generate,* which allows user to generate the J2EEML application model compatible to the J2EEML paradigm.

In the GUI of PICML we capture the features as: all the assemblies, which represent the *ComponentAssembly* in PICML, in order of their hierarchy, are captured in the combo-box with label *"Select_Assembly"*. The two text-boxes for the name of *"ComponentImplementationFolder"* and *"ComponentImplementationContainer"* with the Component Assembly combo box represent the Component Implementation Folder which contain the Component Implementation Container which is the parent of the Component Assembly in the generated PICML model.

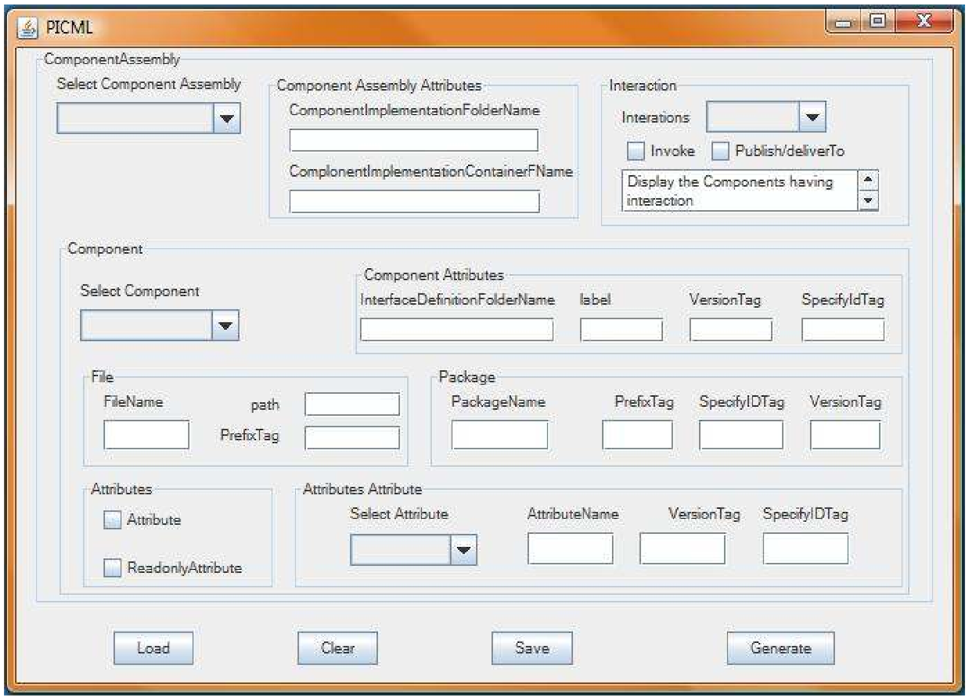


Figure 7: GUI for PICML-specific features

For each assembly, in the *Component* panel the user is allowed to select a Component and the attributes of the Component represented in the *ComponentAttributes* panel. Next in the hierarchy are the three features, *File, Package,* and *Attributes,* that a user can select for each Component. The File allows the user to enter the name of the file which will contain the Package as enter by the user in the PackageName text-box. This Package will contain the Component, selected in the Component combo-box, in the generated PICML model. The user can provide values for the attributes of each of these three features represented in their corresponding panels. Finally, we have 4 buttons in the bottom and their function, as the name suggest, is defined as: *Clear*, which allow the user to clear all the entries he made in the GUI. *Save,* which allows user to save all the entries in a text file, which we called as configuration file, to persist the data. This will reduce the development effort if user wants to regenerate a model with minimum change in some of the features in the GUI. *Load* allows the user to load the configuration file and fill all the entries automatically in the GUI. *Generate,* which allows user to generate the PICML application model compatible to the PICML paradigm.

V.5 Interpreters

GME is a generic, configurable modeling environment. For some GME applications, the only motivation for a modeling project is the desire to describe a system in a structured way. Usually, however, we also want the computer to be able to process data from the model automatically. Typical processing tasks range from the simple to the sophisticated: 1) generating program code or system configuration. 2) Building models automatically from information provided by another data source (e.g. a model). 3) Using the models as a data exchange formats to integrate tools that are incompotible with each other. A common theme for all these applications is that they require programmatic access to the GME model information. To meet this requirement, GME provides several ways to create programs that access its data. The most popular technique is writing a GME interpreter.

Interpreters are not standalone programs; they are components (usually DLLs) that are loaded and executed by GME upon a user's request. In our research work we have used Java Component to write my interpreters. As discuss above in the step-wise refinement section that we consider the two aspects of the application development process. For the same reason we write interpreters for both this aspects. In the first aspect, our interpreter initially asks the user to select the platform (i.e., J2EEML or PICML) in which he/she want to generate the resulting model. After this our interpreter reads the abstract model created based on GCML and fill that data into the GUI of the appropriate platform and open that GUI. Behind the scene, we use TreeMap are the data structure to store and manipulate the data. Therefore, the interpreter reads the data from the abstract model and the GUI and stores them in the TreeMaps of different objects. As mentioned above our interpreter also has the functionality for the user to load the configuration text file and fill the data in the GUI, to save the data from the GUI into the configuration text file, and to generate the output application model using the configuration text file. Once this model is generated it could be open using the platform specific paradigm to which it is compatible.

In the second aspect of the application development process, we have second interpreter which allow the user to generate the abstract model from an existing application model. This interpreter is attached to both, J2EEML or PICML paradigm, so that the user will be able to generate the abstract model from the existing application model of either J2EEML or PICML. In this aspect the interpreter reads the application model for all the common features, as described in the Table 2 of Commonality/variability analysis, and based on that generate the abstract model which is compatible to the GCML paradigm. Finally, this generated abstract model can be used to generate the application model in any of the two platform paradigms using our first interpreter.

CHAPTER VI

CASE STUDY

Over the past decades many DSMLs and associated tools has developed for a wide range of modeling concerns, specially platform specific as well as platform independent component structural middleware technologies, such as PICML [5] for CORBA Component Model, J2EEML [6] for Enterprise JavaBeans, and Embedded Systems Modeling Language ESML [17] for embedded systems. It still needed constantly to develop DSMLs for new domains. To show-case the application development efforts and complexity of reusing DSMLs and DSML transformation for new requirement sets, we provide a case study based on our research approach. In this case study we have focused on DSMLs developed using GME since GCML is also developed using GME. However, the concept behind our approach can be applied in other tool environments. We chose the following DRE system as the application scenario for our experiments:

BasicSP – The Basic Single Processor (BasicSP) [18] is a scenario from the Boeing Bold Stroke component avionics computing product line [19]. BasicSP uses a publish/subscribe service for event-based communication among its components, and has been develop and configured using a QoS-enabled component middleware platform. The application is deployed using a single deployment plan on two physical nodes. A Global Positioning System (GPS) device sends out periodic position updates to a GUI display that presents these updates to a pilot. The desired data request and the display frequencies are fixed at 20 Hz. The scenario shown in Figure 7 begins with the *GPS* component being invoked by the *Timer* component. On receiving a pulse event from the *Timer,* the *GPS* component generates its data and issues a data available event. The *Airframe* component

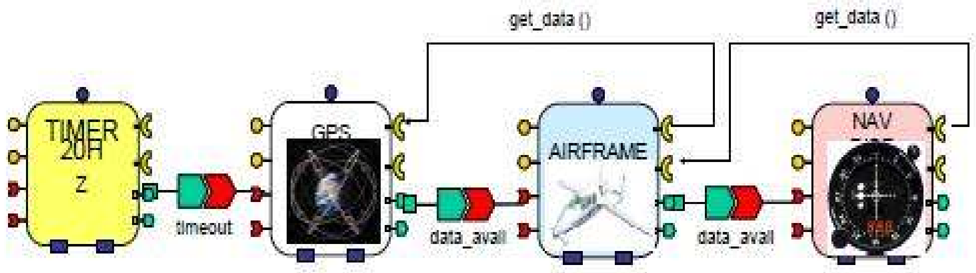


Figure 8: Basic Single Processor

retrieves the data from the *GPS* component, updates its state and issues a data available event. Finally, the *NavDisplay* component retrieves the data from the *Airframe* and updates its state and displays it to the pilot.

The configuration complexity of the application scenario can be represented using 3-tuple {C;I;D} where, 1) *C* defines the number of components in the application. 2) *I* defines distinct number of interactions between components of the application. An interaction exists between two components if the outgoing port of one is connected to incoming port of the other. And 3) *D* defines the distinct number of dependencies between components of the application. A dependency exists between two components if a change in the QoS configuration of one necessitates a change in configuration of the other. The level of configuration complexity of BasicSP can be summarized using the 3-tuple definition as shown in Table 3**.** Figure-8 shows model for the BasicSP developed in

Table 3: Complexity of BasicSP application

Application Scenario	# of components	# of component interactions	# of component dependencies
BasicSP	4	5	6

the GME using the PICML paradigm. This BasicSP application model comprises four components, which is an example with very less configuration complexity as shown

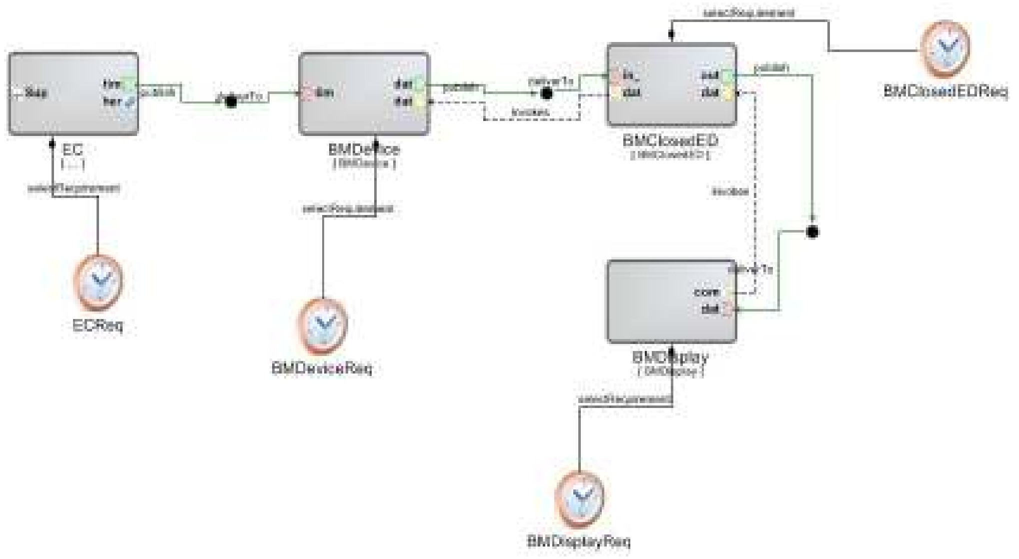


Figure 9: GME model of BasicSP

in the Table 3. Although component middleware and existing MDE tools provide several advantages in software development, several challenges need to be addresses in order to reduce the complexity and development effort. One of the challenges, with respect to complexity and development efforts, involves in the transformation of the BasicSP model, which is developed using the PICML paradigm, into another paradigm, for example, J2EEML paradigm.

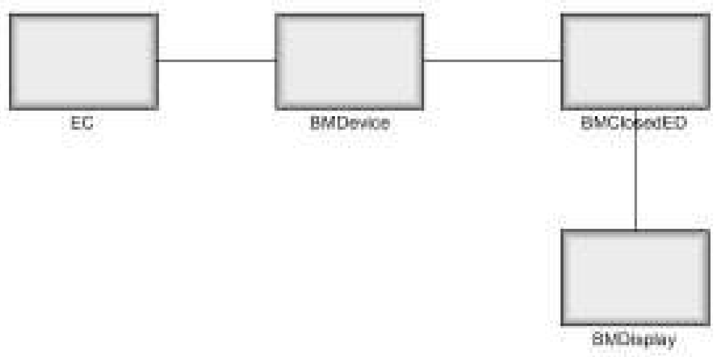


Figure 10: Abstract Model of BasicSP

Our research addresses this challenge by using the step-wise refinement approach. Initially, as discussed in solution approach chapter, using our second interpreter we generated the abstract model (Figure 9) which is compatible with the GCML paradigm. Secondly, using our first interpreter read the abstract model and J2EEML GUI pop-up (as we are developing J2EEML model from the PICML model). This J2EEML GUI will allow the user to provide the values of the J2EEML specific features. Finally, by pressing the generate button the J2EEML application model (Figure 10) will be generated automatically.

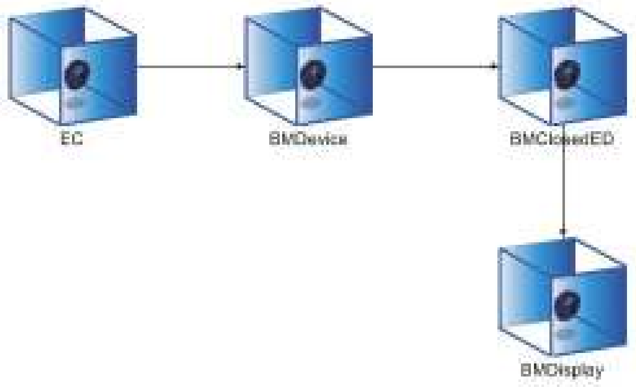


Figure 11: J2EEML Model of BasicSP

As we are using the visual tools and the step-wise refinement technique by creating an abstract model, the transform of the BasicSP application model from PICML paradigm to J2EEML paradigm is done with minimal user interaction and in a simplified manner. Thus reduces the complexity of the application model and the development effort significantly, and yet enhances the reusability in the model-driven engineering.

CHAPTER VII

EXPERIMENTAL RESULTS

In this chapter we discussed the evaluation of our approach's modeling and transformation capabilities in the context of DRE system case study discussed in chapter VI. All the measurements use GME 9.8.28 software package on Windows Vista SP2 workstation. Our prototype implementation of GCML uses PICML and J2EEML paradigms. In order to find the reduction in modeling effort using our approach, we compare its modeling and transformation capabilities with those of traditional approach using the example of BasicSP discussed in the previous chapter.

In order to compare our approach, we use class counts that are created manually to evaluate the modeling effort in using our approach. Class count is an important metric for model-based quantitative software measurements and has been applied and adopted in industrial contexts. For our measurements, while transformation of BasicSP application model of PICML paradigm into J2EEML paradigm or into new version of PICML paradigm, we use the following counts from the (meta) models: 1) number of components created in the application, 2) number of connections created between components of the application, and 3) number of dependencies created between the components of the applications. A comparison of our approach with the traditional approach in terms of the manual creation of class counts given above is tabulated in Table 4**.** In this table all the data are shown in term of number of counts created manually by the user during the transformation process of the BasicSP application mentioned above.

Table 4: Modeling effort in approaches

Approach	**# of components created**	**# of component interactions created**	**% of component dependencies created**
Our Approach	0	0	~50
Traditional Approach	4	5	100

Using this approach, the number of components and the interactions between them created manually are reduced by an average of ~100% while the number of component dependencies created manually are reduced by an average of ~50%, thus results indicate that on an average the modeling effort is reduced by ~75%. Furthermore, the components are known to be the main aspect of the component modeling technologies and the reduction of ~75% in the modeling effort include the ~100% reduction in the effort of creating components manually in the application development process.

These results of our approach show great improvement in the application development process even if the number of components in the BasicSP application example is small. These improvements will increase significantly with the large-scale component modeling applications, such as Magnetospheric Multi-scale (MMS) space mission and Shipboard Computing Environment (SCE). The complexity of these large-scale application scenarios as compare to BasicSP in terms of the manual creation of class counts given above is shown below in the Table 5**.**

Table 5: Complexity of application scenarios

Application Scenarios	**# of components created**	**# of component interactions created**	**# of component dependencies created**
BasicSP	4	5	6
MMS	12	11	43
SCE	150	260	950

CHAPTER VIII

CONCLUSION

In this thesis, the approach presented used to enhance the reusability of model-driven engineering with respect to middleware technologies by incorporating the capabilities of automation and abstraction and reduces the development efforts in the scenario of porting the application models when technology refreshed. To apply our approach we include two main techniques that are developed separately to reduce the application development efforts as: Step-wise refinement [D. Batory et. al.] and Commonality/Variability Analysis [J. Coplien et. al.]. After defining the approach, this thesis presented the case study for Basic Single Processor (BasicSP) example by converting the PICML model of BasicSP into the J2EEML model. The main contributions of this work are listed below:

1. Background research for enhancing the reusability of model-driven engineering and for reducing the application development efforts. This involves:
    a. Model Transformation Techniques.
    b. Enhancing the reusability of Model-driven Engineering
    c. Commonality/Variability Analysis (CVA)
    d. Step-wise Refinement
2. Development of a complete approach for enhancing the reusability of model-driven engineering and for reducing the application development efforts in the

scenario when technology refreshes with respect to middleware technologies. This involves:

   a. Designing a Generic Component Modeling Language (GCML) by applying the CVA technique on the middleware DSMLs.
   b. Designing the graphical user interfaces as the second step of refinements after the abstract model created using GCML.
   c. Developing two interpreters to incorporate the capabilities of automation.
      i. First for generating the application model right from scratch using GCML and GUI.
      ii. Second for converting an existing application model from one middleware DSML to another. This interpreter will generated the abstract model which then will be converted to a different middleware DSML using the first interpreter.

3. This thesis also presented a detailed case study for applying our approach to convert an existing application model from PICML paradigm to the J2EEML paradigm. The experiments conducted in the thesis used the example of BasicSP of the PICML paradigm. The experimental results showed a significant reduction in the development efforts while enhancing the reusability of model-driven engineering when middleware technology refreshed.

REFERENCES

[1]: Object Management Group, *CORBA Component Model Specification,* OMG Document formal/06-04-01, April 2006

[2]: V. Matena and M. Hapner, *Enterprise Java Beans Specification,* Version 1.1, Sun Microsystems, Dec. 1999.

[3]: D. C. Schmidt. Model-Driven Engineering. *IEEE Computer,* pp. 25-31. February'06.

[4]: A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Enviroments. *IEEE Computer,* Nov. 2001.

[5]: 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. *Journal of Computer Systems Science,* 73(2):171-185, 2007.

[6]: J. White, D. C. Schmidt, and A. Gokhale. Simplifying Autonomic Enterprise Java Bean Applications VIA Model-driven Engineering and Simulation. *Journal of Software and System Modeling,* 7(1):3-23, 2008.

[7]: D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering,* 30(6):355-371, June 2004.

[8]: J. Coplien, D. Hoffman, and D. Weiss. Commonality and Variability in Software Engineering. *IEEE Software,* 15(6):37-45, Nov. /Dec. 1998.

[9]: Y. Lin and J. Gray. A Comprehensive Model Transformation Approach to Automatic Model Construction and Evolution. In *20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA),* pp 104-105, San Diego, CA, USA, 2005.

[10]: A. Kavimandan and A. Gokhale. Automated Middleware QoS Configuration Techniques for Distributed Real-time and Embedded Systems. *IEEE Real-Time and Embedded Technology and Application Symposium* 2008: 93-102.

[11]: G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *Journal of Universal Computer Science,* 9(11):1296-1321, 2003. `http:/www.jucs.org/jucs_9_11/on_th_use_of.`

[12]: G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming,* pages 220-242, June 1997.

[13]: The Generic Modeling Environment, Institute of Software Integrated Systems, Vanderbilt University, (www.isis.vanderbilt.edu/Projects/gme/GME reference).

[14]: D. Kaul and A. Gokhale. Middleware Specialization Using Aspect Oriented Programming. In *ACM-SE 44: Proceedings of the 44th Annual Southeast Regional Conference,* pages 319-324, New York, NY, USA, 2006. ACM Press.

[15]: C. Zhang, D. Gao, and H.-A. Jacobsen. Generic Middleware Substrate Through Modelware. In *Proceedings of the 6th International ACM/IFIP/USENIX Middleware Conference,* pages 314-333, Grenoble, France, 2005

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

[17]: 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 Conference,* Los Alamitos, CA, August 2002. IEEE Computer Society.

[18]: K. Balasubramanian, A. S. Krishna, E. Turkay, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt. Applying Model-Driven Development to Distributed Real-time and Embedded Avionics Systems. Invited Paper to *International Journal of Embedded Systems, Special Issue on Design and Verification of Real-time Embedded Software,* Vol. 2, No.3/4, 2006, pp. 142-155.

[19]: D. C. Sharp. Reducing Avionics Software Cost through Component Based Product Line Development. In *Software Product Lines: Experience and Research Directions,* Vol. 576, pages 353-370, Aug 2000.